



UNIVERSITÀ DI PARMA

DEPARTMENT OF MATHEMATICAL, PHYSICAL AND COMPUTER SCIENCES

Master's Degree in Computer Science

TOWARDS STATICALLY REASONING ABOUT R VECTORS

CANDIDATE:
Manuel Di Agostino

ADVISOR:
Prof. Vincenzo Arceri

STUDENT ID:
376511

Contents

Abstract	xii
Acknowledgments	xiii
1 Introduction	1
1.1 Context and Motivation	1
1.1.1 Static Program Analysis and Software Reliability	1
1.1.2 R's Role in Data Science	2
1.1.3 The Central Argument	3
1.2 A Brief History of R	4
1.2.1 Origins in S	4
1.2.2 R's Birth and Philosophy	5
1.2.3 Growth and Current Status	6
1.2.4 Why Static Analysis Never Developed for R	7
1.3 The R Language: Core Features and Challenges	7
1.3.1 Dynamic Typing and Late Binding	8
1.3.2 Further Complexity: Lazy Evaluation, Metaprogramming, and Copy-on-Modify	9
1.3.3 The Vector Data Model	10
1.3.4 Four Challenges	13
1.4 Abstract Interpretation Foundations	13
1.4.1 The Need for Sound Approximations	14
1.4.2 Posets, Lattices, and the Concrete Domain	14
1.4.3 Galois Connections	15
1.4.4 Fixpoint Computation and Widening	16
1.4.5 Abstract Domains: Definition and Examples	17
1.5 Related Work	19
1.5.1 Static Analysis for R	19
1.5.2 Abstract Interpretation for Dynamic Languages	20
1.5.3 Array-Centric Analysis	21
1.6 Research Gap and Contributions	22
1.6.1 The Gap	22
1.6.2 Contributions	22
1.6.3 Scope and Limitations	23
1.7 Thesis Structure	23
2 Concrete Semantics of R Vectors	25
2.1 Syntax of μR Expressions	25
2.2 Syntax and Semantics of μR Statements	28
2.2.1 Statement Syntax	28
2.2.2 Concrete Statement Semantics	29
2.3 Formal Vector Representation	30
2.4 Type Coercion	31

2.5	Generic Functions and Class Dispatch	31
2.6	Attribute Manipulation	32
2.7	Flattening and Concatenation	32
2.8	Recycling	33
2.9	Selection	34
2.9.1	Positive Indexing	34
2.9.2	Negative Indexing	35
2.9.3	Logical Indexing	35
2.10	Update	36
2.10.1	Positive Indexing	36
2.10.2	Negative Indexing	37
2.10.3	Logical Indexing	38
2.11	Chapter Summary	39
2.11.1	Abstraction Challenges	39
3	The Abstract Domain: Definition and Lattice Structure	41
3.1	The Concrete Value Domain	41
3.2	Prerequisites: The Component Domains	42
3.2.1	Length: Positive Integer Intervals	42
3.2.2	Content: Parametric Abstract Domain	42
3.2.3	Attributes: Must/May Pairs	42
3.3	Design Rationale	43
3.4	The Abstract Vector Domain	44
3.5	Order Structure	47
3.6	Join and Meet	49
3.7	Widening and Termination	50
3.8	Concretization and Galois Connection	53
3.9	Chapter Summary	54
4	Abstract Operators	55
4.1	Abstract Coercion	55
4.2	Abstract Attribute Manipulation	56
4.3	Helper Functions	57
4.3.1	Squash and SquashExcept	57
4.3.2	Abstract Concatenation $\oplus^\#$	57
4.4	Abstract Recycling	58
4.4.1	Helpers	58
4.4.2	Abstract Recycling Recycle $^\#$	60
4.4.3	Soundness of Abstract Recycling	61
4.5	Helper Functions	61
4.5.1	Abstract Cardinality and Enumerability	61
4.5.2	Zero Propagation: Propagate	62
4.5.3	Selector Compression: AdjustForZeros	63
4.5.4	Abstract Element Access	64
4.6	Abstract Selection	65
4.6.1	Positive Selection	66
4.6.2	Negative Selection	67
4.6.3	Logical Selection	69
4.7	Abstract Update	71

4.7.1 Positive Update	72
4.7.2 Negative Update	76
4.7.3 Logical Update	78
4.8 Chapter Summary	79
5 From Abstract Operators	
to Program Analysis	81
5.1 Abstract Expression Semantics	81
5.2 Abstract Statement Semantics	83
5.3 Realization in flowR	83
5.3.1 Analysis Pipeline	84
5.3.2 The Vector Inference Visitor	84
5.4 Domain Instantiation	84
5.4.1 Scalar Domain	85
5.4.2 Domain Components	85
5.5 Validation	85
5.5.1 Methodology	85
5.5.2 Test Corpus	86
5.5.3 Results and Discussion	87
5.5.4 Limitations	88
6 Conclusions	90
6.1 Summary of Results	90
6.2 Limitations	91
6.3 Future Work	92
A. Formal Proofs: Domain Structure	94
A.1 Equivalence Relation and Minimum Representative	94
A.2 Partial Order	95
A.3 Meet Preservation	96
A.4 Widening	99
B. Formal Proofs: Abstract Operators	102
B.1 Soundness of Abstract Recycling	102
B.2 Soundness of Abstract Positive Selection	108
Bibliography	110

List of Definitions

Definition 2.7.1	Concrete Concatenation	32
Definition 2.8.1	Concrete Recycling	33
Definition 2.8.2	Data Cycling	33
Definition 2.9.1	Positive Selection	34
Definition 2.9.2	Negative Selection	35
Definition 2.9.3	Logical Selection	35
Definition 2.10.1	Positive Update	37
Definition 2.10.2	Negative Update	37
Definition 2.10.3	Logical Update	38
Definition 3.2.1	Attributes Abstract Domain	42
Definition 3.4.1	Abstract Vector Element	45
Definition 3.4.2	Equivalence Relation	45
Definition 3.4.3	Set of Abstract Vectors	46
Definition 3.4.4	Abstract Domain of R Vectors	46
Definition 3.5.1	Partial Order	47
Definition 3.6.1	Join	49
Definition 3.6.2	Meet	49
Definition 3.7.1	Widening	50
Definition 3.8.1	Concretization	53
Definition 4.2.1	Abstract setAttr and getAttr	56
Definition 4.3.1	Squash	57
Definition 4.3.2	SquashExcept	57
Definition 4.3.3	Abstract Concatenation	57
Definition 4.4.1	Abstract Cycling $\rho_c^\sharp$	58
Definition 4.4.2	Full Recycling $\rho_f^\sharp$	59
Definition 4.4.3	Abstract Recycling	60
Definition 4.5.1	Abstract Cardinality	61
Definition 4.5.2	Enumerable Positions	62
Definition 4.5.3	Propagate	62
Definition 4.5.4	AdjustForZeros	63
Definition 4.5.5	Abstract Element Access	64
Definition 4.6.1	Abstract Positive Selection, Finite Selector	66
Definition 4.6.2	Abstract Positive Selection, Infinite Selector	66
Definition 4.6.3	Abstract Negative Selection, Non-Enumerable Position	68
Definition 4.6.4	Abstract Negative Selection, Enumerable, Infinite Selector	68

Definition 4.6.5	Abstract Logical Selection	70
Definition 4.7.1	Positive Update, Non-Enumerable Positions	72
Definition 4.7.2	InitKnown	72
Definition 4.7.3	UpdateKnown	73
Definition 4.7.4	Positive Update, Infinite Selector, Enumerable Positions	73
Definition 4.7.5	Positive Update, Finite Selector, Enumerable Positions	75
Definition 4.7.6	Negative Update, Non-Enumerable Position	77
Definition 4.7.7	Negative Update, Infinite Selector, All Enumerable	77
Definition 4.7.8	Negative Update, Finite Selector, Enumerable Positions	78
Definition 4.7.9	Abstract Logical Update	78

List of Theorems

Theorem 3.5.1	Partial Order	48
Theorem 3.7.1	Well-Foundedness of Widening	51
Theorem 3.8.1	Meet-Preservation	53
Theorem 3.8.2	Galois Connection	54
Theorem 4.4.1	Soundness of Abstract Recycling	61
Theorem 4.6.1	Soundness of Positive Selection	67

List of Lemmas

Lemma 3.4.1	Equivalence Relation	45
Lemma 1.1.1	Minimum Representative	94
Lemma 1.3.1	Bottom-Meet Length Bound	96
Lemma 1.3.2	Attribute Concretization is Meet-Preserving	97
Lemma 2.1.1	Padded Covers	102
Lemma 2.1.2	Cycling Soundness	102
Lemma 2.1.3	Join Soundness	103
Lemma 2.1.4	Summary Soundness	103
Lemma 2.1.5	AdjustForZeros Soundness	105
Lemma 2.1.6	Abstract Access Soundness	107

List of Proofs

Proof of Theorem 3.8.2	54
Proof of Lemma 3.4.1	94
Proof of Lemma 1.1.1	94
Proof of Theorem 3.5.1	95
Proof of Lemma 1.3.1	96
Proof of Lemma 1.3.2	97
Proof of Theorem 3.8.1	97
Proof of Theorem 3.7.1	99
Proof of Theorem 4.4.1	104
Proof of Theorem 4.6.1	108

List of Remarks

Remark	65
--------------	----

List of Figures

Figure 1	Hasse diagram of the Sign lattice.	18
Figure 2	Syntax of μR expressions.	26
Figure 3	Syntax of μR statements.	29

List of Tables

Table 1	Concrete semantics of μR expressions.	27
Table 2	Auxiliary functions for the concrete semantics.	28
Table 3	Concrete statement semantics of μR	29
Table 4	Abstract semantics of μR expressions.	82
Table 5	Mapping from R constructs to abstract operators.	84
Table 6	Representative test cases from the evaluation corpus.	87

Abstract

R is a dynamically typed, vector-oriented language widely used in data science and scientific computing. Its vector semantics, including automatic type coercion, recycling, and flexible selection and update operators, are pervasive, non-trivial, and deeply intertwined with the semantics of virtually every R operation. The complexity of these semantics makes static reasoning about R programs particularly challenging, and existing tools offer only shallow, best-effort analyses that fall short of reasoning about R vector-manipulating programs.

CONTEXT

In this thesis, we present a parametric abstract domain for R vectors, defined over μR , a core calculus we design to capture the most representative vector manipulation operations in R. An abstract vector simultaneously captures the possible lengths of the vector, the values of its elements, and its potential attributes. The domain is parametric in the element abstract domain.

OBJECTIVE

We define abstract operators for all μR operations and equip the domain with a widening operator to ensure termination of fixpoint computation. We prove the domain forms a complete lattice and establish a formal Galois connection with the concrete domain. Formal soundness proofs are provided for the recycling and positive selection operators. We implement the domain on top of flowR, a static analysis framework for R, and evaluate it on a suite of 61 handcrafted programs covering six categories of vector operations, demonstrating the feasibility and precision of the approach.

METHOD

Acknowledgments

I would like to express my sincere gratitude to my advisor, Prof. Vincenzo Arceri, for his guidance, encouragement, and trust throughout this journey. His mentorship has been invaluable in shaping both this thesis and my growth as a researcher. I look forward to continuing our collaboration in the future.

I am deeply grateful to Florian Sihler, whose coordination during my Erasmus+ period at Ulm University was instrumental in developing the work that forms the foundation of this thesis. The discussions, shared ideas, and his sharp insights made the research both productive and enjoyable. His thesis template served as an inspiration for the one developed for this work.

I also wish to thank Prof. Matthias Tichy for hosting me at Ulm University and supervising my work during that period, and Oliver Gerstl for his helpful feedback and suggestions.

Personal acknowledgments to family and friends have been removed in this digital version for privacy reasons.

Introduction

This chapter establishes the foundations on which the thesis rests, defining the problem of static analysis of R vectors whose runtime-determined lengths, implicit type coercion, and recycling semantics defeat conventional techniques. A discussion of static program analysis and R’s role in data science establishes the central claim that sound static reasoning about R requires a principled abstraction of its vector data model. A historical account traces R from its Bell Labs origins through its emergence as a pillar of statistical computing, explaining why static analysis tools never accompanied the language’s growth. The language features that challenge static analysis, including dynamic typing, lazy evaluation, first-class metaprogramming, and the vector data model with implicit coercion, recycling, and attribute-based structure, are surveyed in subsequent sections. Section 1.4 introduces abstract interpretation at a tutorial level: posets, lattices, Galois connections, fixpoint iteration, and widening. Section 1.5 positions the thesis within three strands of prior work: existing analysis tools for R, abstract interpretation for dynamic languages, and array-centric abstract domains. Section 1.6 identifies the gap no existing work addresses and enumerates four contributions together with the thesis’s scope and limitations. Section 1.7 provides a roadmap of the remaining chapters.

1.1 Context and Motivation

1.1.1 Static Program Analysis and Software Reliability

Static program analysis is the discipline of automatically reasoning about program behavior without executing the program. An analyzer examines source code or compiled representations and derives properties of interest: which variables might be uninitialized, where a buffer overflow could occur, whether a function consistently returns the expected type, or which resources a program allocates. These analyses serve several broad purposes. Bug detection tools identify latent defects before they surface in production. Optimizing compilers use static information to eliminate redundant computations and to allocate registers efficiently. Security audits depend on static reasoning to certify that untrusted code conforms to safety policies. Documentation generators extract structural summaries from source without running it. Static analyses also support program understanding, refactoring, and automatic test generation, among others.

1	Context and Motivation,	1
2	A Brief History of R,	4
3	The R Language: Core Features and Challenges,	7
4	Abstract Interpretation Foundations,	13
5	Related Work,	19
6	Research Gap and Contributions,	22
7	Thesis Structure,	23

⊗	Section 1.4,	13
⊗	Section 1.5,	19
⊗	Section 1.6,	22
⊗	Section 1.7,	23

Every useful static analysis faces a fundamental limit. Rice’s theorem proves that any non-trivial semantic property of programs is undecidable in the general case: no tool can correctly answer arbitrary questions about arbitrary programs (we formalize this result in Section 1.4). An analyzer inescapably approximates. The central choice is which direction the approximation takes. A sound analysis guarantees that it never misses a bug: if the analysis reports that a property holds at some program point, the property genuinely holds in every possible execution. The price of soundness is false alarms: the analysis may report a potential error even when no concrete execution could trigger it, because the abstraction loses precision. The alternative, a complete analysis, guarantees that every reported bug is real but cannot promise to find all of them. Abstract interpretation [1] chooses soundness. It supplies a mathematical framework to design analyses that over-approximate all possible program behaviors and to formally prove that the approximation is correct.

↻ Section 1.4,

13

[1] Cousot et al., 1977
*Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,*

Dynamic programming languages amplify these difficulties. In a statically typed language like Java, the compiler determines the type of every expression before the program runs, and a static analyzer can build its reasoning on that foundation. In a dynamic language like R, Python, or JavaScript, variable types, object structures, and even control flow targets are resolved only at runtime. A variable that holds an integer on one execution path may hold a character string on another. Functions can be created, reassigned, or deleted while the program runs. The traditional machinery of type-based static analysis does not apply to this setting. Sound static analyzers for dynamic languages must rely on abstractions of a different kind: abstractions that cope with type uncertainty, structural dynamism, and runtime code generation.

1.1.2 R’s Role in Data Science

R occupies an unusual position among programming languages. Developed as an interactive environment for statistical computing, it has become one of the most widely adopted languages for data analysis, bioinformatics, computational social science, and quantitative finance. The language centers on the workflow of the working data analyst: vectorized operations that manipulate entire columns of data in a single expression, a formula interface for specifying statistical models, and an extensive graphics system for exploratory visualization [2]. These affordances have made R the primary working language for several large scientific communities. Bioconductor, a curated repository of over two thousand packages for genomic data analysis, has established R as the de facto standard in bioinformatics [3]. In pharmaceutical research, R is used throughout the clinical trial pipeline, from study design through regulatory submission. Quantitative finance teams employ R for risk modeling, portfolio optimization, and time-series forecasting [4].

[2] R Core Team, 2026
R Language Definition,

[3] Gentleman et al., 2004
*Bioconductor: Open Software
Development for
Computational Biology and
Bioinformatics,*

[4] Nolan, 2012
*R as a Tool in Computational
Finance,*

The Comprehensive R Archive Network, CRAN, hosts 23,938 contributed packages as of June 2026 [5]. This ecosystem is a principal reason for the language’s sustained growth. A researcher who needs

[5] CRAN, 2026
Contributed Packages,

a specialized statistical method, a machine learning algorithm, or an interface to a big-data platform can usually find a well-tested, peer-reviewed R package that supplies it. The community has also invested in production infrastructure: packages for connecting R to distributed computing frameworks, such as Spark [6], tools for building web APIs around R models, and containerized deployment pipelines that package R analytics as microservices. R code is increasingly written not as one-off scripts for a single analyst's desktop but as components of multi-stage data processing pipelines that serve predictions to production systems.

[6] Venkataraman et al., 2016
SparkR: Scaling R Programs with Spark,

This trend raises the cost of bugs. A logical error in a statistical script run once by its author is an annoyance. The same error in a production pipeline that feeds automated decisions into a trading system, a medical device, or a public-facing service becomes a liability. Type confusion, silent recycling of mismatched vector lengths, and unintended coercion of data types are common sources of runtime failures in R programs. These errors are not caught until execution reaches the offending line, under data conditions that did not occur during development.

The tooling gap is striking. Languages with comparable or smaller adoption in data science, notably Python, benefit from a growing ecosystem of static analysis tools: type checkers such as mypy¹ and Pyright², linters including Pylint³ and Ruff⁴.

Security-focused analyzers like Bandit⁵ and abstract-interpretation-based tools such as Pyra⁶ [7] further strengthen this ecosystem.

JavaScript, despite its famously dynamic semantics, has TypeScript's gradual type system, the TAJs⁷ abstract interpreter, and a family of linting and verification tools. Java and C# integrate industrial-grade static analysis into their standard development environments. For R, by contrast, the available tooling for program correctness is sparse. Several projects have addressed pieces of the problem (Section 1.5), but no current tool provides a sound, value-level static analysis for the fundamental data structure of the language: the vector. This thesis addresses that gap.

1. <https://github.com/python/mypy>

2. <https://github.com/microsoft/pyright>

3. <https://github.com/pylint-dev/pylint>

4. <https://github.com/astal-sh/ruff>

5. <https://github.com/PyCQA/bandit>

6. <https://github.com/spangea/Pyra>

[7] Dolcetti et al., 2025
Introducing Pyra: A High-Level Linter for Data Science Software,

7. <https://github.com/cs-au-dk/TAJS>

© Section 1.5,

19

1.1.3 The Central Argument

The vector is the fundamental data structure in R. Every value in the language is a vector. The scalar constants `TRUE`, `3.14`, and `"hello"` are vectors of length one; the special value `NULL` is a vector of length zero. Every operation in R, from arithmetic to subsetting to statistical modeling, acts on vectors, and the semantics of that operation depends on three properties of its operands: their lengths, their element types, and the attributes they carry. None of these properties are known statically in general R code. Lengths are determined at runtime by the data being processed. Element types depend on the values the program constructs. Attributes are attached and modified dynamically.

A sound static analyzer for R must first provide a sound and precise abstraction of vectors. Without a model of vector length, the analyzer cannot detect recycling mismatches that silently corrupt results. Element types are equally essential; without modeling them, the analyzer cannot warn about coercions that destroy information, such as the conversion of numeric data to character strings through accidental concatenation. Without a model of attributes, it cannot resolve S3 method dispatch or track the dimensions of matrices and arrays. The vector abstraction is not an optional component of an R static analyzer: it is the foundation on which every other analysis must be built.

This thesis develops such an abstraction. We present a parametric abstract domain for R vectors that simultaneously tracks vector length, a sequence of individually tracked element values, a summary approximation for all remaining positions, and the attribute metadata attached to each vector. The domain is parametric in the element abstract domain: any lattice providing arithmetic, relational, and logical operations over scalar values can be plugged in, allowing the analysis to be tuned for different precision requirements. We define the domain's lattice structure and prove that it forms a complete lattice. We provide join, meet, and widening operators, and establish a formal Galois connection with the concrete vector domain. Chapter 3 develops this machinery in full. We then lift every operator of a minimal core calculus for R vectors to its abstract counterpart and prove each abstract operator sound with respect to its concrete semantics. Chapter 4 presents these operators and their proofs. The result is the first static analysis framework for R that provides formal soundness guarantees for reasoning about vectors under coercion, recycling, selection, and update.

1.2 A Brief History of R

The design choices that make R resistant to static analysis are not accidents. They are the accumulated consequences of a forty-year intellectual lineage that began in the statistical computing group at Bell Laboratories in the mid-1970s. This section traces that evolution from its origins through R's rise to its current position as a production language without production tooling. Understanding this history clarifies why the gap this thesis addresses is a gap at all, and not a problem that would have been solved organically by the R community.

1.2.1 Origins in S

The S language was created at Bell Laboratories in 1976 by John Chambers and colleagues [8]. Their objective was practical: give working statisticians an interactive computing environment designed around the data analysis workflow. At the time, statistical computing meant writing Fortran subroutines and submitting batch jobs; the turnaround between a hypothesis and its numerical test took hours. S replaced this cycle with a **REPL**: the analyst typed an

☞ Chapter 3, 41

☞ Chapter 4, 55

[8] Chambers, 1998
Programming with Data: A Guide to the S Language,

REPL, Read-Eval-Print Loop. An interactive programming environment that reads user input, evaluates it, prints the result, and loops for the next input. The primary interface for exploratory data analysis in R; also the dominant interaction model for Python, Lisp, and other languages descended from the interactive computing tradition.

expression, saw the result immediately, and adjusted the analysis on the fly.

This interactive orientation shaped every subsequent design decision. S adopted vectorized operations as its fundamental computation model because statistical work operates on variables across entire datasets, not on individual scalars. `mean(x)` computes the average of a vector `x`; `lm(y ~ x1 + x2)` fits a linear model and returns a structured result containing coefficients, residuals, and diagnostic statistics. The formula interface (`y ~ x1 + x2`) lets the user express statistical relationships in a notation that mirrors how they appear in textbooks. Chambers described the philosophy as "*programming with data*": the language should be a natural medium in which a statistician thinks about data, not a systems-programming tool [9].

[9] Becker et al., 1988
*The New S Language: A
Programming Environment for
Data Analysis and Graphics*,

Several design choices made in these early years carried through to R and continue to define the language's character. S was dynamically typed: variables had no declared types, and a function's argument could be a number on one call and a character string on the next. Functions were first-class values, and the language was extensible through user-defined functions that were indistinguishable from built-ins. The object system was pragmatic rather than formal: S version 3 introduced the S3 class mechanism in 1988, based on attaching a class attribute to an object and dispatching methods by name lookup. S version 4 replaced this with the more structured S4 system in 1998, adding formal class definitions and multiple dispatch. Both systems remain in R today and coexist with several newer object frameworks.

The S language was designed for statisticians, not for programming-language designers. Its syntax was chosen to feel natural for expressing statistical operations. The same ergonomic focus that made `plot(x, y)` an intuitive command for drawing a scatterplot also meant that S prioritized expressiveness over everything that static verification depends on: declared types, fixed memory layouts, and predictable evaluation order. This trade-off was entirely rational in 1976, and it remained so throughout S's lifespan: the community's principal concerns were numerical accuracy and statistical correctness, neither of which was addressed by compiler-style static checks.

[10] Ihaka et al., 1996
*R: A Language for Data
Analysis and Graphics*,

🔗 **Dynamic Scoping.** A variable scoping discipline where free variables resolve to bindings in the calling environment at runtime, following the dynamic call chain rather than the lexical structure of the program. Used by early Lisp dialects and the S language. R replaced this with lexical scoping, borrowing the evaluation model from Scheme.

🔗 **Lexical Scoping.** A variable scoping discipline where free variables in a function body resolve to bindings in the lexically enclosing definition environment, the environment where the function was defined, not where it is called. Also called **static scoping**. R inherited this model from Scheme, replacing the dynamic scoping used by S. Lexical scoping makes functions self-contained and enables closures and higher-order functions.

1.2.2 R's Birth and Philosophy

Ross Ihaka and Robert Gentleman, two statisticians at the University of Auckland, began developing R in 1993 and publicly announced the project in 1993 on the S-news mailing list, later introducing it to a wider audience through a 1996 paper [10]. R was conceived as a free implementation of the S language, but with one foundational change: Ihaka and Gentleman replaced S's **Dynamic Scoping** with **Lexical Scoping**, borrowing the evaluation model from Scheme. This combination of S's data analysis ergonomics with Scheme's evaluation discipline determined R's semantic character. Lexical scoping meant that free variables in a function body resolved to bindings in the enclosing definition environment, not the calling environment.

This made functions self-contained in a way that S functions were not, and it enabled a cleaner treatment of closures and higher-order functions.

R was released under the GNU General Public License. This licensing decision set R apart from the dominant commercial implementation of S, S-PLUS, which was distributed by MathSoft under a proprietary license. The GPL made R free in both senses: zero cost and modifiable source code. Academic users could inspect the interpreter, fix bugs, and write extensions knowing that their contributions would remain open. This openness attracted a global community of contributors who produced packages, documentation, and infrastructure at a rate that no single organization matched.

The philosophy that guided R's development was shaped by the working habits of its creators and their intended audience. Both Ihaka and Gentleman were statisticians who wrote code to analyze data; they built R to support that workflow and that community. The interactive **REPL** was the primary interface, designed for rapid iteration. An analyst would load a dataset, try several transformations, plot intermediate results, refine the model, and export the final figures, all within a single session. The idea of running a static analyzer on R code was not merely unimplemented; it was outside the philosophy that the language was built to serve. Correctness of an analysis was established through visualization of residuals, comparison against known statistical distributions, and expert judgment, not through a proof that the program was free of type errors.

1.2.3 Growth and Current Status

CRAN was established in 1997. As noted in Section 1.1.2, it has since grown to over twenty thousand packages, and R's adoption has spread across bioinformatics, quantitative finance, pharmaceutical research, and academic statistics [3, 4, 11]. This growth trajectory created a tension that static analysis could address, but the community and its tools evolved without it.

An empirical study of real-world R code by Sihler et al. provides a systematic picture of how R is used in practice [12]. The study analyzed thousands of R scripts and identified the language features that occur most frequently: vectorized operations, name-based indexing, reflective function calls, and attribute-based dispatch. These findings confirm that the vector-level semantics, the focus of this thesis, are not an edge case but the central computation pattern in practice.

The growth trajectory creates a tension. R was designed for interactive data analysis on a single analyst's machine, where the cost of a runtime error is a few minutes of debugging. The language now runs in production pipelines where the same error can propagate through automated trading systems, clinical decision support tools, or public-facing data services. A data type coercion that silently converts numbers to strings, a recycling rule that repeats a short vector to match a longer one without raising an error, a missing attribute that

Section 1.1.2,

2

[3] Gentleman et al., 2004
*Bioconductor: Open Software
Development for
Computational Biology and
Bioinformatics*,

[4] Nolan, 2012
*R as a Tool in Computational
Finance*,

[11] Fox, 2009
*Aspects of the Social
Organization and Trajectory of
the R Project*,

[12] Sihler et al., 2024
*On the Anatomy of Real-World
R Code for Static Analysis*,

changes how a model function dispatches: all of these are routine in R and all of them are invisible until execution reaches the offending line, sometimes under data conditions that did not occur during development. Static analysis tools detect these classes of bug before deployment, but the R ecosystem has developed few of them.

1.2.4 Why Static Analysis Never Developed for R

The absence of static analysis tooling for R is not a technical verdict on the language's analyzability. It is a historical contingency produced by the intersection of several factors.

R was born in a culture where program correctness was established through visualization, statistical testing, and peer review. Statisticians check their work by examining residual plots, computing diagnostic statistics, and comparing results against published benchmarks. Compiler checks were irrelevant to this workflow, and the language was never embedded in a development environment that required them. The user base consisted of statisticians and data analysts, not programming-language researchers or compiler engineers. There was no constituency within the R community that would naturally generate a demand for static analysis tools, and no community of tool builders who would supply them. The GNU GPL license ensured that R remained open, but it also meant that no corporate vendor had a commercial incentive to invest in industrial-strength compilers, type checkers, or static analyzers. Languages with comparable adoption, e.g., Java, C#, and JavaScript, all received substantial corporate investment in compiler and tooling infrastructure. R, despite its widespread use, had no equivalent. Finally, R's explosive growth through CRAN and Bioconductor happened organically, driven by package contributions from domain experts, not by a coordinated language-development effort. The tooling infrastructure that normally accompanies language adoption (debuggers, profilers, linters, and analyzers) developed slowly and unevenly, with static analysis remaining the least developed category.

The result is an asymmetry. R today has one of the richest package ecosystems of any programming language, covering statistical methods, machine learning, visualization, and domain-specific analysis for dozens of scientific fields. It has a mature interpreter, extensive documentation, and an active community of millions of users, even though its ecosystem of static analysis tools remains thin.

1.3 The R Language: Core Features and Challenges

The historical forces described in the previous section produced a language whose design resists static analysis at multiple levels. This section examines those levels concretely. We begin with the foundational issue: variables in R have no declared types, and all type checking is deferred to runtime. We then survey three additional layers of complexity that compound the challenge: lazy evaluation, metaprogramming, and copy-on-modify semantics. The third and longest subsection focuses on the vector data model, the dimension

of R that this thesis targets directly. We end by synthesizing these observations into a precise characterization of why static analysis of R is simultaneously necessary and hard.

1.3.1 *Dynamic Typing and Late Binding*

R is dynamically typed in the strongest sense. A variable does not have a type; a value does. In R, the same name can refer to values of entirely different kinds at different points in a program. The sequence `x ← 3; x ← "hello"` is not merely legal in R: the language makes no distinction between the integer 3 and the character string "hello" as variable contents. The assignment operator `←` rebinds a name to a new value with no static check that the types are compatible.

Type checking in R happens when an operation is evaluated, not when the program is parsed or loaded. The expression `x + 1` succeeds if `x` resolves to a numeric vector at the moment of evaluation and fails with a runtime error if `x` resolves to a character string. The parser does not know, and the interpreter does not check before execution begins. A function definition contains no type annotations for its parameters; a call is syntactically valid regardless of the argument types, and the failure, if any, occurs inside the function body when an operation receives an operand of unexpected type.

Name resolution follows environment chains that are constructed and modified at runtime [2]. At definition time, R captures the enclosing environment as the function's parent environment. When a name is referenced inside the function body, R searches the chain from the evaluation environment upward through successive parent environments until it finds a binding or exhausts the chain. The search happens at the moment of reference, not at the moment of definition, and the environment chain can be altered by package loading, namespace attachments, and explicit environment manipulation before the reference executes. A variable that resolves without error on one call may fail on the next because the environment chain has changed.

[2] R Core Team, 2026
R Language Definition,

The consequence for static analysis is severe. At any program point where a variable appears, neither its type nor its structure is statically known. Traditional type systems assign a type to every expression based on the types of its subexpressions; in R, the subexpression types are themselves unknown, so the type inference rule has nothing to work with. A static analyzer cannot rely on declared types because there are no declared types. It cannot rely on a fixed class hierarchy because classes are dynamically attached as attributes and dispatch is resolved by name lookup at runtime. Every approach to static analysis that depends on type information must first reconstruct that information from the program text without the assistance of type declarations, a problem that is undecidable in the general case.

The abstract domain we develop in this thesis does not directly address dynamic typing in its full generality. It assumes that a program

point has been reached and that the vector values present at that point are abstracted by the domain. The formal grounding for these values, and the environment model that governs name resolution, are defined in Chapter 2. What the domain does provide is the ability to soundly approximate the content and structure of a vector once it is known, or once an abstraction of it has been constructed: to track how its length, type, and attributes change under each vector operation in the language.

1.3.2 Further Complexity: Lazy Evaluation, Metaprogramming, and Copy-on-Modify

Dynamic typing is the first layer of difficulty, but R adds three further layers that deepen the challenge for any static analysis. None of these are directly modeled by the abstract domain we present, which focuses on the vector data model. They are, however, essential context for understanding why R is a uniquely demanding target and why scoping the problem to vectors, rather than attempting a full-language analysis, is the right first step.

R function arguments are not values; they are **promises**: unevaluated expressions paired with the environment in which they were captured [13]. A promise's body is evaluated only when the formal parameter is first accessed inside the function, an event R calls *forcing*. If the parameter is never accessed, the expression is never evaluated at all. A promise forced twice in succession may produce different values: the environment captured with the promise can be mutated between the first and second forcing, and the expression can embed side effects.

```
f ← function(x) { x; x }
y ← 1
f({ y ← y + 1; y }) # evaluates to 2, then 3
```

On the first access, *y* increments to 2 and returns; on the second, it increments to 3. A static analyzer cannot determine, in general, which promises will be forced, in what order, or what values they will produce [14]. Several formalizations have attempted to tame R's lazy evaluation by specifying its semantics in operational or denotational frameworks [15], but the gap between a formal model and a computable abstraction suitable for a static analyzer has not been closed by existing work.

R exposes environments as first-class values. The function `parent.env()` retrieves the enclosing environment of any environment object; `new.env()` creates a fresh environment; closures capture their definition environment and carry it through program execution. More problematically for static analysis, R provides a family of **NSE** primitives: `substitute()` retrieves the unevaluated syntactic form of its argument, `quote()` creates an expression object, and `eval()` evaluates an arbitrary expression in an arbitrary environment.

LAZY EVALUATION

[13] Wickham, 2019
Advanced R,

[14] Goel et al., 2019
On the Design, Implementation, and Use of Laziness in R,

[15] Morandat et al., 2012
Evaluating the Design of the R Language: Objects and Functions for Data Analysis,

FIRST-CLASS ENVIRONMENTS AND METAPROGRAMMING

☞ **NSE, Non-Standard Evaluation.** R's ability to capture, inspect, and manipulate unevaluated code before execution. Key primitives include `substitute()`, which retrieves the unevaluated syntactic form of its argument, `quote()`, which creates an expression object, and `eval()`, which evaluates an arbitrary expression in an arbitrary environment. NSE enables domain-specific interfaces (e.g., `subset()`) but prevents any purely static analysis from determining which code will execute.

```
eval(parse(text = "x ← 42")) # defines a variable from a
string at runtime
```

An expression passed to `eval()` can be constructed at runtime from string manipulation, file input, or network data. No static analyzer can predict its content, yet that expression can define variables, attach attributes, and alter the program's control flow. The combination of first-class environments and **NSE** means that a significant proportion of real-world R code reflects on and modifies its own structure dynamically [12]. R's intermediate representation is correspondingly complex: the $\tilde{R}$ project documents numerous syntactic forms that the evaluator must handle, many of them related to metaprogramming constructs [16].

R's memory model avoids aliasing through a copy-on-modify discipline: when a vector is assigned to a new variable or passed as a function argument, no copy is made initially, but the first write to the shared data triggers a copy so that modifications do not affect other references [17]. This semantics is benign from a correctness standpoint; it eliminates the aliasing bugs that plague languages with mutable shared state. It does, however, complicate pointer and alias analysis, since references that appear to share a vector may or may not share the underlying storage depending on which paths have executed writes.

The three layers interact. A metaprogramming construct can construct an expression containing a variable reference, which is then evaluated in a lazy context, where it is forced at a point that triggers a copy-on-modify action. A static analysis that attempts to soundly account for all three simultaneously faces an intractable explosion of cases. This thesis sidesteps that explosion by isolating the vector layer. We define μR , a core calculus whose semantics includes no lazy evaluation, no metaprogramming, and no mutation. It models vectors, vector operations, and nothing else. Sound abstractions of vectors built on μR can later be integrated into analyses that handle the remaining language features; those analyses are beyond the scope of this work, but the abstraction they would need is exactly what we provide.

1.3.3 The Vector Data Model

As established in Section 1.1.3, the vector is the fundamental data structure of R and the focus of this thesis. This subsection examines the four properties that make R vectors resistant to static reasoning: the absence of scalars, implicit type coercion, length recycling, and the attribute system. Every R programmer works with these mechanisms daily; they are the source of the language's conciseness and also of its most pervasive bugs.

The most important fact about R's data model is that there are no scalars. Every value is an atomic vector. The logical constant `TRUE`, the integer `3L`, the floating-point number `3.14`, and the string `"hello"` are vectors of length one. The special value `NULL` is a vector of length zero. This has no parallel in languages like Python or C, where 3 is

[12] Sihler et al., 2024
*On the Anatomy of Real-World
R Code for Static Analysis*,
8. [https://github.com/
reactorlabs/rir](https://github.com/reactorlabs/rir)

[16] Flückiger et al., 2019
*R Melts Brains: An IR for
First-Class Environments and
Lazy Effectful Arguments*,

**COPY-ON-MODIFY
SEMANTICS**

[17] Bodin et al., 2018
*A Trustworthy Mechanized
Formalization of R*,

☞ Section 1.1.3,

3

No SCALARS

a scalar and `[3]` is a list containing a scalar. In R, `3` and `c(3)` are the same value: a numeric vector of length one. Every operator is defined over vectors. Even a function expecting a single number receives a length-one vector and must extract its sole element.

This has direct consequences for static analysis. Length is a property of every value at every program point, and it must be tracked. Operations that implicitly assume length-one inputs silently recycle their operands when lengths differ, a behavior we examine below. A static analyzer that treats `x` as a scalar ignores the dimension that determines the semantics of every operation on `x`.

R enforces type homogeneity within atomic vectors. A vector cannot contain a logical value in position one, an integer in position two, and a character string in position three. When values of different types are combined, for instance through the concatenation operator `c()`, R promotes all elements to a common type according to the hierarchy:

TYPE COERCION

logical < integer < double < character

Logical values become integers (or doubles, or strings); integers become doubles (or strings); doubles become strings if any element is a string. The coercion is implicit: the user writes no cast and receives no warning beyond the silent loss of the original type distinctions.

```
c(TRUE, 3L, 2.5)      # → c(1.0, 3.0, 2.5)
c(TRUE, "hello")      # → c("TRUE", "hello")
```

In the first example, `TRUE` coerces to `1.0` and `3L` coerces to `3.0` because the presence of the double `2.5` pulls the vector type upward. In the second, every element becomes a string. This behavior is defined by R's coercion rules [2] and is independent of context: `c()` always applies the same promotion logic. A static analyzer that cannot predict which types will appear as arguments to `c()` cannot predict the type of the result, and therefore cannot predict the behavior of every operation that consumes that result.

[2] R Core Team, 2026
R Language Definition,

Coercion also occurs during arithmetic. When a double and an integer are added, the integer is promoted to double before the addition; the result is always a double. Logical vectors in arithmetic contexts are coerced to integer: `TRUE` becomes `1`, `FALSE` becomes `0`. The ubiquity of implicit coercion means that type information propagates backward through a program: the type of an expression on line fifty determines the coercion applied to a variable bound on line ten, through a chain of intermediate operations.

R applies vectorized operations to operands position by position: `c(1, 2, 3) + c(10, 20)` adds the first element of each vector, then the second, then attempts to add a third. When the operands have different lengths, R recycles the shorter one: it repeats the shorter vector cyclically until it reaches the length of the longer.

VECTOR RECYCLING

```
c(1, 2, 3) + c(10, 20)  # → c(11, 22, 13)
```

The second vector `c(10, 20)` becomes `c(10, 20, 10)` before the addition. R issues a warning when the longer length is not a multiple of the shorter, but it does not raise an error [13]. The computation proceeds silently with the recycled values. If the length of the shorter operand is only known at runtime, the analyst cannot determine, by reading the source, how many times recycling will occur or whether the last elements of the longer vector will be paired with recycled elements from a previous cycle. This is a rich source of subtle numerical errors: a vector intended to be length five assembled from a database query of length four will recycle positions one to three of another vector to form the result, producing numerically valid but semantically wrong output.

[13] Wickham, 2019
Advanced R,

Every R vector can carry an arbitrary set of attributes: a collection of named key-value pairs attached to the object. Most attributes are user-defined and have no effect on the language's built-in behavior, but several have special semantics.

ATTRIBUTES

The `names` attribute labels individual elements. A vector `c(x = 1, y = 2)` is a numeric vector of length two whose first element is accessed by position as `1` or by name as `"x"`. The names are metadata; they affect display and subsetting but not arithmetic.

The `dim` attribute reshapes a vector into a matrix or array. A numeric vector of length six with `dim` set to `c(2, 3)` becomes a 2×3 matrix. The underlying storage is unchanged: a matrix is a vector with a dimension attribute, and a data frame is a list with a dimension attribute and a class.

The `class` attribute triggers R's S3 object system. When a generic function such as `print()` or `summary()` is called on a value, R inspects the value's `class` attribute, searches for a method with the corresponding name, and dispatches to that method. The same syntax `print(x)` can invoke different code depending on whether `x` carries the class `"lm"` (a linear model) or `"data.frame"` (a tabular dataset). Since the class attribute is set dynamically, a static analyzer cannot resolve method dispatch without tracking attributes.

The `levels` attribute defines the set of valid categories for a factor, R's representation of categorical data. A factor is an integer vector whose integer codes index into a character vector of level labels stored as an attribute. Operations that combine factors must reconcile their level sets, and the rules governing this reconciliation are complex.

Attributes are pervasive in R. Every data structure above the atomic vector, including matrices, arrays, data frames, factors, and S3 objects, is a vector decorated with one or more attributes. A static analysis that ignores attributes cannot distinguish a matrix from a plain vector, a linear model from a list of numbers, or a factor from an integer. The abstract domain we develop in Chapter 3 includes an attribute component precisely for this reason: attributes are not an optional decoration but an integral part of R's type system.

The formal definitions of the concepts introduced here appear in Chapter 2. Vector representation, including the distinction between atomic vectors and their attribute sets, is specified in Section 2.3. The type coercion hierarchy and the coercion function are defined in Section 2.4. The recycling operator, which cyclically repeats a shorter vector to match a longer one, is formalized in Section 2.8. Attribute construction and inspection are treated in Section 2.6.

↻ Chapter 2,	25
↻ Section 2.3,	30
↻ Section 2.4,	31
↻ Section 2.8,	33
↻ Section 2.6,	32

1.3.4 Four Challenges

The preceding sections have isolated four families of features that make R resistant to static analysis. Dynamic typing means that no type information is available at analysis time: the type of every variable must be reconstructed from the program text. Lazy evaluation means that the order and even the occurrence of evaluations depend on runtime forcing patterns that are undecidable to predict. First-class environments and metaprogramming mean that a program can construct and evaluate arbitrary code at runtime, defeating any analysis that assumes a fixed syntactic structure. The vector data model adds implicit coercion, length recycling, and dynamic attributes: the length, type, and metadata of every value depend on the runtime lengths and types of the values combined to produce it.

Any one of these alone would challenge a static analysis. R combines all four in a language that was designed without any provision for static verification. The combination is not an incremental increase in difficulty; it is a qualitative change. Standard approaches that work for languages with one or two of these features founder when faced with their interaction.

This thesis does not attempt to solve the full problem. As argued in Section 1.1.3, it isolates the vector dimension: the data model that determines how values are constructed, combined, and accessed. A sound abstraction of vectors is the essential building block without which no complete static analyzer for R can exist. Lazy evaluation, metaprogramming, and dynamic dispatch are layers that sit on top of the vector model; a static analysis targeting those layers will need a vector abstraction underneath them. By developing that abstraction with formal soundness guarantees, we provide the foundation that subsequent work can build upon. The next two chapters of this thesis construct that foundation, and the remainder of this introduction provides the theoretical and contextual grounding needed to follow the construction.

↻ Section 1.1.3,	3
------------------	---

1.4 Abstract Interpretation Foundations

This section introduces the theoretical framework that supports the remainder of this thesis.

We define the mathematical structures that underpin abstract interpretation: partially ordered sets, complete lattices, Galois connections, fixpoint computation, and widening. Readers already familiar with these concepts may proceed directly to the related work discussion in Section 1.5. Throughout this section we adopt notation from

↻ Section 1.5,	19
----------------	----

the standard abstract interpretation literature and provide concrete examples to ground each concept before it is deployed in Chapter 3 and Chapter 4.

☞ Chapter 3,

41

☞ Chapter 4,

55

1.4.1 The Need for Sound Approximations

A program analyzer confronts a hard computational boundary. It aims to automatically determine semantic properties of programs, answering questions about what values a variable can assume, whether a specific line of code is reachable, whether an array index is always within bounds. All such questions instantiate a single pattern: given an arbitrary program P , does P possess property φ ?

Rice’s theorem [18] bounds what is possible. A property φ is called *semantic* if it depends only on the partial function computed by the program, not on its syntactic form. It is *non-trivial* if there exists at least one program that has it and at least one program that does not. Rice’s theorem states that every non-trivial semantic property of partial functions is undecidable: no algorithm can correctly answer the question for all programs. The reachability of a given program point, the boundedness of an integer variable, and the absence of type errors are all non-trivial semantic properties. None can be decided automatically in the general case.

[18] Rice, 1953
*Classes of Recursively
Enumerable Sets and Their
Decision Problems,*

Faced with undecidability, a static analysis must compromise. There are two directions. A **Soundness** analysis guarantees that every reported property genuinely holds: if the analysis claims that a variable cannot be negative at line 42, then no possible execution can produce a negative value at that point. The cost is incompleteness: the analysis may report a potential error even when no concrete execution would trigger it. The analysis returns “maybe” when the truth is “no”. Conversely, a complete analysis guarantees that every reported error is real, but cannot promise to find all of them: it returns “no” only when provably impossible, and risks returning a false “yes”.

🔔 **Soundness.** The property that abstract operations over-approximate their concrete counterparts. Formally: for all $x \in \gamma(a_1)$ and $y \in \gamma(a_2)$, the result of a concrete operation $\text{op}(x, y)$ must be contained in $\gamma(\text{op}^i(a_1, a_2))$. Soundness guarantees that the analysis never misses a concrete behavior.

Abstract interpretation [1] provides a mathematical framework for designing sound analyses. It treats the set of all possible program states as a concrete semantic domain and systematically constructs an abstract domain, a simplified and computable space of program properties, together with a formal relationship that guarantees soundness. An analysis derived within this framework is provably conservative: it may compute a superset of the actual behaviors, but it never misses one. The remainder of this section develops the mathematical machinery that makes this guarantee precise.

[1] Cousot et al., 1977
*Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,*

1.4.2 Posets, Lattices, and the Concrete Domain

The formal development of abstract interpretation begins with order theory.

A *partially ordered set*, or poset, is a set P equipped with a binary relation $\sqsubseteq$ that is reflexive ($x \sqsubseteq x$), antisymmetric ($x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$), and transitive ($x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$) for all $x, y, z \in P$.

PARTIALLY ORDERED
SET

The relation $\sqsubseteq$ is called a partial order; it captures a notion of relative information content or precision.

Given a subset $S \subseteq P$, an element $u \in P$ is an *upper bound* of S if $s \sqsubseteq u$ for all $s \in S$. An upper bound u is the *least upper bound*, or *join*, of S , denoted $\sqcup S$, if $u \sqsubseteq v$ for every upper bound v of S . Lower bounds and the *greatest lower bound* (or *meet*, $\sqcap S$) are defined dually. The join and meet of two elements a, b are written $a \sqcup b$ and $a \sqcap b$, respectively.

A poset L is a *complete lattice* if every subset $S \subseteq L$ has both a join $\sqcup S$ and a meet $\sqcap S$ in L . Two distinguished elements follow immediately from this definition: the join of the empty set is the least element of the lattice, denoted $\perp$ (bottom), and the meet of the empty set is the greatest element, denoted $\top$ (top). The bottom element represents total ignorance or an impossible state, depending on the interpretation; the top element represents maximal imprecision.

COMPLETE LATTICE

A subset of elements need not be finite for a join to exist, but finite joins have special significance for computation. A sequence $x_0, x_1, x_2, \dots$ in L is an **Ascending Chain (AC)** if $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots$. Equivalently, the poset satisfies the **Ascending Chain Condition (ACC)** if every ascending chain eventually stabilizes: there exists an index n such that $x_n = x_{n+1} = x_{n+2} = \dots$. If the condition is satisfied, a naive fixpoint iteration over the sequence is able to converge in finitely many steps. Lattices without **ACC**, which are usually the ones of practical interest for program analysis, require additional machinery to guarantee termination.

A property, in abstract interpretation, is identified with the set of program states that satisfy it. A program state $\sigma \in \Sigma$ is a mapping from variable names to their current values, so $\sigma(x)$ denotes the value of variable x in state σ . For a program with state space Σ , the property “the variable x holds a negative value” is the set $\{\sigma \in \Sigma \mid \sigma(x) < 0\}$, a subset of Σ . Every property, then, is simply a set of values drawn from the state space. The collection of all properties expressible over Σ is the powerset $\mathcal{P}(\Sigma)$, and this collection, ordered by subset inclusion, forms the concrete lattice $\langle \mathcal{P}(\Sigma), \subseteq \rangle$. The partial order captures precision directly: $S \subseteq T$ means that S describes fewer possible states than T and is therefore more informative. Their join is set union, their meet intersection. The empty set is the bottom element (no state satisfies the property) and Σ itself is the top element (the property provides no information).

CONCRETE DOMAIN

This powerset lattice is the reference point for every analysis we construct. It is exact but uncountable in general: it cannot be manipulated algorithmically. The task of abstract interpretation is to replace it with a lattice whose elements are finite representations while retaining a formal guarantee that the replacement is conservative.

1.4.3 Galois Connections

The bridge between the concrete and abstract worlds is a pair of adjoint functions. Let $\langle C, \subseteq \rangle$ be the concrete lattice (for our purposes, $C = \mathcal{P}(\Sigma)$) and $\langle A, \sqsubseteq \rangle$ an abstract lattice. The *abstraction function* α :

$C \rightarrow A$ maps a concrete property to the abstract element that best describes it. The *concretization function* $\gamma : A \rightarrow C$ maps an abstract element back to the set of concrete states it represents.

The pair (α, γ) forms a *Galois connection* when the following adjoint property holds for all $c \in C$ and $a \in A$:

GALOIS CONNECTION

$$\alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a).$$

This condition is the formal guarantee of soundness. It states that the abstraction of a concrete element c is below an abstract element a precisely when c is contained in the concretization of a . From this adjoint property two soundness conditions follow directly:

$$c \subseteq \gamma(\alpha(c)) \quad \text{and} \quad \alpha(\gamma(a)) \sqsubseteq a.$$

The first condition says that abstraction followed by concretization can only enlarge the set: α may lose precision, but γ recovers a superset. The second condition says that concretization followed by abstraction can only move downward in the abstract lattice: the abstract element a already over-approximates every concrete state in $\gamma(a)$. In other words, $\gamma \circ \alpha$ is extensive while $\alpha \circ \gamma$ is reductive.

1.4.4 Fixpoint Computation and Widening

Program semantics are defined by recursive equations. Consider a loop: the set X of states reachable at the loop head must satisfy $X = I \cup F_{\text{body}}(X)$, where I is the set of entry states and F_{body} models the effect of executing the loop body on a set of states. This is a fixpoint equation $X = F(X)$ with $F(Y) = I \cup F_{\text{body}}(Y)$, and we seek the *least* solution, i.e., the smallest X satisfying the equation, which excludes spurious unreachable states.

Given a program, its *collecting semantics* associates each program point with the set of concrete states that can reach that point. This set is the fixpoint, specifically the least fixpoint, of a monotone semantic function $F : C \rightarrow C$ that models the effect of executing one program step. The Knaster-Tarski fixpoint theorem [19] guarantees that such a least fixpoint exists when F is monotone over a complete lattice. Under the standard assumption that F is continuous, the least fixpoint can be expressed as the limit of iterated application from the bottom element [1]:

COLLECTING SEMANTICS

$$\text{lfp}(F) = \bigsqcup_{i \in \mathbb{N}} F^i(\perp).$$

Intuitively, we start with the empty set of states ($\perp$), apply F repeatedly to accumulate reachable states, and take the limit.

The analysis cannot compute $\text{lfp}(F)$ directly, since F operates on the uncountable concrete lattice. Instead, we define an abstract semantic function $F^\sharp : A \rightarrow A$ that operates on the abstract lattice and soundly approximates F . The soundness condition for abstract interpretation, as established by Cousot and Cousot [1], requires:

[19] Tarski, 1955
A Lattice-Theoretical Fixpoint
Theorem and Its Applications,

[1] Cousot et al., 1977
Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,

[1] Cousot et al., 1977
Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,

$$\alpha \circ F \subseteq F^\sharp \circ \alpha,$$

where $\subseteq$ denotes pointwise ordering of functions. This condition, together with the Galois connection, guarantees that the abstract fixpoint over-approximates the concrete fixpoint:

$$\text{lfp}(F) \subseteq \gamma(\text{lfp}(F^\sharp)).$$

Computing the abstract fixpoint via naive iteration, $a_0 = \perp_A$, $a_{i+1} = a_i \sqcup_A F^\sharp(a_i)$, would converge if the abstract lattice satisfied **ACC**. Most abstract lattices used in practice do not.

Consider the set of abstract intervals **Itv**, defined as

$$\text{Itv} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\} \cup \{\perp\},$$

where $\perp$ is a distinguished bottom element representing the empty interval. The lattice has infinite height: the chain $[0, 0] \sqsubseteq [0, 1] \sqsubseteq [0, 2] \sqsubseteq \dots$ never stabilizes. Analyzing a simple loop that increments a counter will produce an unbounded sequence of interval abstractions, necessitating widening.

The **Widening** operator $\nabla : A \times A \rightarrow A$ enforces termination. It must satisfy two conditions [20]:

1. For all $a, b \in A$: $a \sqsubseteq a \nabla b$ and $b \sqsubseteq a \nabla b$ (the result is an upper bound of both operands).
2. For every ascending chain $(x_i)_{i \geq 0}$, the chain $(y_i)_{i \geq 0}$ defined by $y_0 = x_0$, $y_{i+1} = y_i \nabla x_{i+1}$ eventually stabilizes.

The intuition is pragmatic. When naive iteration generates an ever-growing sequence of abstract states, widening jumps to a safe over-approximation that stabilizes in finitely many steps. The jump trades precision for termination, but the result remains a sound upper bound of all intermediate states.

A standard example is interval widening. Given two intervals $[l_1, u_1]$ and $[l_2, u_2]$, the widened interval is:

$$[l_1, u_1] \nabla [l_2, u_2] = \left[\begin{cases} -\infty & \text{if } l_2 < l_1 \\ l_1 & \text{otherwise} \end{cases}, \begin{cases} +\infty & \text{if } u_2 > u_1 \\ u_1 & \text{otherwise} \end{cases} \right].$$

If the upper bound repeatedly increases, widening immediately pushes it to $+\infty$, stabilizing the chain at the next iteration. The lower bound is treated symmetrically.

A related operator, *narrowing* $\Delta : A \times A \rightarrow A$, can recover some precision after widening by iterating downward from the widened result. We do not use narrowing in this thesis; the widening operator $\nabla_{\mathcal{V}_B^\sharp}$ defined in Section 3.7 is sufficient for our domain, and its well-foundedness is proved in Section A.4.

1.4.5 Abstract Domains: Definition and Examples

An abstract domain is a set A equipped with four components:

WIDENING
[20] Cousot, 2021
Principles of Abstract Interpretation,

↻ Section 3.7,

50

↻ Section A.4,

99

ABSTRACT DOMAIN

1. A partial order $\sqsubseteq$ that forms at least a poset, and ideally a complete lattice.
2. A concretization function $\gamma : A \rightarrow C$ that gives each abstract element a meaning in the concrete domain.
3. Abstract counterparts for every program operation that appears in the language being analyzed.
4. A widening operator ∇ if the lattice has infinite height.

The **Sign** domain is the simplest example. It consists of five elements: $\text{Sign} = \{\perp, -, 0, +, \top\}$, with the partial order $\perp \sqsubseteq - \sqsubseteq \top$, $\perp \sqsubseteq 0 \sqsubseteq \top$, $\perp \sqsubseteq + \sqsubseteq \top$, where $-$ and 0 and $+$ are pairwise incomparable. Figure 1 shows the Hasse diagram.

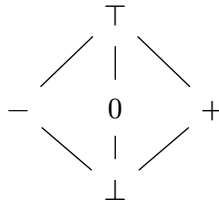


Figure 1: Hasse diagram of the **Sign** lattice.

The concretization $\gamma : \text{Sign} \rightarrow \mathcal{P}(\mathbb{Z})$ is:

$$\begin{aligned}
 \gamma(\perp) &= \emptyset, \gamma(\top) = \mathbb{Z}, \gamma(0) = \{0\}, \\
 \gamma(-) &= \{n \in \mathbb{Z} \mid n < 0\}, \\
 \gamma(+) &= \{n \in \mathbb{Z} \mid n > 0\}
 \end{aligned}$$

The abstraction of a set $S \subseteq \mathbb{Z}$ is the most precise abstract element whose concretization covers S : $\alpha(\{1, 3\}) = +$, and $\gamma(\alpha(\{1, 3\})) = \{n \mid n > 0\}$, which is a strict superset of $\{1, 3\}$. The sign domain is sound but imprecise: it tracks only the sign, losing the exact values.

Abstract operations are defined by case tables that pairwise combine signs and return the tightest abstract sign containing the result. For example, $(+) +_{\text{Sign}} (+) = +$, while $(+) +_{\text{Sign}} (-) = \top$ because the sum of a positive and a negative integer can have any sign.

The **l_{tv}** domain provides strictly more precision than **Sign**: it tracks bounds, not merely signs. Abstract addition is component-wise: $[l_1, u_1] +_{\text{l_{tv}}} [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$. The price is infinite height and the consequent need for widening.

These two examples illustrate a fundamental design axis: the trade-off between precision and cost. Both are *non-relational* domains: they track properties of each variable independently. Relational domains, by contrast, capture constraints between variables. The octagon domain [21] tracks inequalities of the form $\pm x \pm y \leq c$, useful for array bounds analysis. Polyhedra [22] track arbitrary linear constraints, offering maximal precision at substantial computational cost. The choice of domain is always application-specific.

Some domains are *parametric*: they accept another domain as a parameter and enrich it with structural information. The FunArray domain of Cousot, Cousot, and Logozzo [23] is a parametric segmen-

[21] Miné, 2006
The Octagon Abstract Domain,
[22] Cousot et al., 1978
Automatic Discovery of Linear
Restrains Among Variables of
a Program,

[23] Cousot et al., 2011
A Parametric Segmentation
Functor for Fully Automatic
and Scalable Array Content
Analysis,

tation functor that automatically partitions arrays into consecutive non-overlapping segments, each abstracted uniformly by a user-supplied element domain. Our vector domain follows the same parametric strategy. It is parametric in the element abstract domain $\mathcal{B}^\sharp$ (defined in Section 3.2), which must provide arithmetic, relational, and logical operations over scalar values. Any domain satisfying this interface can be plugged in.

The framework introduced in this section, i.e., concrete domain, abstract domain, Galois connection, and soundness condition, is the mathematical backbone of the thesis.

1.5 Related Work

We review related work from three complementary perspectives. First, we provide an overview of existing static analysis techniques for the R language. We then discuss abstract interpretation approaches for dynamic languages. Finally, we examine prior work on array-centric analysis. Each tier concludes with the gap that this thesis fills.

1.5.1 Static Analysis for R

Early work on static analysis for R includes ROSA⁹ [24], which provides type inference and standard compiler optimizations such as reaching definitions, live variables, and aliasing. ROSA lacks a foundation in abstract interpretation theory and focuses on the imperative subset of the language without modeling R’s type coercions, recycling semantics, or attribute system. Its contributions lie in optimization, not in sound value-level reasoning.

The `codetools`¹⁰ package [25] provides basic syntactic analysis utilities: global variable detection, unused variable checking, and function argument inspection. It performs no value-level reasoning and makes no attempt to model R’s complex runtime semantics. Its utility is limited to surface-level code hygiene checks.

Several interpreters and ahead-of-time compilers for R defer complexity to runtime instrumentation. `FastR`¹¹ [26] is a high-performance R interpreter built on the Truffle framework that uses runtime type feedback for speculative optimization. `pqR`¹² [27] extends the reference R interpreter with deferred evaluation and memory optimizations. The `Ŕ` project [28] employs sampling-based profiling to generate AOT-compiled native code from R. These systems target execution speed, not static correctness verification, and their design centers on runtime mechanisms rather than compile-time reasoning.

More recently, `flowR`¹³ [29] statically derives dataflow graphs and control-flow graphs from R source code to support program slicing, dependency tracking, value resolution, and data shape inference. It is the most advanced static analysis framework for R to date and provides the program representation on which the proof-of-concept implementation in Chapter 5 is built. The same authors conducted large-scale empirical studies of real-world R code [12], identifying

↻ Section 3.2,

42

9. <https://github.com/UWQuickstep/rosa>
[24] Sen et al., 2017
ROSA: R Optimizations with Static Analysis,

10. <https://cran.r-project.org/package=codetools>
[25] Tierney, 2024
`codetools`: Code Analysis Tools for R,

11. <https://github.com/oracle/fastr>
[26] Kalibera et al., 2014
A Fast Abstract Syntax Tree Interpreter for R,
12. <https://github.com/pqR-project/pqR>
[27] Neal, 2014
Speed Improvements in pqR: Current Status and Future Plans,
[28] Flückiger et al., 2020
Sampling Optimized Code for Type Feedback,

13. <https://github.com/Code-Inspect/flowr>
[29] Sihler et al., 2025
Statically Analyzing the Dataflow of R Programs,

↻ Chapter 5,

81

[12] Sihler et al., 2024
On the Anatomy of Real-World R Code for Static Analysis,

the language features that occur most frequently in practice: vectorized operations, name-based indexing, reflective function calls, and attribute-based dispatch. These findings confirm that the vector-level semantics targeted by this thesis are not an edge case but the central computation pattern in R, and they inform which operations the abstract domain must cover.

What remains unaddressed is that none of these approaches constructs an abstract domain that simultaneously tracks vector length, type, element values, and attributes while soundly modeling coercion and recycling. This is the gap the present work fills.

1.5.2 Abstract Interpretation for Dynamic Languages

Beyond R, abstract interpretation has been successfully applied to dynamic languages with complex runtime behaviors. For JavaScript, TAJIS [30] employs abstract interpretation to model prototype chains and dynamically created properties, constructing a flow-sensitive type analysis that infers both primitive and object types. JSAI [31] provides a configurable static analysis platform for JavaScript that models prototype-based inheritance, dynamic property addition, and first-class functions. Arceri et al. [32] extend this line of work with an abstract domain for object maps that uses finite-state automata to approximate sets of possible property names, targeting the heterogeneous, string-keyed nature of JavaScript objects.

Similar techniques appear for PHP. Phantm [33] offers flow-sensitive type checking for PHP, including support for associative arrays whose keys and values are statically unknown. Hauzar and Kofroň [34] integrate heap and value abstractions for mutable arrays and objects within a unified static analysis framework, combining multiple abstract domains to track both scalar values and complex data structures.

For Python, Monat et al. [35] develop abstract domains for lists and dictionaries within the Mopsa¹⁴ static analysis framework, achieving both flow and context sensitivity through a modular domain architecture that can be instantiated with different numeric and collection abstractions.

A common characteristic of all these works is their focus on heterogeneous, string-keyed collections: objects in JavaScript, associative arrays in PHP, and dictionaries in Python. The key modeling challenge in these settings is approximating the set of string keys that a collection may possess at runtime. R vectors present a fundamentally different challenge. They are homogeneous: every element in an atomic vector has the same type. They are position-indexed: elements are accessed by integer position, not by string key. And they are subject to implicit type coercion and length recycling, two semantic features with no direct analogue in string-keyed collections. The techniques developed for heterogeneous collections, which center on key-set approximation, do not transfer to the vector setting. The central concerns in the vector setting are length tracking, element-type

[30] Jensen et al., 2009
Type Analysis for JavaScript,

[31] Kashyap et al., 2014
JSAI: A Static Analysis Platform for JavaScript,

[32] Arceri et al., 2020
An Abstract Domain for Objects in Dynamic Programming Languages,

[33] Kneuss et al., 2010
Phantm: PHP Analyzer for Type Mismatch,

[34] Hauzar et al., 2015
Framework for Static Analysis of PHP Applications,

[35] Monat et al., 2020
Static Type Analysis by Abstract Interpretation of Python Programs,

14. <https://gitlab.com/mopsa/mopsa>

inference through coercion hierarchies, and position-level approximation of vector content.

What is missing from this field is an abstract domain that handles the combination of positional indexing, automatic coercion, and recycling that R vectors require. This thesis presents the first domain designed specifically for these semantics.

1.5.3 Array-Centric Analysis

Classical array domains for imperative languages address properties of array indices. Bounds-checking analyses [36] verify that all array accesses fall within allocated ranges, while dependence analyses for loop optimization [37] determine which array elements are read and written by which loop iterations. These techniques assume fixed-size arrays with statically declared element types and do not handle dynamic resizing, implicit type promotion, or the attribute metadata that R vectors carry.

More closely related are domains for dynamically-sized and content-abstracted collections. The FunArray parametric segmentation functor [23] automatically partitions arrays into consecutive, non-overlapping segments and abstracts each segment uniformly. The domain is parametric in the segment abstraction, allowing element domains to be plugged in, a design that directly inspired the parametric structure of the domain presented in Chapter 3. FunArray, however, targets .NET arrays in a statically typed setting and does not handle coercion, recycling, or the R-specific attribute system. The segmentation logic assumes memory-level array representations and fixed element types, neither of which holds for R.

Giet et al. [38] propose a product of shape and sequence abstractions that tracks length, extremal values, sortedness, and structural properties of linked data structures. Their domain targets systems programming contexts where type stability and predictable memory layouts are the norm. The Deskcheck framework [39] combines heap, array, and numeric domains to verify rich quantified invariants over complex data structures, using a shared predicate language inspired by Nelson–Oppen combination. It is powerful but assumes static typing and is designed for verification rather than lightweight static analysis.

Recent work on data science programs proposes an abstract domain that models dataframe transformations as a control-flow graph [40], enabling static analyses over data-processing pipelines by reasoning on transformation sequences rather than raw code. This work is relevant as a rare example of static analysis targeting data-science code, but it operates at the level of transformation DAGs and does not model the underlying vector semantics on which every transformation depends.

What remains absent is an array domain that handles the combination of runtime-determined length, implicit type coercion, recycling semantics, and attribute metadata that R vector analysis requires.

[36] Blanchet et al., 2003
A Static Analyzer for Large Safety-Critical Software,

[37] Maydan et al., 1993
Array-Data Flow Analysis and Its Use in Array Privatization,

[23] Cousot et al., 2011
A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis,

Chapter 3,

41

[38] Giet et al., 2023
A Product of Shape and Sequence Abstractions,

[39] McCloskey et al., 2010
Statically Inferring Complex Heap, Array, and Numeric Invariants,

[40] Negrini et al., 2023
Static Analysis of Data Transformations in Jupyter Notebooks,

Our domain is the first to address all four dimensions simultaneously.

1.6 Research Gap and Contributions

1.6.1 The Gap

R is one of the most widely used languages in data science, but its ecosystem of static analysis tools remains thin: despite recent advances such as `flowR`, no existing tool for the language provides a sound abstract domain for vectors. The vector is the fundamental data structure of the language: every value is a vector, every operation consumes and produces vectors, and the semantics of every operation depends on its operands' lengths, element types, and attributes. None of these properties are statically known in general R code. Existing tools for R, including `ROSA`, `codetools`, and the various optimizing interpreters, do not provide a sound abstraction of vectors. Existing abstract domains for collections, whether targeting heterogeneous objects in dynamic languages or fixed-size arrays in imperative settings, do not handle the unique combination of positional indexing, implicit type coercion, length recycling, and attribute metadata. This combination is what characterizes R's vector model. There is no parametric abstract domain for R vectors that tracks length, type, element values, and attributes with formal soundness guarantees. This thesis develops that domain.

1.6.2 Contributions

This thesis makes four contributions.

First, we define μR , a minimal core calculus for R vectors (Chapter 2). The calculus captures the essential vector operations of the language: construction, concatenation, type coercion, recycling, selection, and update. It provides a formal concrete semantics that serves as the reference for the abstract domain and the foundation for all soundness proofs. The calculus is intentionally minimal: it isolates the vector-level semantics from R's other complexities, including lazy evaluation, first-class environments, and metaprogramming, so that the domain can be developed and proven sound in a controlled setting against a clean formal baseline.

⌘ Chapter 2,

25

Second, we present a parametric abstract domain for R vectors (Chapter 3). The domain tracks four dimensions simultaneously: the length of the vector, a sequence of individually tracked element values, a summary approximation for all remaining positions, and the attribute metadata attached to the vector.

⌘ Chapter 3,

41

It is parametric in the element abstract domain: any lattice providing arithmetic, relational, and logical operations over scalar values can be plugged in, allowing the analysis to be tuned for different precision requirements. We prove that the domain forms a complete lattice, provide join, meet, and widening operators that guarantee termination of fixpoint iteration, and establish a formal Galois connection between the abstract domain and the concrete vector domain.

Third, we lift every μR operator to a sound abstract counterpart (Chapter 4). For each concrete operator, including concatenation, coercion, recycling, selection through positive, negative, and logical indices, and update through the same three index mechanisms, we define an abstract version. Formal soundness proofs are provided for the recycling and positive selection operators; their complete proofs appear in Appendix B. The remaining operators are defined and implemented, but their formal soundness proofs are left to future work.

⌘ Chapter 4,

55

Fourth, we implement the domain as a proof of concept and validate it empirically (Chapter 5). The implementation is built on top of flowR, which provides dataflow and control-flow graphs from R source code. We instantiate the abstract domain with an interval domain for numeric values and evaluate it on a suite of handcrafted μR test programs that systematically cover all six categories of vector operations. For each test case, we verify that the abstract result soundly over-approximates the concrete output produced by the R interpreter. We also confirm that the abstraction retains enough precision to produce informative results.

⌘ Chapter 5,

81

1.6.3 Scope and Limitations

The domain developed in this thesis models the vector layer of R semantics. It does not directly handle lazy evaluation, first-class environments, metaprogramming through `eval()` and `substitute()`, or S3 and S4 method dispatch, although the attribute component of the domain provides the information that a dispatch analysis would need. μR is a core calculus, not full R; porting the domain to cover more of R's surface syntax and additional language features is future work. The empirical validation uses handcrafted test programs, not real-world R packages, and measures soundness and precision on a per-operator basis rather than on end-to-end analysis tasks. The implementation is a proof of concept; performance, scalability, and integration into a production-quality analysis pipeline are not evaluated in this thesis.

1.7 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 defines μR , a minimal core calculus for R vectors, and presents its formal syntax and concrete operational semantics. This chapter establishes the notation, the mathematical model of vectors, and the reference semantics against which all abstract operators are proven sound.

⌘ Chapter 2,

25

Chapter 3 defines the parametric abstract domain. It introduces the domain's lattice structure, the four dimensions of vector abstraction, the join, meet, and widening operators, and the Galois connection with the concrete domain. This chapter contains the core theoretical contribution of the thesis.

⌘ Chapter 3,

41

Chapter 4 lifts each concrete μR operator to its abstract counterpart and proves soundness. It covers type coercion, attribute manipula-

⌘ Chapter 4,

55

tion, concatenation, recycling, selection, and update, organized by operator category. This is the longest and most technically dense chapter.

Chapter 5 describes the proof-of-concept implementation built on `flowR`, the instantiation with an interval domain, and the empirical validation on the μR test suite. It presents the test cases, the analysis results, and a discussion of precision loss in the interval abstraction.

↻ Chapter 5,

81

Chapter 6 summarizes the results, discusses the limitations identified above, and outlines directions for future work, including extension to a larger fragment of `R`, integration with analyses for lazy evaluation and metaprogramming, and evaluation on real-world `R` packages.

↻ Chapter 6,

90

Appendix A contains the full proofs of the abstract domain's structural properties: the equivalence-class construction, the partial order and its antisymmetry, the meet operator and its correctness, the Galois connection, and the widening operator's termination guarantee. Appendix B contains the complete soundness proofs for the abstract operators: recycling with its helper lemmas, and selection and update through all three index mechanisms.

Concrete Semantics of R Vectors

The concrete semantics of R 's vector operations are the reference against which any abstraction must be proven sound. Building on the informal overview of R vectors given in Chapter 1, we define the formal model of concrete R vectors and the key vector operations: type coercion, generic function dispatch, attribute manipulation, flattening concatenation, recycling, selection, and update. Chapter 3 and Chapter 4 lift these concrete definitions to a computable abstract domain via the Galois connection established in Theorem 3.8.2.

The chapter is organized as follows. Section 2.1 introduces μR , the core language whose expressions we analyze, and presents its syntax, while Section 2.3 defines the formal representation of concrete R vectors and the collecting semantics lattice. Section 2.4 formalizes type coercion, the mechanism that ensures type homogeneity, and Section 2.5 discusses R 's S3 generic function dispatch and its implications for our framework. Section 2.6 covers attribute manipulation via `setAttr` and `getAttr`, and Section 2.7 presents the flattening and concatenation operator $\oplus$. Section 2.8 formalizes vector recycling via `Recycle` and its auxiliary function ρ_c , while Section 2.9 and Section 2.10 define the selection and update operators, covering positive, negative, and logical indexing.

2.1 Syntax of μR Expressions

The μR is a core subset of R defined to be expressive enough to capture the most common vector operations. We omit aspects concerning environments, promises, and lazy evaluation, while focusing on vector operations including **Subsetting**, assignment, and structural properties such as attributes.¹

Explicit vector creation occurs via the language primitive $c()$, which can be called with zero or more arguments. For subsetting, we consider two operators:

- `[` (single bracket), used to select multiple items from a vector;
- `[[` (double bracket), used to extract a single item.

The `$` operator is excluded since it requires reasoning about specific attribute names, while in μR we do not model name-sensitive access mechanisms. Binary operations (arithmetic, relational, and logical) are all vectorized and coerce to a common type. Logical operators (`&` and `|`) act element-wise, evaluating both arguments. Attribute

1 Syntax of μR Expressions,	25
2 Syntax and Semantics of μR Statements,	28
3 Formal Vector Representation,	30
4 Type Coercion,	31
5 Generic Functions and Class Dispatch,	31
6 Attribute Manipulation,	32
7 Flattening and Concatenation,	32
8 Recycling,	33
9 Selection,	34
10 Update,	36
11 Chapter Summary,	39

⊗ Section 2.1, 25

⊗ Section 2.3, 30

⊗ Section 2.4, 31

⊗ Section 2.5, 31

⊗ Section 2.6, 32

⊗ Section 2.7, 32

⊗ Section 2.8, 33

⊗ Section 2.9, 34

⊗ Section 2.10, 36

🔗 **Subsetting.** R 's bracket operators `[` and `[[` for accessing and selecting elements from vectors. The single bracket returns a sub-vector (possibly of multiple elements), while the double bracket extracts a single element. Both support positive, negative, and logical indexing modes. Concrete selection and update semantics are formalized via `Select` and `Update`.

1. See <https://adv-r.hadley.nz/vectors-chap.html> for a detailed description of R 's vector data structures.

manipulation functions `setAttr` and `getAttr` abstract the behavior of R's attribute modifiers such as `names()`, `dim()`, and `class()`.

In the following², scalar values are modeled as vectors of length one. The special value `NULL` is interpreted as the empty vector of length zero. While this slightly differs from R's semantics, which treat `NULL` as a distinct value, correctness can be obtained by specifying simple edge cases (e.g., returning `FALSE` when used with R's `is.vector` check function).

2. For readability, multi-character code symbols are rendered as ligatures in the graphical output; the actual language tokens are the corresponding ASCII sequences.

E	$::=$	$Atm \mid \triangleright E$
		$e_1 \bowtie_{arith} e_2$
		$e_1 \bowtie_{rel} e_2$
		$e_1 \bowtie_{log} e_2$
		$c()$
		$c(E_1, \dots, E_n)$
		$x \leftarrow e$
		$x[E_{idx}]$
		$x[[E_{idx}]]$
		$x[E_{idx}] \leftarrow e$
		$x[[E_{idx}]] \leftarrow e$
		$setAttr(x, \{s_1, \dots, s_n\})$
		$getAttr(x)$
E_{idx}	$::=$	$\varepsilon \mid e$
Atm	$::=$	Var
		$Float$
		Int
		$Bool$
		$NULL$
$Bool$	$::=$	$TRUE$
		$FALSE$
		NA
$\bowtie_{arith}$	$::=$	$+ \mid - \mid * \mid /$
$\bowtie_{rel}$	$::=$	$> \mid < \mid = \mid \neq \mid \leq \mid \geq$
$\bowtie_{log}$	$::=$	$\& \mid \mid$
$\triangleright$	$::=$	$! \mid -$

Figure 2: Syntax of μR expressions.

μR includes production rules for atoms (variables, numeric and logical constants, `NULL`), unary operations (`!` for logical negation, `-` for arithmetic negation), binary operations (arithmetic `+`, `-`, `*`, `/` relational `>`, `<`, `=`, `≠`, `≤`, `≥` logical `&`, `|`), variable assignment, vector construction via `c()`, subsetting and extraction, assignment with subsetting, and attribute manipulation. The concrete semantics of these expression forms is given in Table 1.

We denote by $\mathbb{V}_{\text{Float}}$, $\mathbb{V}_{\text{Int}}$, $\mathbb{V}_{\text{Bool}}$, and Null the sets of floating-point values, integer values, Boolean values, and the null value, respectively. The set of concrete atomic values is $\mathbb{V}_{\text{Atm}} = \mathbb{V}_{\text{Float}} \cup \mathbb{V}_{\text{Int}} \cup \mathbb{V}_{\text{Bool}} \cup \{\text{Null}\}$. For brevity, we exclude other values such as complex numbers, as their inclusion would not significantly alter the framework's complexity. We use $\mathbb{N}$ to denote the set of positive natural numbers $\{1, 2, 3, \dots\}$, excluding zero, which correspond to the 1-indexed vector positions in R .

Concrete program states, called environments, are partial functions $\mathcal{E} : \text{Var} \rightarrow \mathcal{V}$ mapping identifiers to concrete vectors, where $\mathcal{V}$ denotes the set of all concrete vectors (formalized in Section 2.3). We write ρ for a typical environment.

⊗ Section 2.3,

30

$\mathbb{E}[e] : \mathcal{E} \rightarrow (\mathcal{E} \times (\mathcal{V} \cup \mathcal{P}(\text{String})))$, where $a \in \text{Atm}$, $x \in \text{Var}$, $e \in E$, $s \in \text{String}$

$$\begin{aligned}
 \mathbb{E}[a]\rho &\stackrel{\text{def}}{=} \begin{cases} (\rho, \rho(a)) & \text{if } a \in \text{Var} \\ (\rho, [a]) & \text{if } a \in \text{Float} \cup \text{Int} \cup \text{Bool} \\ (\rho, []) & \text{if } a \equiv \text{NULL} \end{cases} \\
 \mathbb{E}[-e]\rho &\stackrel{\text{def}}{=} (\rho', [-\text{ToNum}(c_1), \dots, -\text{ToNum}(c_k)]) \\
 &\quad \text{where } (\rho', [c_1, \dots, c_k]) = \mathbb{E}[e]\rho \\
 \mathbb{E}[\neg e]\rho &\stackrel{\text{def}}{=} (\rho', [\neg \text{ToBool}(c_1), \dots, \neg \text{ToBool}(c_k)]) \\
 &\quad \text{where } (\rho', [c_1, \dots, c_k]) = \mathbb{E}[e]\rho \\
 \mathbb{E}[e_1 \diamond e_2]\rho &\stackrel{\text{def}}{=} (\rho_2, v'_1 \diamond v'_2) \\
 &\quad \text{where } (\rho_1, v_1) = \mathbb{E}[e_1]\rho \\
 &\quad \quad (\rho_2, v_2) = \mathbb{E}[e_2]\rho_1 \\
 &\quad \quad (v'_1, v'_2) = \text{Recycle}((v_1, v_2)) \\
 &\quad \quad \diamond \in \{\bowtie_{\text{arith}}, \bowtie_{\text{rel}}, \bowtie_{\text{log}}\} \\
 \mathbb{E}[x \leftarrow e]\rho &\stackrel{\text{def}}{=} (\rho'[x \mapsto v], v) \\
 &\quad \text{where } (\rho', v) = \mathbb{E}[e]\rho \\
 \mathbb{E}[x[] \leftarrow e]\rho &\stackrel{\text{def}}{=} (\rho'[x \mapsto v'], v') \\
 &\quad \text{where } (\rho', v) = \mathbb{E}[e]\rho, \\
 &\quad \quad v' = \rho_c(v, |\rho'(x)|) \\
 \mathbb{E}[x[e_{\text{idx}}] \leftarrow e]\rho_0 &\stackrel{\text{def}}{=} (\rho_3, \rho_3(x)) \\
 &\quad \text{where } (\rho_1, v) = \mathbb{E}[e]\rho_0, \\
 &\quad \quad (\rho_2, v_{\text{idx}}) = \mathbb{E}[e_{\text{idx}}]\rho_1, \\
 &\quad \quad \rho_3 = \rho_2[x \mapsto \text{Update}(\rho_2(x), v_{\text{idx}}, v)] \\
 \mathbb{E}[x[[e_{\text{idx}}]] \leftarrow e]\rho &\stackrel{\text{def}}{=} \mathbb{E}[x[e_{\text{idx}}] \leftarrow e]\rho \\
 &\quad \text{where } (\rho', [c]) = \mathbb{E}[e_{\text{idx}}]\rho, c > 0 \\
 \mathbb{E}[c()]\rho &\stackrel{\text{def}}{=} (\rho, [])
 \end{aligned}$$

$\mathbb{E}[\mathbf{c}(e_1, \dots, e_n)]\rho_0$	$\stackrel{\text{def}}{=} (\rho_n, v_1 \oplus \dots \oplus v_n)$ where $\forall i \in [1, n]. (\rho_i, v_i) = \mathbb{E}[e_i]\rho_{i-1}$
$\mathbb{E}[\mathbf{x}[]]\rho$	$\stackrel{\text{def}}{=} \mathbb{E}[\mathbf{x}]\rho$
$\mathbb{E}[\mathbf{x}[e_{\text{idx}}]]\rho$	$\stackrel{\text{def}}{=} (\rho', \text{Select}(\rho'(\mathbf{x}), v_{\text{idx}}))$ where $(\rho', v_{\text{idx}}) = \mathbb{E}[e_{\text{idx}}]\rho$
$\mathbb{E}[\mathbf{x}[[e_{\text{idx}}]]]\rho$	$\stackrel{\text{def}}{=} (\rho', \text{Select}(\rho'(\mathbf{x}), [c]))$ where $(\rho', [c]) = \mathbb{E}[e_{\text{idx}}]\rho, c > 0$
$\mathbb{E}[\mathbf{getAttr}(\mathbf{x})]\rho$	$\stackrel{\text{def}}{=} (\rho, \mathcal{A})$ where $(c, \mathcal{A}) = \mathbb{E}[\mathbf{x}]\rho$
$\mathbb{E}[\mathbf{setAttr}(\mathbf{x}, \{s_1, \dots, s_n\})]\rho$	$\stackrel{\text{def}}{=} (\rho[\mathbf{x} \mapsto v'], v')$ where $(c, \mathcal{A}) = \mathbb{E}[\mathbf{x}]\rho,$ $v' = (c, \{s_1, \dots, s_n\})$

Table 1: Concrete semantics of μR expressions.

$\text{ToBool}/\overline{\text{ToBool}} : \mathbb{V}_{\text{Atm}} \rightarrow \mathbb{V}_{\text{Bool}}$

$$\text{ToBool}(a) \stackrel{\text{def}}{=} \begin{cases} \text{NA} & a = \text{NA} \\ \text{False} & a \in \{\text{False}, 0\} \\ \text{True} & \text{otherwise} \end{cases}$$

$$\overline{\text{ToBool}}(a) \stackrel{\text{def}}{=} \begin{cases} \text{NA} & a = \text{NA} \\ \neg \text{ToBool}(a) & \text{else} \end{cases}$$

$\text{ToNum} : \mathbb{V}_{\text{Atm}} \rightarrow \mathbb{V}_{\text{Atm}}$

$$\text{ToNum}(a) \stackrel{\text{def}}{=} \begin{cases} \text{NA} & a = \text{NA} \\ a & \tau(a) \neq \text{Bool} \\ 1 & a = \text{True} \\ 0 & \text{otherwise} \end{cases}$$

Table 2: Auxiliary functions for the concrete semantics.

2.2 Syntax and Semantics of μR Statements

The expression syntax and semantics of μR (Section 2.1) define how values are computed. This section defines the statement syntax and concrete semantics, completing the concrete language reference.

↻ Section 2.1,

25

2.2.1 Statement Syntax

Figure 3 extends the expression language of μR with conditionals, loops, sequencing, and block grouping.

$$\begin{aligned}
P &::= S \\
S &::= E \\
&\quad | \text{if } (E) \{S_1\} \text{ else } \{S_2\} \\
&\quad | \text{while } (E) \{S\} \\
&\quad | S_1 S_2 \\
&\quad | \{S\}
\end{aligned}$$

Figure 3: Syntax of μR statements.

A program P is a statement S . We define five statement forms, each with a distinct evaluation behavior. An expression statement E produces a side effect, i.e., the environment update, and discards the resulting vector value. Conditionals $\text{if } (E) \{S_1\} \text{ else } \{S_2\}$ branch on a Boolean guard. Loops $\text{while } (E) \{S\}$ repeat a statement while the guard evaluates to True. Sequential composition $S_1 S_2$ executes the first statement, then the second in the resulting environment. Block grouping $\{S\}$ provides explicit scoping and is semantically transparent: it evaluates to the same result as S .

2.2.2 Concrete Statement Semantics

The concrete semantics of statements is a function $\mathbb{S} : \mathcal{E} \rightarrow \mathcal{E}$ that maps a concrete environment to a concrete environment. Unlike expressions, statements do not produce a vector value; they transform the environment through side effects.

$\mathbb{S}[s] : \mathcal{E} \rightarrow \mathcal{E}, \quad \text{where } e \in E, s \in \text{String}$	
$\mathbb{S}[e]\rho$	$\stackrel{\text{def}}{=} \rho'$ where $(\rho', v) = \mathbb{E}[e]\rho$
$\mathbb{S}[s_1 s_2]\rho$	$\stackrel{\text{def}}{=} \mathbb{S}[s_2](\mathbb{S}[s_1]\rho)$
$\mathbb{S}[\text{if } (e) \{s_1\} \text{ else } \{s_2\}]\rho$	$\stackrel{\text{def}}{=} \begin{cases} \mathbb{S}[s_1]\rho' & \text{if } \text{ToBool}(v[1]) = \text{True} \\ \mathbb{S}[s_2]\rho' & \text{if } \text{ToBool}(v[1]) = \text{False} \end{cases}$ where $(\rho', v) = \mathbb{E}[e]\rho$
$\mathbb{S}[\text{while } (e) \{s\}]\rho$	$\stackrel{\text{def}}{=} \begin{cases} \mathbb{S}[s \text{ while } (e) \{s\}]\rho' & \text{if } \text{ToBool}(v[1]) = \text{True} \\ \rho & \text{if } \text{ToBool}(v[1]) = \text{False} \end{cases}$ where $(\rho', v) = \mathbb{E}[e]\rho$

Table 3: Concrete statement semantics of μR .

The expression statement e evaluates the expression to obtain an updated environment and a vector, then discards the vector. Sequential composition $s_1 s_2$ chains the environment transformation: evaluating s_2 in the environment produced by s_1 . The conditional $\text{if } (e) \{s_1\} \text{ else } \{s_2\}$ evaluates the guard expression, extracts the first element of the resulting vector, and branches on ToBool of that element. If the guard is True, the true branch s_1 is executed in the updated environment; otherwise the false branch s_2 is executed.

The while loop is expressed in unrolling form, which captures the operational intuition: when the guard holds, the body s is executed followed by the loop itself; when the guard fails, the environment is returned unchanged.

The concrete semantics above is single-state: each expression maps an environment to an environment-value pair, each statement maps an environment to an environment. The **Collecting Semantics** lifts this to sets of states by the standard powerset construction [1]. It is the target of the abstraction developed in Chapter 3, which establishes a Galois connection at the value level between the concrete vector domain $\mathcal{P}(\mathcal{V})$ and the abstract domain $\mathcal{V}_B^\#$.

2.3 Formal Vector Representation

We model a concrete R vector v as a pair $(\text{data}, \mathcal{A})$ where $\text{data} : \mathbb{N} \rightarrow \mathbb{V}_{\text{Atm}}$ is a total function mapping positive natural numbers to atomic values, and $\mathcal{A} \subseteq \text{Attr}$ is the set of known attributes of the vector. Access to the underlying data at position i is written as $v(i)$ and is equivalent to $v.\text{data}(i)$.

For readability, we represent a vector containing the sequence of atoms $c_1, \dots, c_n$ and the set of attributes A using the notation $[c_1, \dots, c_n]_A$. Formally, this notation defines the vector's internal state as:

$$[c_1, \dots, c_n]_A(i) \stackrel{\text{def}}{=} \begin{cases} c_i & \text{if } 1 \leq i \leq n \\ \text{Null} & \text{otherwise} \end{cases}$$

Using Null rather than NA as the out-of-bounds sentinel is essential: it allows us to distinguish between positions beyond the vector's length and legitimate vector elements, including NA itself (e.g., $[1, \text{NA}, 3]$). This design enables the length detection of a vector v via its **Active Domain**:

$$\text{dom}(v) \stackrel{\text{def}}{=} \{i \in \mathbb{N} \mid v(i) \neq \text{Null}\}$$

The length of a vector v , denoted $|v|$, is the number of non-Null elements: $|v| \stackrel{\text{def}}{=} |\text{dom}(v)|$. For vectors represented as $[c_1, \dots, c_n]_A$, this yields $|v| = n$.

When the set of attributes is empty we write $[c_1, \dots, c_n]$. The empty vector, denoted by $[]$, is the constant function returning Null for every input, with $||[]| = 0$.

For instance:

Example

Concrete Vectors

The vector $v = [10, 20, \text{NA}]_{\{\text{names}\}}$ has length 3, content $(10, 20, \text{NA})$, and the “names” attribute. Its data function satisfies $v(1) = 10, v(2) = 20, v(3) = \text{NA}$, and $v(i) = \text{Null}$ for all $i > 3$. The active domain is $\{1, 2, 3\}$.

Collecting Semantics. The standard powerset lift of the concrete semantics, operating over sets of program states. It associates each program point with the set of all possible environments that can reach it, defined as the least fixpoint of a monotone semantic function over a complete lattice of sets of states ordered by inclusion. The collecting semantics serves as the reference semantics that any abstraction must over-approximate.

[1] Cousot et al., 1977 *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*,

Chapter 3,

41

The empty vector $[]$ has $||[]| = 0$ and $\text{dom}([]) = \emptyset$.

2.4 Type Coercion

R is dynamically typed but enforces strict type homogeneity within vectors. When an operation involves vectors of different types, elements are implicitly coerced to a common type before the operation proceeds.

We define the set of atomic vector types $\mathcal{T} = \{\text{Bool}, \text{Int}, \text{Float}\}$, equipped with the total order $\text{Bool} \prec \text{Int} \prec \text{Float}$. This is a subset of R's full typing and mode lattice.³ The type of a concrete vector is obtained by the function $\tau : \mathcal{V} \rightarrow \mathcal{T}$. The least upper bound type of two vectors x and y is $\tau(x) \sqcup \tau(y)$, where $\sqcup$ denotes the supremum under $\prec$.

The semantic type coercion is specified through the helper function

$$\text{Cast} : \mathcal{V} \times \mathcal{T} \rightarrow \mathcal{V}$$

whose behavior is induced by $\prec$: each element of the input vector is individually coerced to the target type. For instance, when a Boolean vector $[\text{True}, \text{False}]$ is coerced to Float , the result is $[1, 0]$. The NA value propagates through coercion: coercing NA to any type yields NA of the target type. Although R provides distinct NA constants for each atomic type (e.g., `NA_integer_`, `NA_real_`), our model abstracts them into the single symbol NA; the concrete missing value is implicitly determined by the vector's type.⁴

We denote by $\mathcal{V}_{\text{Int}}$, $\mathcal{V}_{\text{Bool}}$, and $\mathcal{V}_{\text{Float}}$ the sets of integer, Boolean, and floating-point vectors, respectively.

To illustrate coercion:

Example

Concrete Coercion

Let $x = [\text{True}, \text{NA}]$ be a Boolean vector and $y = [1, 2]$ an integer vector. The common type is $\tau(x) \sqcup \tau(y) = \text{Int}$. Applying coercion:

$$\text{Cast}(x, \text{Int}) = [1, \text{NA}], \quad \text{Cast}(y, \text{Int}) = [1, 2]$$

The NA value in x remains NA after coercion, now of integer type.

2.5 Generic Functions and Class Dispatch

R employs the S3 object system, a generic function mechanism that enables object-oriented polymorphism: operations behave differently based on the “class” attribute of their arguments.⁵ A generic function dispatches to a specific method based on the class of its operands. For example, when the first argument of `c()` has `factor` as its first class, the function `c.factor` is called; it preserves the `levels` attribute when concatenating factors. For plain atomic vectors, no specific method exists and `c.default` is used, which only preserves the `names` attribute. Users can extend this mechanism by defining new classes and implementing corresponding methods.

3. R's full typing system includes character strings, complex numbers, raw bytes, lists, and S4 objects. Moreover, the full system does not form a consistent, operation-independent lattice: the coercion order for `c()` places `raw` at the bottom while comparison operators push it to the top, and many base types (closures, environments) do not participate in any coercion ordering. Our restricted subset $\{\text{Bool}, \text{Int}, \text{Float}\}$ avoids these issues; it is a finite total order, hence a complete lattice, and consistent across all relevant operations.

4. For instance, NA is a logical constant; if a vector contains only NA values, its type is `Bool` by default.

5. See <http://adv-r.had.co.nz/S3.html> for a detailed description of R's S3 object system.

Our framework describes the semantics of the default methods for atomic vectors. This may seem reductive, given that the language allows for the extension and overriding of the behavior of such functions; however, this is quite rare in practice and, more importantly, is not the focus of this work. Unless otherwise specified, the definitions in this chapter apply to vectors that do not have the “class” attribute.

2.6 Attribute Manipulation

R provides a rich set of functions for accessing and modifying vector attributes, which we capture through two helper functions:

$$\text{setAttr} : \mathcal{V} \times \mathcal{P}(\text{String}) \rightarrow \mathcal{V} \quad \text{getAttr} : \mathcal{V} \rightarrow \mathcal{P}(\text{String})$$

Here, $\text{setAttr}(v, A)$ returns v with its attribute set replaced by A , and $\text{getAttr}(v)$ returns the attribute set of v . The most commonly used attribute functions in R include:

- `names()` and `names<-()` for retrieving or assigning the names attribute;
- `dim()` and `dim<-()` for getting or setting dimensions, thereby converting vectors to matrices or arrays;
- `class()` and `class<-()` for accessing or modifying the class attribute that determines S3 method dispatch;
- `attr()` and `attr<-()` for general-purpose attribute access by name;
- `attributes()` and `attributes<-()` for manipulating the entire set of attributes at once;
- `levels()` specifically for factor vectors.

As discussed in Section 2.5, our framework models the default methods for atomic vectors that do not possess the “class” attribute.

↗ Section 2.5,

31

2.7 Flattening and Concatenation

The concatenation operator is fundamental to R vector construction. The primitive `c()` flattens nested vector inputs into a single linear sequence. We model this behavior via the binary operation $\oplus$ on vectors.

Definition 2.7.1

Concrete Concatenation

The operator $\oplus : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$ concatenates two atomic vectors, preserving only the “names” attribute, according to standard R semantic. Let x and y be concrete vectors, with $t = \tau(x) \sqcup \tau(y)$ being their common type. After coercing both operands to t via `Cast`, the result is the concatenation of their data components:

For non-empty vectors, given $[x_1, \dots, x_m]_{A_1}$ and $[y_1, \dots, y_n]_{A_2}$, let $t = \tau(x) \sqcup \tau(y)$ and define the coerced vectors $[x'_1, \dots, x'_m]_{A'_1} = \text{Cast}(x, t)$ and $[y'_1, \dots, y'_n]_{A'_2} = \text{Cast}(y, t)$. Then:

$$\begin{aligned}
[\] \oplus x &= x \oplus [\] \stackrel{\text{def}}{=} (x.\text{data}, x.A_1 \cap \{\text{names}\}) \\
[x_1, \dots, x_m]_{A_1} \oplus [y_1, \dots, y_n]_{A_2} &\stackrel{\text{def}}{=} [x'_1, \dots, x'_m, y'_1, \dots, y'_n]_{(A_1 \cup A_2) \cap \{\text{names}\}}
\end{aligned}$$

By abuse of notation, we use the same operator $\oplus$ to concatenate data components directly.

2.8 Recycling

R's binary operations are vectorized: when two vectors of different length appear as operands, the shorter one is implicitly **recycled**, i.e., its elements are reused cyclically until the target length is reached. This mechanism is also employed in selection and update operations with logical masks.

Definition 2.8.1

Concrete Recycling

The function $\text{Recycle} : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ aligns two vectors to a common length by recycling the shorter one. Empty vector cases:

$$\text{Recycle}([\], x) = \text{Recycle}(x, [\]) \stackrel{\text{def}}{=} ([\], [\])$$

For non-empty vectors, let $m = \max(|x|, |y|)$ be the target length. Let $t = \tau(x) \sqcup \tau(y)$ be the common type, and $x' = \text{Cast}(x, t)$, $y' = \text{Cast}(y, t)$. Then:

$$\text{Recycle}(x, y) \stackrel{\text{def}}{=} ((\rho_c(x'.\text{data}, m), x'.\mathcal{A}), (\rho_c(y'.\text{data}, m), y'.\mathcal{A}))$$

The auxiliary function ρ_c performs the actual data cycling.

Definition 2.8.2

Data Cycling

The function $\rho_c : (\mathbb{N} \rightarrow \mathbb{V}_{\text{Atm}}) \times \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{V}_{\text{Atm}})$ lengthens a data sequence to a target length l by repeated concatenation:

$$\rho_c([x_1, \dots, x_n], l) \stackrel{\text{def}}{=} \underbrace{[x_1, \dots, x_n] \oplus \dots \oplus [x_1, \dots, x_n]}_{\lfloor \frac{l}{n} \rfloor \text{ times}} \oplus [x_1, \dots, x_{l \bmod n}]$$

That is, the sequence is concatenated $\lfloor \frac{l}{n} \rfloor$ times, then the first $l \bmod n$ elements are appended to reach exactly length l .

For example:

Example

Concrete Recycling

Consider $x = [1, 2]$ and $y = [3, 4, 5]$. The target length is $m = \max(2, 3) = 3$. Then:

$$\text{Recycle}(x, y) = (\rho_c([1, 2], 3), \rho_c([3, 4, 5], 3))$$

$$\rho_c([1, 2], 3) = [1, 2] \oplus [1] = [1, 2, 1]$$

$$\rho_c([3, 4, 5], 3) = [3, 4, 5]$$

The result is $([1, 2, 1], [3, 4, 5])$. The shorter vector x is cyclically extended to length 3, while the longer vector y is unchanged.

2.9 Selection

The function `Select` formalizes the behavior of the `[]` operator, supporting three indexing modes: positive, negative, and logical.

$$\text{Select} : \mathcal{V} \times \mathcal{V}_{\text{Int}} \rightarrow \mathcal{V}$$

Without loss of generality, we take the second argument from integer interval vectors, since R implicitly casts any vector in a **Subsetting** operation to `Int`.

2.9.1 Positive Indexing

Positive indexing extracts elements at the specified positions. Out-of-bounds positions (where $v(i) = \text{Null}$) are converted to the missing value NA, while legitimate NA elements are preserved. The auxiliary function v^* handles this conversion.

Definition 2.9.1

Positive Selection

For all $c \in \mathcal{V}_{\text{Int}}$ such that $\forall i \in [1, |c|]. c(i) \geq 0 \vee c(i) = \text{NA}$, the selection is defined by concatenating the contributions of each index:

$$\text{Select}(v, c) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{|c|} \sigma_i$$

where the per-index result σ_i is determined by a case analysis:

$$\sigma_i \stackrel{\text{def}}{=} \begin{cases} [v^*(c(i))] & \text{if } c(i) > 0 \\ [] & \text{if } c(i) = 0 \\ [\text{NA}] & \text{if } c(i) = \text{NA} \end{cases}$$

and the out-of-bounds helper v^* maps `Null` to `NA` while preserving all other values:

$$v^*(i) \stackrel{\text{def}}{=} \begin{cases} v(i) & \text{if } v(i) \neq \text{Null} \\ \text{NA} & \text{otherwise} \end{cases}$$

The definition captures three behaviors. When $c(i) > 0$, the element at position $c(i)$ is extracted via v^* , which converts an out-of-bounds `Null` into `NA`. When $c(i) = 0$, no element is contributed (the empty vector acts as the neutral element of $\oplus$). When $c(i) = \text{NA}$, a single `NA`

is produced, following R's convention of propagating missingness through indexing operations.

2.9.2 Negative Indexing

Negative indexing returns the vector with specific indices excluded. The valid indices (in absolute value) are removed from the **Active Domain**, and the remaining elements are returned in increasing order of position.

Active Domain. The set of positions in a concrete R vector that contain a value other than the out-of-bounds sentinel Null. Formally, $\text{dom}(v) = \{i \in \mathbb{N} \mid v(i) \neq \text{Null}\}$. The length of a vector is the cardinality of its active domain, enabling formal distinction between missing data (NA) and positions beyond the vector's bounds.

Definition 2.9.2

Negative Selection

For all $c \in \mathcal{V}_{\text{Int}}$ such that $\forall i \in [1, |c|]. c(i) \leq 0$ and $c(i) \neq \text{NA}$, the selection is:

$$\text{Select}(v, c) \stackrel{\text{def}}{=} [v(u_1), \dots, v(u_k)]$$

where the exclusion set is constructed from the negative indices (with 0 always excluded):

$$O = \{-c(1), \dots, -c(|c|), 0\}$$

and $\langle u_1 \dots u_k \rangle$ is the increasing sequence obtained from $[1, |v|]$ by removing every element of O . That is, $1 \leq u_1 \leq \dots \leq u_k \leq |v|$ such that $\forall i \in [1, k], \forall j \in [1, |c|] : u_i \neq -c(j)$, with $k = |\{i \in [1, |v|] \mid i \notin O\}|$.

Since 0 belongs to the exclusion set O , it prevents 0-based indexing from selecting elements and ensures that the selection operates on the 1-indexed domain of R vectors. The resulting selection length is k , i.e., the number of positions not excluded.

2.9.3 Logical Indexing

Logical indexing selects elements corresponding to True values in a Boolean mask. The mask is recycled to match the data vector's length before filtering, and NA entries in the mask produce NA in the output.

Definition 2.9.3

Logical Selection

For all $c \in \mathcal{V}_{\text{Bool}}$, the selection is defined using the recycled mask c' :

$$\text{Select}(v, c) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{|c'|} \sigma_i$$

where $c' \stackrel{\text{def}}{=} \rho_c(c, \max(|v|, |c|))$ recycles the Boolean mask to the maximum of the two lengths, and:

$$\sigma_i \stackrel{\text{def}}{=} \begin{cases} [v^*(i)] & \text{if } c'(i) = \text{True} \\ [] & \text{if } c'(i) = \text{False} \\ [\text{NA}] & \text{if } c'(i) = \text{NA} \end{cases}$$

The recycling of the mask via ρ_c mirrors the behavior of binary operations (Section 2.8): a shorter mask is cyclically extended to match the data length. Positions where the mask is False are omitted. When the mask contains NA, a corresponding NA is emitted in the result, consistent with R's rule that missingness propagates through logical indexing.

↻ Section 2.8,

33

The following example illustrates the three modes.

Example

Concrete Selection

Consider the vector $v = [10, 20, 30, 40]$. The three indexing modes produce:

- $\text{Select}(v, [2, 5]) = [20, \text{NA}]$, since position 2 yields 20 and the out-of-bounds position 5 yields NA via v^* .
- $\text{Select}(v, [-1, -3]) = [20, 40]$, where positions 1 and 3 are excluded from v , leaving positions 2 and 4.
- $\text{Select}(v, [\text{True}, \text{False}]) = [10, 30]$, where the mask $[\text{True}, \text{False}]$ is recycled to $[\text{True}, \text{False}, \text{True}, \text{False}]$ to match $|v| = 4$, selecting positions 1 and 3.

2.10 Update

The function `Update` manages the assignment operation with subsetting, which allows specific elements of a vector to be modified based on positive, negative, or logical indices. The operation can extend the vector if assignment positions exceed the original length.

$$\text{Update} : \mathcal{V} \times \mathcal{V}_{\text{Int}} \times \mathcal{V} \rightarrow \mathcal{V}$$

As for the selection, we take the second argument from integer interval vectors.

We introduce a compact notation for positional updates. Given a vector r and sequences of positions $\langle u_1 \dots u_k \rangle$ and replacement values $\langle w_1 \dots w_k \rangle$, we write $r[u_i \mapsto w_i \mid i \in \{1, \dots, k\}]$ to denote the vector obtained from r by replacing the value at each position u_i with w_i . Positions not listed in the update remain unchanged. This notation is used throughout the three update modes defined below.

2.10.1 Positive Indexing

Positive indexing overwrites elements at specified positions. When the assignment positions exceed the original length of v , the vector is first extended with NA values to accommodate the new indices, then the target positions are updated. The replacement values are recycled to match the number of target positions.

Definition 2.10.1**Positive Update**

For all $c \in \mathcal{V}_{Int}$ such that $\forall i \in [1, |c|]. c(i) \geq 0 \vee c(i) = \text{NA}$, the update is:

$$\text{Update}(v, c, v_e) \stackrel{\text{def}}{=} r[u_i \mapsto v'_e(i) \mid i \in \{1, \dots, k\}]$$

where the components are defined in three steps. First, the vector is extended to accommodate the maximum index:

$$r \stackrel{\text{def}}{=} v \oplus \underbrace{[\text{NA}, \dots, \text{NA}]}_{\substack{\max(C-|v|, 0) \\ \text{times}}}, \quad C = \max\{c(i) \mid i \in [1, |c|] \wedge c(i) \neq \text{NA}\}$$

Second, the replacement values are recycled to match the number of target positions:

$$v'_e \stackrel{\text{def}}{=} \rho_c(v_e, k)$$

Third, $\langle u_1 \dots u_k \rangle$ is the increasing sequence obtained from $\{1, \dots, |c|\}$ by removing every index equal to 0. That is, $1 \leq u_1 \leq \dots \leq u_k \leq |c|$ such that $\forall i \in [1, k] : u_i \neq 0$, with $k = |\{i \in [1, |c|] \mid c(i) \neq 0\}|$.

The extension step fills the gap between the original length and the new indices with NA, mimicking R's behavior: when assigning to positions beyond the current length, intermediate positions are set to NA. The $\oplus$ operator is reused here with a sequence of NA values, preserving attributes as defined in Definition 2.7.1. The replacement values v_e are recycled via ρ_c to match exactly k target positions, where k excludes zero-index entries.

:= Definition,

32

2.10.2 Negative Indexing

Negative indexing identifies the subset of positions (by exclusion) on which to apply the update. The length of the original vector is invariant. The replacement values are recycled to match the number of non-excluded positions.

Definition 2.10.2**Negative Update**

For all $c \in \mathcal{V}_{Int}$ such that $\forall i \in [1, |c|]. c(i) \leq 0$ or $c(i) = \text{NA}$, the update is:

$$\text{Update}(v, c, v_e) \stackrel{\text{def}}{=} v[u_i \mapsto v'_e(i) \mid i \in \{1, \dots, k\}]$$

where $v'_e \stackrel{\text{def}}{=} \rho_c(v_e, k)$, the exclusion set is $O = \{-c(1), \dots, -c(|c|), 0\}$, and $\langle u_1 \dots u_k \rangle$ is the increasing sequence obtained from $\{1, \dots, |v|\}$ by removing every element of O . That is, $1 \leq u_1 \leq \dots \leq u_k \leq |v|$ such that $\forall i \in [1, k], \forall j \in [1, |c|] : u_i \neq -c(j)$, with $k = |\{i \in [1, |v|] \mid i \notin O\}|$.

Unlike positive update, the vector v is not extended: the length $|v|$ is invariant. The replacement values v_e are recycled to match k , the number of positions to update. Positions outside the non-excluded set retain their original values.

2.10.3 Logical Indexing

Logical indexing updates elements corresponding to True values in a Boolean mask. As in the positive case, the operation can extend the vector when the mask has more True entries than the current vector length.

Definition 2.10.3

Logical Update

For all $c \in \mathcal{V}_{\text{Bool}}$ such that $\forall i \in [1, |c|]. (c(i) \neq \text{NA} \vee |v_e| = 1)$, the update is:

$$\text{Update}(v, c, v_e) \stackrel{\text{def}}{=} r[u_i \mapsto v'_e(i) \mid i \in \{1, \dots, k\}]$$

where the recycled mask is $c' \stackrel{\text{def}}{=} \rho_c(c, \max(|v|, |c|))$, the extended vector is:

$$r \stackrel{\text{def}}{=} v \oplus \underbrace{[\text{NA}, \dots, \text{NA}]}_{\substack{\max(k-|v|, 0) \\ \text{times}}}$$

and the recycled replacement values are $v'_e \stackrel{\text{def}}{=} \rho_c(v_e, k)$. The target positions $\langle u_1 \dots u_k \rangle$ form the increasing sequence obtained from $\{1, \dots, |c'|\}$ by removing every index corresponding to a False entry in c' . That is, $1 \leq u_1 \leq \dots \leq u_k \leq |c'|$ such that $\forall i \in [1, k] : c'(u_i) \neq \text{False}$, with $k = |\{i \in [1, |c'|] \mid c'(i) = \text{True}\}|$.

The constraint $|v_e| = 1$ when c contains NA encodes R's semantic rule: an NA entry in a logical index represents an unknown truth value and the corresponding element may or may not be selected. When the replacement vector has length $|v_e| > 1$, the number of target positions is indeterminate, since it depends on how many NA entries would resolve to True. R's semantics therefore restricts the replacement to a scalar, which recycles identically regardless of the number of selected positions.

Consider a concrete case.

Example

Concrete Update

Consider the vector $v = [10, 20, 30]$. The three update modes produce:

- $\text{Update}(v, [5], [99]) = [10, 20, 30, \text{NA}, 99]$, where the vector is extended from length 3 to accommodate index 5, the gap at position 4 is filled with NA, and the value 99 is placed at position 5.

- $\text{Update}(v, [-2], [5]) = [5, 20, 5]$, where position 2 is excluded, so the replacement value 5 updates positions 1 and 3 after recycling.
- $\text{Update}(v, [\text{True}, \text{False}], [0]) = [0, 20, 0]$, where the logical mask is recycled to $[\text{True}, \text{False}, \text{True}]$, selecting positions 1 and 3 for update with the recycled value 0.

2.11 Chapter Summary

This chapter defined the concrete semantics of R vector operations that serve as the reference model for our static analysis framework. We introduced μR , the core language whose expressions we analyze (Section 2.1), and formalized the representation of concrete vectors as total functions paired with attribute sets (Section 2.3). We then defined the key auxiliary operations that encode R's vector semantics: type coercion to ensure type homogeneity (Section 2.4), attribute manipulation via `setAttr` and `getAttr` (Section 2.6), flattening and concatenation via $\oplus$ (Section 2.7), and vector recycling via `Recycle` and ρ_c (Section 2.8). Each operator was presented with its concrete definition and illustrated through examples.

↻ Section 2.1,	25
↻ Section 2.3,	30
↻ Section 2.4,	31
↻ Section 2.6,	32
↻ Section 2.7,	32
↻ Section 2.8,	33

We then defined the selection operator `Select` (Section 2.9), which extracts sub-vectors using positive, negative, or logical indices, and the update operator `Update` (Section 2.10), which modifies vectors in place (possibly extending them) through the same three index mechanisms. These concrete operators are lifted to the abstract domain in Chapter 4, where each abstract counterpart is proved sound with respect to the concretization function established in Chapter 3.

↻ Section 2.9,	34
↻ Section 2.10,	36
↻ Chapter 4,	55
↻ Chapter 3,	41

2.11.1 Abstraction Challenges

The definitions above expose several behaviors that resist straightforward lifting to the abstract domain. Three patterns recur across operators and deserve attention, as they inform the design of the abstract domain presented in Chapter 3.

↻ Chapter 3,	41
--------------	----

Zero-index discarding. In positive update (Definition 2.10.1) and positive selection (Definition 2.9.1), indices equal to zero are silently ignored. The concrete semantics discards them when constructing the sequence of target positions $u_1, \dots, u_k$, but an abstract counterpart must handle the case where the content domain cannot determine whether a given index is zero. When the analysis lacks this information, the corresponding position must be considered both potentially updated (or selected) and potentially unchanged, forcing a join that degrades precision.

:= Definition,	37
:= Definition,	34

Runtime-dependent index values. Both selection and update branch on the actual values of index vectors at runtime. Positive selection (Definition 2.9.1) distinguishes $c(i) > 0$, $c(i) = 0$, and $c(i) = \text{NA}$; logical selection (Definition 2.9.3) branches on $c'(i) = \text{True}$, $c'(i) = \text{False}$, and $c'(i) = \text{NA}$. When the abstract content domain cannot exclude any of these cases (for instance, when the index vector is the result of a previous computation), the abstract operator must take the join of

:= Definition,	34
:= Definition,	35

all branches. This collapses precise positional information, pushing the affected entries into the summary element of the abstract vector.

Negative indexing and exclusion precision. Negative selection (Definition 2.9.2) and negative update (Definition 2.10.2) construct an exclusion set from the absolute values of index entries. The length of the result depends on how many positions survive exclusion, which in turn depends on knowing the precise values in the index vector. When those values are only known imprecisely, the exclusion set is itself imprecise: the analysis cannot determine with certainty which positions are removed. Surviving positions that would be tractable with precise indices are then collapsed into the summary, losing positional fidelity.

These patterns share a common root: the concrete semantics inspect individual index values to determine control flow, while the abstract domain must conservatively approximate all possible outcomes. The hybrid design of our abstract vector, which pairs a known content with a fallback summary element, is a direct response to this tension, as discussed in Section 3.3.

:= Definition,
:= Definition,

35
37

☞ Section 3.3,

43

The Abstract Domain: Definition and Lattice Structure

The design of an abstract domain for R vectors must balance precision against computability while accommodating the language's dynamic length changes, implicit type coercion, and recycling semantics. This chapter presents the parametric abstract domain that achieves this balance. We define the domain structure, establish its lattice properties, and prove the existence of a Galois connection at the value level.

The chapter is organized as follows. Section 3.1 introduces the concrete value domain, and Section 3.2 introduces the three component domains that serve as building blocks. Section 3.3 discusses the design choices underlying our domain shape. Section 3.4 presents the formal definition of the abstract vector domain, including the equivalence relation and quotient construction, Section 3.5 defines the partial order and establishes that it forms a complete lattice, and Section 3.6 gives the join and meet operators. Section 3.7 introduces widening and proves its well-foundedness, and Section 3.8 defines concretization and establishes the Galois connection.

3.1 The Concrete Value Domain

We consider the complete lattice $\langle \mathcal{P}(\mathcal{V}), \subseteq \rangle$ as the concrete value domain of interest, where $\mathcal{P}(\mathcal{V})$ is the powerset of concrete R vectors presented in Chapter 2.

Recall that concrete program states map variables to vectors ($\mathcal{E} : \text{Var} \rightarrow \mathcal{V}$). The collecting semantics, the standard powerset lift of the deterministic semantics, operates over sets of such states but is left implicit. Following standard practice [1], we work directly with the value-level domain $\mathcal{P}(\mathcal{V})$.

The abstract state domain $\text{Var} \rightarrow \mathcal{V}_B^\sharp$ lifts this value-level domain pointwise: each variable's vectors are abstracted independently into $\mathcal{V}_B^\sharp$ via the Galois connection $\mathcal{P}(\mathcal{V}) \xrightleftharpoons[\alpha_{\mathcal{V}_B^\sharp}]{\gamma_{\mathcal{V}_B^\sharp}} \mathcal{V}_B^\sharp$ established in Theorem 3.8.2 with formal soundness proofs for recycling and positive selection in Chapter 4. Because all operator soundness proofs are per-vector, the state-level abstraction is orthogonal to the value domain. The thesis instantiates the non-relational lift; a relational state domain could reuse the same value-level results without modification.

1	The Concrete Value Domain,	41
2	Prerequisites: The Component Domains,	42
3	Design Rationale,	43
4	The Abstract Vector Domain,	44
5	Order Structure,	47
6	Join and Meet,	49
7	Widening and Termination,	50
8	Concretization and Galois Connection,	53
9	Chapter Summary,	54

⊗	Section 3.1,	41
⊗	Section 3.2,	42
⊗	Section 3.3,	43
⊗	Section 3.4,	44
⊗	Section 3.5,	47
⊗	Section 3.6,	49
⊗	Section 3.7,	50
⊗	Section 3.8,	53

⊗	Chapter 2,	25
---	------------	----

[1] Cousot et al., 1977
Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,

□	Theorem,	54
⊗	Chapter 4,	55

3.2 Prerequisites: The Component Domains

Our abstract domain is built from three independent components: a length domain, a parametric content domain, and an attribute domain. Each is a complete lattice, and their Cartesian product yields the abstract vector domain.

3.2.1 Length: Positive Integer Intervals

The length of an R vector is a non-negative integer. We abstract it using the positive integer interval domain $\langle \text{Itv}_0, \sqsubseteq_{\text{Itv}_0}, \sqcup_{\text{Itv}_0}, \sqcap_{\text{Itv}_0}, \perp_{\text{Itv}_0}, \top_{\text{Itv}_0} \rangle$, where Itv_0 consists of intervals $[a, b]$ with $a, b \in \mathbb{Z}_{\geq 0}$ or $[a, +\infty]$. The partial order is interval inclusion: $[a_1, b_1] \sqsubseteq_{\text{Itv}_0} [a_2, b_2]$ iff $a_2 \leq a_1$ and $b_1 \leq b_2$. The bottom element $\perp_{\text{Itv}_0}$ represents the empty set of lengths (unreachable), while $\top_{\text{Itv}_0} = [0, +\infty]$ represents all possible lengths.

3.2.2 Content: Parametric Abstract Domain

The content of each vector element is abstracted using a generic abstract domain $\langle \mathcal{B}^\sharp, \sqsubseteq_{\mathcal{B}^\sharp}, \sqcup_{\mathcal{B}^\sharp}, \sqcap_{\mathcal{B}^\sharp}, \perp_{\mathcal{B}^\sharp}, \top_{\mathcal{B}^\sharp} \rangle$, which approximates the set of atomic values $\mathcal{P}(\mathbb{V}_{\text{Atm}})$. This domain is **parametric**: it can be instantiated with any abstract domain providing the required lattice operations and value abstractions.

The abstract value domain $\mathcal{B}^\sharp$ must provide:

- Abstraction and concretization functions: $\alpha_{\mathcal{B}^\sharp} : \mathcal{P}(\mathbb{V}_{\text{Atm}}) \rightarrow \mathcal{B}^\sharp$ and $\gamma_{\mathcal{B}^\sharp} : \mathcal{B}^\sharp \rightarrow \mathcal{P}(\mathbb{V}_{\text{Atm}})$
- Lattice operators: $\sqsubseteq_{\mathcal{B}^\sharp}, \sqcup_{\mathcal{B}^\sharp}, \sqcap_{\mathcal{B}^\sharp}$
- Widening: $\nabla_{\mathcal{B}^\sharp}$
- Arithmetic, relational, and logical operations on abstract values

We write $\alpha_{\mathcal{B}^\sharp}(v)$ for the abstraction of a concrete atomic value $v \in \mathcal{P}(\mathbb{V}_{\text{Atm}})$ into $\mathcal{B}^\sharp$. We write $\diamond_{\mathcal{B}^\sharp}$ generically to denote the operation selected by a binary operator $\diamond^\sharp$.

3.2.3 Attributes: Must/May Pairs

Definition 3.2.1

Attributes Abstract Domain

We define the complete lattice

$$\langle \text{Attr}^\sharp, \sqsubseteq_{\text{Attr}^\sharp}, \sqcup_{\text{Attr}^\sharp}, \sqcap_{\text{Attr}^\sharp}, \perp_{\text{Attr}^\sharp}, \top_{\text{Attr}^\sharp} \rangle$$

where:

- $\text{Attr}^\sharp = \{(\text{MustHave}, \text{MayHave}) \in \mathcal{P}(\text{Attr}) \times \mathcal{P}(\text{Attr}) \mid \text{MustHave} \subseteq \text{MayHave}\} \cup \{\perp_{\text{Attr}^\sharp}\}$, and $\text{Attr} = \{\text{names}, \text{dim}, \text{class}, \text{other}\}$ denotes the set of possible vector attributes.
- The partial order is defined as $(\text{MustHave}_1, \text{MayHave}_1) \sqsubseteq_{\text{Attr}^\sharp} (\text{MustHave}_2, \text{MayHave}_2) \iff \begin{cases} \text{MustHave}_2 \subseteq \text{MustHave}_1 \\ \text{MayHave}_1 \subseteq \text{MayHave}_2 \end{cases}$, with $\perp_{\text{Attr}^\sharp}$ being below all elements.

- The top element is $\top_{\text{Attr}^\sharp} = (\emptyset, \text{Attr})$, representing the least precise information where any attribute may be present.

The component **MustHave** captures the attributes that the vector **definitely** possesses, while **MayHave** represents the attributes it **may** have. Intuitively, **MustHave** provides a lower bound and **MayHave** an upper bound on the set of attributes, with the constraint $\text{MustHave} \subseteq \text{MayHave}$ ensuring consistency: the definite attributes are always a subset of the possible ones.

Two particularly relevant elements are $\text{AttrEmpty} \stackrel{\text{def}}{=} (\emptyset, \emptyset)$, representing vectors with no attributes, and $\text{AttrFull} \stackrel{\text{def}}{=} (\text{Attr}, \text{Attr})$, representing vectors that definitely have all attributes.

3.3 Design Rationale

Before presenting the formal definition, we discuss the design choices that led to our domain shape. The fundamental challenge in abstracting R vectors is the tension between precision and tractability.

A naive approach would track every position of a vector individually. For a vector of length n , this requires n abstract elements. Since n may be unbounded (e.g., vectors read from user input), this approach is intractable: the abstract state space would be infinite, and no fixed-point computation could terminate.

At the other extreme, one could collapse all positions into a single summary abstract element, losing all positional information. While tractable, this is too imprecise for R code, where operations frequently access specific positions (e.g., $x[1] + x[2]$).

Existing array domains offer intermediate solutions. **FunArray** [23] uses segmentation, partitioning the array into homogeneous regions. **Shape analysis** [38] tracks dimensions and element types but not individual values. Parametric segmentation domains combine both but are designed for languages with static array bounds, not R's dynamic length changes.

Our design is a hybrid: we maintain a finite sequence of explicitly tracked positions (the **known content**), combined with a single **summary** abstract element that over-approximates all remaining positions. This design is motivated by three observations about R:

1. *Statically identifiable indices.* R code often uses literal indices (e.g., $x[1]$, $x[3]$), which our domain can track precisely in the known content.
2. *Unbounded length.* Vectors obtained from I/O or accumulation loops have no statically known upper bound. The summary element is valorized only in this case, providing a compact over-approximation of all positions beyond the known content.
3. *Variable length.* Even within finite bounds, vectors can vary in length across execution paths. A precise content is essential

[23] Cousot et al., 2011
A Parametric Segmentation
Functor for Fully Automatic
and Scalable Array Content
Analysis,

[38] Giet et al., 2023
A Product of Shape and
Sequence Abstractions,

The length interval $[l, u]$ captures this compactly; the known content extends to the upper bound and pads missing positions from shorter concretizations via the join operator.

The following example illustrates this trade-off.

Example

A Fixed Prefix is Not Enough

Consider two branches that produce similar vectors, analyzed with the interval instantiation of $\mathcal{B}^\sharp$:

```
if (cond) {
  x ← c(0, 0)           # length 2
} else {
  x ← c(0, 0, 1, 100)  # length 4
}
```

A fixed prefix of length 2 would track $(0, 0)$ precisely in the known content and dump everything else into a summary. Since the summary is a single abstract value, positions 3 and 4 both get $[1, 100]$, losing the information that position 3 is always 1. The summary collapses positions that could have been distinguished.

Instead of a fixed prefix, our domain tracks every position individually whenever the vector length has a known finite upper bound, and only valorizes the summary when the bound is $+\infty$.

3.4 The Abstract Vector Domain

We define the R vector abstract domain as the Cartesian product of four components: (i) the positive integer interval domain Itv_0 approximating the length of vectors, (ii) a sequence of abstract elements of $\mathcal{B}^\sharp$ representing the known content, (iii) a summary abstract element of $\mathcal{B}^\sharp$, and (iv) the attributes abstract domain $\text{Attr}^\sharp$.

Informally, the idea is to maintain an abstraction for those indices that are known to exist according to the current interval length abstraction, storing one abstract element of $\mathcal{B}^\sharp$ for each explicitly tracked position of the vector. This allows the analysis to preserve precise information about individual entries whenever their index is statically identifiable. The summary element provides a single abstract element of $\mathcal{B}^\sharp$ that over-approximates the values of all remaining positions not explicitly represented in the sequence. The summary is only valorized when the length interval has no upper bound, representing vectors that may have infinitely many positions beyond the known ones.

Definition 3.4.1**Abstract Vector Element**

An abstract element is a tuple $\nu = ([l, u], \langle p_1 \dots p_k \rangle, s, a)$ where:

- $[l, u] \in \text{Itv}_0$ is an interval abstraction of the vector's **length**,
- $\langle p_1 \dots p_k \rangle \in \mathcal{B}^{\#*}$ is a finite sequence representing the **known content**,
- $s \in \mathcal{B}^{\#}$ is a **summary** abstract value, valorized for infinite vectors to abstract unknown positions,
- $a \in \text{Attr}^{\#}$ is an **attribute** abstraction representing the set of attributes the vector may possess.

The raw set of such tuples must satisfy the domain invariant:

$$\tilde{\mathcal{V}}_{\mathcal{B}}^{\#} \stackrel{\text{def}}{=} \left\{ ([l, u], \langle p_1 \dots p_k \rangle, s, a) \mid \begin{array}{l} [l, u] \in \text{Itv}_0, \langle p_1 \dots p_k \rangle \in \mathcal{B}^{\#*}, \\ s \in \mathcal{B}^{\#}, a \in \text{Attr}^{\#}, l \leq k \leq u, \\ (u \neq +\infty \implies s = \perp_{\mathcal{B}^{\#}} \wedge u = k) \end{array} \right\}$$

The invariant ensures that the known content length k is consistent with the length interval $[l, u]$, and that the summary is bottom ($\perp_{\mathcal{B}^{\#}}$) whenever the length is finite (in which case all positions are known).

Definition 3.4.2**Equivalence Relation**

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ with $k_1 = |\mathbf{p}_1|$ and $k_2 = |\mathbf{p}_2|$. Then $\nu_1 \equiv \nu_2 \iff L_1 = L_2, s_1 = s_2, a_1 = a_2$, and one of the following holds:

- $k_1 = k_2$ and $\forall i \in [1, k_1] : p_{1,i} = p_{2,i}$
- $k_1 > k_2$ and $\forall i \in [1, k_2] : p_{1,i} = p_{2,i}$ and $\forall j \in [k_2 + 1, k_1] : p_{1,j} = s_1$
- $k_2 > k_1$ and $\forall i \in [1, k_1] : p_{1,i} = p_{2,i}$ and $\forall j \in [k_1 + 1, k_2] : p_{2,j} = s_2$

Intuitively, two raw elements are equivalent if they agree on the length, summary, and attributes, and their known content sequences agree on the common prefix, with the longer sequence's extra positions equal to its summary. This captures the fact that trailing positions equal to the summary carry no additional information.

Lemma 3.4.1**Equivalence Relation**

The relation $\equiv$ defined in Definition 3.4.2 is an equivalence relation on $\tilde{\mathcal{V}}_{\mathcal{B}}^{\#}$.

:= Definition,

45

Proof

Reflexivity and symmetry follow directly from the definition. For transitivity, the key case $k_1 < k_2 < k_3$ propagates equality through

the middle vector: $p_{1,i} = p_{2,i}$ and $p_{2,j} = s$ combine with $p_{2,i} = p_{3,i}$ and $p_{3,j} = s$ to yield the $k_1 < k_3$ condition. The remaining length combinations are similar or trivial. The full proof is in Proof (of Lemma 3.4.1).

■ Proof,

94

□

Definition 3.4.3**Set of Abstract Vectors**

The abstract domain is the set of equivalence classes:

$$\mathcal{V}_B^\# \stackrel{\text{def}}{=} \left\{ [\nu]_{\equiv} \mid \nu \in \tilde{\mathcal{V}}_B^\# \right\} \cup \left\{ \perp_{\mathcal{V}_B^\#}, \top_{\mathcal{V}_B^\#} \right\}$$

For each class $[\nu]_{\equiv}$, the **minimum representative** $\nu^\downarrow$ is obtained by removing all trailing positions of p equal to s , while keeping at least l positions. When we write $\nu \in \mathcal{V}_B^\#$, we abuse notation and mean the class $[\nu]_{\equiv}$ represented by its minimum representative $\nu^\downarrow$.

Definition 3.4.4**Abstract Domain of R Vectors**

The abstract domain for finite R vectors $\mathcal{D}_{\mathcal{V}_B^\#}$ is the complete lattice:

$$\mathcal{D}_{\mathcal{V}_B^\#} \stackrel{\text{def}}{=} \left\langle \mathcal{V}_B^\#, \sqsubseteq_{\mathcal{V}_B^\#}, \sqcup_{\mathcal{V}_B^\#}, \sqcap_{\mathcal{V}_B^\#}, \perp_{\mathcal{V}_B^\#}, \top_{\mathcal{V}_B^\#} \right\rangle$$

The special elements of the domain are:

- $\perp_{\mathcal{V}_B^\#} \stackrel{\text{def}}{=} (\perp_{\text{ltv}_0}, \varepsilon, \perp_{\mathcal{B}^\#}, \perp_{\text{Attr}^\#})$, the bottom element
- $\top_{\mathcal{V}_B^\#} \stackrel{\text{def}}{=} (\top_{\text{ltv}_0}, \varepsilon, \top_{\mathcal{B}^\#}, \top_{\text{Attr}^\#})$, the top element

Note that the abstraction of the empty vector $[\]$ differs from $\perp_{\mathcal{V}_B^\#}$ by the length element, which is the valid interval $[0, 0]$, and by the attribute abstraction, which is empty:

$$\alpha_{\mathcal{V}_B^\#}([\]) \stackrel{\text{def}}{=} ([0, 0], \varepsilon, \perp_{\mathcal{B}^\#}, \text{AttrEmpty})$$

Throughout this thesis, we use p to denote the entire known content sequence, and p_i (or $p_{\nu,i}$ when referring to a specific vector ν) to denote the abstract value at position i within that sequence. When multiple vectors are in scope, we write $p_1 = \langle p_{1,1} \cdots p_{1,k_1} \rangle$ and $p_2 = \langle p_{2,1} \cdots p_{2,k_2} \rangle$. Moreover in our examples, if not differently specified, we instantiate $\mathcal{B}^\#$ with the interval abstract domain ltv .

For instance:

Example**Abstracting Concrete Vectors**

Consider the concrete vector $[1, 2, 3]$, which has length 3, content $\langle 1, 2, 3 \rangle$, and no attributes. Using the interval instantiation for $\mathcal{B}^\#$, its best abstraction is:

$$\nu_1 = ([3, 3], \langle [1, 1], [2, 2], [3, 3] \rangle, \perp_{\text{ltv}}, \text{AttrEmpty})$$

Now consider a vector whose length is known to be at least 2, with the first two elements definitely 1 and 2, and the remaining ones non zero integers. Its abstraction is:

$$\nu_2 = ([2, 4], \langle [1, 1], [2, 2] \rangle, [1, +\infty], \text{AttrEmpty})$$

3.5 Order Structure

Having defined the abstract domain, we now equip it with a partial order that captures the precision of abstract elements. An abstract element ν_1 is less than or equal to ν_2 if every concrete vector represented by ν_1 is also represented by ν_2 .

Definition 3.5.1

Partial Order

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ be minimum representatives with $k_1 = |\mathbf{p}_1|$ and $k_2 = |\mathbf{p}_2|$. The partial order on $\mathcal{V}_{\mathcal{B}}^\sharp$ is defined as:

$$\nu_1 \sqsubseteq_{\mathcal{V}_{\mathcal{B}}^\sharp} \nu_2 \iff L_1 \sqsubseteq_{\text{ltv}_0} L_2 \wedge s_1 \sqsubseteq_{\mathcal{B}^\sharp} s_2 \wedge a_1 \sqsubseteq_{\text{Attr}^\sharp} a_2 \wedge K(\nu_1, \nu_2)$$

where $K(\nu_1, \nu_2)$ is the following case distinction on the known content:

$$K(\nu_1, \nu_2) \stackrel{\text{def}}{=} \begin{cases} \forall i \in [1, k_1] : p_{1,i} \sqsubseteq_{\mathcal{B}^\sharp} p_{2,i} & \text{if } k_1 \leq k_2 \wedge u_1 \neq +\infty \\ \forall i \in [1, k_1] : p_{1,i} \sqsubseteq_{\mathcal{B}^\sharp} p_{2,i} \\ \wedge \forall j \in [k_1 + 1, k_2] : s_1 \sqsubseteq_{\mathcal{B}^\sharp} p_{2,j} & \text{if } k_1 \leq k_2 \wedge u_1 = +\infty \\ \forall i \in [1, k_2] : p_{1,i} \sqsubseteq_{\mathcal{B}^\sharp} p_{2,i} \\ \wedge \forall j \in [k_2 + 1, k_1] : p_{1,j} \sqsubseteq_{\mathcal{B}^\sharp} s_2 & \text{if } k_1 > k_2 \wedge u_2 = +\infty \end{cases}$$

The three cases of K handle different relative lengths and finiteness conditions:

1. **Finite length, shorter or equal known content:** All positions of ν_1 must be bounded by corresponding positions in ν_2 . Since ν_1 has finite length, no positions beyond k_1 exist.
2. **Infinite length, shorter or equal known content:** In addition to the common prefix, the extra positions in ν_2 must bound the summary s_1 , which over-approximates all unknown positions of ν_1 .
3. **Longer known content, infinite target:** The common prefix is bounded as before, and the extra positions in ν_1 must be bounded by the summary s_2 of ν_2 .

Note that, even if the content domain $\mathcal{B}^\sharp$ is finite (e.g., the sign abstract domain), it is always possible to consider an AC of elements from $\mathcal{V}_{\mathcal{B}}^\sharp$. As a concrete instance:

⊞ **AC, Ascending Chain.** In a poset $(P, \leq)$, an ascending chain is a sequence of elements where each term is greater than or equal to the previous one under the partial order, i.e., $x_0 \leq x_1 \leq x_2 \leq \dots$.

Example

Consider the sequence:

$$\begin{aligned}
 & ([1, 1], \langle [0, 0] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 & \sqsubseteq_{\mathcal{V}_{\text{Itv}}^\#} ([1, 2], \langle [0, 0][0, 0] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 & \sqsubseteq_{\mathcal{V}_{\text{Itv}}^\#} ([1, 3], \langle [0, 0][0, 0][0, 0] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 & \sqsubseteq_{\mathcal{V}_{\text{Itv}}^\#} \dots
 \end{aligned}$$

where $x_i = ([1, i], \langle [0, 0] \dots [0, 0] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $x_{i+1} = ([1, i + 1], \langle [0, 0] \dots [0, 0] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$.

This justifies the introduction of the widening operator in Section 3.7, which guarantees convergence of the analysis while keeping a sound over-approximation of the least fix point.

↻ Section 3.7,

50

Theorem 3.5.1**Partial Order**

The relation $\sqsubseteq_{\mathcal{V}_B^\#}$ is a partial order on $\mathcal{V}_B^\#$.

Proof

Reflexivity follows from the component-wise orders. **Anti-symmetry** splits into finite and infinite length cases: when both vectors are finite they share the same fixed length; in the infinite case, trailing positions equal to the summary are removed by minimization, forcing the sequences to coincide. **Transitivity** composes the ordering constraints, propagating summary bounds across the middle vector through case analysis on the relative positions of the known sequences. The full proof is in Proof (of Theorem 3.5.1).

■ Proof,

95

□

To illustrate the partial order, consider:

Example**Comparing Abstract Elements**

Let $\nu_1 = ([2, 3], \langle [1, 1], [2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $\nu_2 = ([2, 4], \langle [1, 1], [2, 2], [3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$. Both have finite upper bounds ($u_1 = 3$, $u_2 = 4$), so their summaries are $\perp_{\text{Itv}}$ (the summary is only valorized when the length is unbounded). We verify $\nu_1 \sqsubseteq_{\mathcal{V}_{\text{Itv}}^\#} \nu_2$:

- Length: $[2, 3] \sqsubseteq_{\text{Itv}_0} [2, 4]$ since $2 \leq 2$ and $3 \leq 4$.
- Summary: $\perp_{\text{Itv}} \sqsubseteq_{\text{Itv}} \perp_{\text{Itv}}$
- Attributes: $\text{AttrEmpty} \sqsubseteq_{\text{Attr}^\#} \text{AttrEmpty}$
- Known content: $k_1 = 2 \leq k_2 = 3$ and $u_1 = 3 \neq +\infty$, so the first case of K applies. We need $p_{1,1} \sqsubseteq_{\text{Itv}} p_{2,1}$ and $p_{1,2} \sqsubseteq_{\text{Itv}} p_{2,2}$, i.e., $[1, 1] \sqsubseteq_{\text{Itv}} [1, 1]$ and $[2, 2] \sqsubseteq_{\text{Itv}} [2, 2]$, both true.

Hence $\nu_1 \sqsubseteq_{\mathcal{V}_{\text{lv}}^\#} \nu_2$. Intuitively, ν_1 represents vectors of length 2 or 3 with elements 1 and 2 in the first two positions, while ν_2 represents vectors of length 2 to 4 with elements 1, 2, and possibly 3. Every vector in ν_1 is also in ν_2 .

3.6 Join and Meet

The join operator combines two abstract elements, producing the least upper bound. The meet operator computes the greatest lower bound. Both must handle the case where the known content sequences have different lengths.

Definition 3.6.1

Join

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ with $k_1 = |\mathbf{p}_1|$ and $k_2 = |\mathbf{p}_2|$. The join operator is defined as:

$$\nu_1 \sqcup_{\mathcal{V}_B^\#} \nu_2 \stackrel{\text{def}}{=} (L_1 \sqcup_{\text{lv}_0} L_2, \mathbf{p}_1 \sqcup_{B^\#} \mathbf{p}_2, s_1 \sqcup_{B^\#} s_2, a_1 \sqcup_{\text{Attr}^\#} a_2)$$

where $\mathbf{p}_1 \sqcup_{B^\#} \mathbf{p}_2 \stackrel{\text{def}}{=} \langle p_{1,i} \sqcup_{B^\#} p_{2,i} \rangle_{i=1}^m \oplus \text{tail}$ and:

$$\text{tail} = \begin{cases} \langle p_{1,m+1} \sqcup_{B^\#} s_2 \cdots p_{1,k_1} \sqcup_{B^\#} s_2 \rangle & \text{if } k_1 > k_2 \\ \langle p_{2,m+1} \sqcup_{B^\#} s_1 \cdots p_{2,k_2} \sqcup_{B^\#} s_1 \rangle & \text{if } k_2 > k_1 \\ \varepsilon & \text{otherwise} \end{cases}$$

with $m = \min(k_1, k_2)$.

The $\oplus$ operator represents pure sequence concatenation. The attribute join is $a_1 \sqcup_{\text{Attr}^\#} a_2 = (\text{MustHave}_1 \cap \text{MustHave}_2, \text{MayHave}_1 \cup \text{MayHave}_2)$ when $a_j = (\text{MustHave}_j, \text{MayHave}_j)$, and absorbs $\perp_{\text{Attr}^\#}$.

The join handles different-length known content by extending the shorter sequence with the other element's summary.

Definition 3.6.2

Meet

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ with $L_1 = [l_1, u_1]$, $L_2 = [l_2, u_2]$, $k_1 = |\mathbf{p}_1|$, $k_2 = |\mathbf{p}_2|$, and $M = \max(k_1, k_2)$. The meet operator is defined as:

$$\nu_1 \sqcap_{\mathcal{V}_B^\#} \nu_2 \stackrel{\text{def}}{=} \begin{cases} \perp_{\mathcal{V}_B^\#} & \text{if } \max(l_1, l_2) > u' \\ \xi_1 & \text{if } \begin{aligned} &u_1 = u_2 = +\infty \\ &\wedge s_1 \sqcap_{B^\#} s_2 \neq \perp_{B^\#} \\ &\wedge u' = M \end{aligned} \\ \xi_2 & \text{otherwise} \end{cases}$$

where:

$$\begin{aligned}\xi_1 &\stackrel{\text{def}}{=} ([\max(l_1, l_2), +\infty], \mathbf{p}', s_1 \sqcap_{\mathcal{B}^\#} s_2, a_1 \sqcap_{\text{Attr}^\#} a_2), \\ \xi_2 &\stackrel{\text{def}}{=} ([\max(l_1, l_2), u'], \mathbf{p}', \perp_{\mathcal{B}^\#}, a_1 \sqcap_{\text{Attr}^\#} a_2).\end{aligned}$$

The resulting known content is $\mathbf{p}' = \langle p'_i \rangle_{i=1}^{u'}$ built with elements $p'_i = p'_{1,i} \sqcap_{\mathcal{B}^\#} p'_{2,i}$, and:

$$p'_{1,i} = \begin{cases} p_{1,i} & \text{if } i \leq k_1 \\ s_1 & \text{if } i > k_1 \end{cases} \quad p'_{2,i} = \begin{cases} p_{2,i} & \text{if } i \leq k_2 \\ s_2 & \text{if } i > k_2 \end{cases}$$

with $u' = \max\{i \in [0, \max(l_1, l_2, M)] \mid \forall j \in [1, i] : p'_j \neq \perp_{\mathcal{B}^\#}\}$.

The attribute meet is defined as

$$a_1 \sqcap_{\text{Attr}^\#} a_2 = (\text{MustHave}_1 \cup \text{MustHave}_2, \text{MayHave}_1 \cap \text{MayHave}_2)$$

when $\text{MustHave}_1 \cup \text{MustHave}_2 \subseteq \text{MayHave}_1 \cap \text{MayHave}_2$, otherwise is $\perp_{\text{Attr}^\#}$.

The meet truncates the known content at the first position where the meet of the corresponding abstract values is $\perp_{\mathcal{B}^\#}$, since no concrete vector can satisfy both constraints at that position.

Consider the following.

Example

Join and Precision Loss

Let $\nu_1 = ([2, 2], \langle [1, 1], [1, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $\nu_2 = ([1, +\infty], \langle [1, 1], [2, 2], [3, 3] \rangle, [-1, -1], \text{AttrEmpty})$. Their join is:

$$\nu_1 \sqcup_{\mathcal{V}_{\text{Itv}}^\#} \nu_2 = ([1, +\infty], \langle [1, 1], [2, 2], [3, 3] \rangle, [-1, -1], \text{AttrEmpty})$$

while the meet is:

$$\nu_1 \sqcap_{\text{Itv}} \nu_2 = ([2, 2], ([1, 1], [2, 2]), \perp_{\text{Itv}}, \text{AttrEmpty})$$

3.7 Widening and Termination

Widening is essential for ensuring termination of fixed-point iteration over the abstract domain.

Definition 3.7.1

Widening

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ with $k_1 = |\mathbf{p}_1|$, $k_2 = |\mathbf{p}_2|$, $m = \min(k_1, k_2)$, and $M = \max(k_1, k_2)$. We define:

$$\nu_1 \nabla_{\mathcal{V}_B^\#} \nu_2 \stackrel{\text{def}}{=} ([l', u'], \mathbf{p}', s', a_1 \sqcup_{\text{Attr}^\#} a_2)$$

where $[l', u'] = L_1 \nabla_{\text{Itv}_0} L_2$. The content extends to the maximum length when the widened length remains finite, or when both inputs are finite but their widening produces an infinite bound; otherwise,

when at least one input already has infinite length, it truncates to the common length:

$$p' = \begin{cases} \langle p'_1 \cdots p'_M \rangle & \text{if } u' \neq +\infty, \\ & \vee (u_1 \neq +\infty \wedge u_2 \neq +\infty) \\ \langle p_{1,1} \nabla_{\mathcal{B}^\#} p_{2,1} \cdots p_{1,m} \nabla_{\mathcal{B}^\#} p_{2,m} \rangle & \text{otherwise} \end{cases}$$

where $p'_j = p_{1,j} \nabla_{\mathcal{B}^\#} p_{2,j}$ for $j \leq m$, and $p'_j = p_{i,j}$ for elements from the longer vector i when $m < j \leq M$. The summary depends on whether the widened length is infinite:

$$s' = \begin{cases} \perp_{\mathcal{B}^\#} & \text{if } u' \neq +\infty \\ s_1 \nabla_{\mathcal{B}^\#} \left(s_2 \sqcup_{\mathcal{B}^\#} \bigsqcup_{j=m+1}^{k_1} p_{1,j} \sqcup_{\mathcal{B}^\#} \bigsqcup_{j=m+1}^{k_2} p_{2,j} \right) & \text{otherwise} \end{cases}$$

The attributes use the least upper bound, since $\text{Attr}^\#$ is finite and does not need widening.

A peculiarity of this widening is that it does not lose precision immediately by truncating to the common prefix $m = \min(k_1, k_2)$ of the two known contents. Instead, the first case of p' extends to the maximum length $M = \max(k_1, k_2)$ whenever the widened upper bound remains finite ($u' \neq +\infty$) or *both* input upper bounds are finite ($u_1 \neq +\infty$ and $u_2 \neq +\infty$).

Truncating always to the common prefix m would discard precision already in the very first widening step between two elements that still have finite length but whose known contents differ in size. The second disjunct preserves this information: as long as both input upper bounds are finite, the known content retains the maximum length M , even when the length widening already produced $+\infty$.

This is not a license for unbounded growth. Once the first widening step sets the upper bound to $+\infty$, the result's length component becomes $+\infty$. When this result is later used as the left operand ν_1 in a subsequent widening, $u_1 \neq +\infty$ is false. At that point both conditions of the disjunction fail, the first case of p' no longer applies, and the widening falls back to the common prefix m . The content thus stops growing after the very first step in which the length diverges. In short, the widening uses M while both operands are finite-bounded (even if the widened length already diverged), and truncates to m once the left operand itself carries infinite length.

These properties are formalized as follows.

Theorem 3.7.1

Well-Foundedness of Widening

The widening operator $\nabla_{\mathcal{V}^\#_{\mathcal{B}}}$ is well-founded:

1. For all $\nu_1, \nu_2 \in \mathcal{V}_B^\sharp$, $\nu_1 \sqsubseteq_{\mathcal{V}_B^\sharp} \nu_1 \nabla_{\mathcal{V}_B^\sharp} \nu_2$ and $\nu_2 \sqsubseteq_{\mathcal{V}_B^\sharp} \nu_1 \nabla_{\mathcal{V}_B^\sharp} \nu_2$.
2. For every sequence $(\nu_i)_{i \in \mathbb{N}}$ in $\mathcal{V}_B^\sharp$, the iterates $x_0 = \nu_0$ and $x_{i+1} = x_i \nabla_{\mathcal{V}_B^\sharp} \nu_{i+1}$ stabilize after finitely many steps.

Proof

The upper-bound property follows component-wise: interval widening for the length, value widening for the content and summary, and the finite lattice least upper bound for attributes. For the known content, a case analysis on the two shapes of p' verifies the appropriate clause of $K(\nu_1, \nu')$.

For convergence, the length intervals stabilize by the well-foundedness of interval widening. If they stabilize to a finite bound u^* , the known content length is bounded by u^* and each position evolves by value widening. If they stabilize to $+\infty$, the known content length becomes non-increasing and stabilizes; each position evolves by value widening. The summary then becomes a pure value-widening sequence and stabilizes. Attributes stabilize immediately because $\text{Attr}^\sharp$ is finite. The full proof is in Proof (of Theorem 3.7.1).

■ Proof,

99

□

The following example shows the widening in action.

Example

Widening Stabilizing a Chain

Consider the ascending chain:

$$\begin{aligned}
 \nu_0 &= ([0, 0], \varepsilon, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 \nu_1 &= ([1, 1], \langle [1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 \nu_2 &= ([2, 2], \langle [1, 1], [2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 \nu_3 &= ([3, 3], \langle [1, 1], [2, 2], [3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \\
 &\vdots
 \end{aligned}$$

derived from the following code:

```

x ← c()
i ← 1

while (TRUE) {
  x ← c(x, i)
  i ← i+1
}
```

Without widening, this chain never stabilizes. With widening:

- $x_0 = \nu_0$
- $x_1 = x_0 \nabla_{\mathcal{V}_{\text{Itv}}^\sharp} \nu_1 = ([0, +\infty], \langle [1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$
- $x_2 = x_1 \nabla_{\mathcal{V}_{\text{Itv}}^\sharp} \nu_2 = ([0, +\infty], \langle [1, 1], [2, 2] \rangle, \text{AttrEmpty})$
- $x_3 = x_2 \nabla_{\mathcal{V}_{\text{Itv}}^\sharp} \nu_3 = ([0, +\infty], \langle [1, 1], [2, +\infty] \rangle, \text{AttrEmpty})$
- $x_4 = x_3 \nabla_{\mathcal{V}_{\text{Itv}}^\sharp} \nu_4 = x_3$

3.8 Concretization and Galois Connection

The concretization function maps abstract elements back to sets of concrete vectors, establishing the formal link between the abstract and concrete domains.

Definition 3.8.1

Concretization

The concretization function $\gamma_{\mathcal{V}_B^\#} : \mathcal{V}_B^\# \rightarrow \mathcal{P}(\mathcal{V})$ is defined as:

$$\gamma_{\mathcal{V}_B^\#}((\llbracket l, u \rrbracket, \mathbf{p}, s, a)) \stackrel{\text{def}}{=} \left\{ [c_1, \dots, c_n]_{\mathcal{A}} \in \mathcal{V} \mid \begin{array}{l} n \in \gamma_{\text{Itv}_0}(\llbracket l, u \rrbracket), \mathcal{A} \in \gamma_{\text{Attr}^\#}(a), \\ \forall i \in [1, \min(n, |\mathbf{p}|)] : c_i \in \gamma_{\mathcal{B}^\#}(p_i), \\ \forall j \in [|\mathbf{p}| + 1, n] : c_j \in \gamma_{\mathcal{B}^\#}(s) \end{array} \right\}$$

The attribute concretization $\gamma_{\text{Attr}^\#} : \text{Attr}^\# \rightarrow \mathcal{P}(\mathcal{P}(\text{Attr}))$ maps abstract attribute elements to sets of concrete attribute sets:

$$\gamma_{\text{Attr}^\#}(a) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \emptyset & \text{if } a = \perp_{\text{Attr}^\#} \\ \left\{ \mathcal{A} \subseteq \text{Attr} \mid \begin{array}{l} \text{MustHave} \\ \subseteq \mathcal{A} \\ \subseteq \text{MayHave} \end{array} \right\} & \text{if } a = (\text{MustHave}, \text{MayHave}) \end{array} \right\}$$

We can now establish the meet-preservation property.

Theorem 3.8.1

Meet-Preservation

Given the meet-preserving $\gamma_{\mathcal{B}^\#}$, the concretization function $\gamma_{\mathcal{V}_B^\#}$ is meet-preserving, i.e., for all $\nu_1, \nu_2 \in \mathcal{V}_B^\#$:

$$\gamma_{\mathcal{V}_B^\#}(\nu_1 \sqcap_{\mathcal{V}_B^\#} \nu_2) = \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$$

Proof

If the meet is $\perp_{\mathcal{V}_B^\#}$, the intersection is empty. Otherwise, we use the meet-preservation of $\gamma_{\mathcal{B}^\#}$ and $\gamma_{\text{Attr}^\#}$ to show that the content and attribute constraints of the meet exactly capture the shared concrete vectors. The proof splits into finite and infinite meet cases, with the finite case relying on the fact that the length bounds exclude vectors beyond the conflicting position. The full proof is in Proof (of Theorem 3.8.1).

■ Proof,

97

□

Having established meet-preservation, the Galois connection follows.

Theorem 3.8.2**Galois Connection**

Given that the concretization function $\gamma_{\mathcal{V}_B^\#}$ is meet-preserving,

$$\langle \mathcal{P}(\mathcal{V}), \sqsubseteq \rangle \xLeftrightarrow[\alpha_{\mathcal{V}_B^\#}]{\gamma_{\mathcal{V}_B^\#}} \langle \mathcal{V}_B^\#, \sqsubseteq_{\mathcal{V}_B^\#} \rangle$$

where the abstraction function $\alpha_{\mathcal{V}_B^\#} : \mathcal{P}(\mathcal{V}) \rightarrow \mathcal{V}_B^\#$ is defined as:

$$\alpha_{\mathcal{V}_B^\#}(C) \stackrel{\text{def}}{=} \bigsqcap_{\mathcal{V}_B^\#} \left\{ \nu \in \mathcal{V}_B^\# \mid C \subseteq \gamma_{\mathcal{V}_B^\#}(\nu) \right\}$$

Proof**of Theorem 3.8.2**

Follows from Theorem 3.8.1 and the classical result of Cousot and Cousot [1]: a meet-preserving concretization function induces a Galois connection via the meet-of-upper-bounds construction.

[1] Cousot et al., 1977
Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,

□

For instance:

Example**Concretization of an Abstract Element**

Let $\nu = ([2, 3], \langle [0, 1], [2, 2], [3, 3] \rangle, \perp_{\text{ltv}}, \text{AttrEmpty})$. The concretization $\gamma_{\mathcal{V}_{\text{ltv}}^\#}(\nu)$ contains:

- Vectors of length 2: $\{[0, 2]_\emptyset, [1, 2]_\emptyset\}$
- Vectors of length 3: $\{[0, 2, 3]_\emptyset, [1, 2, 3]_\emptyset\}$

So $\gamma_{\mathcal{V}_{\text{ltv}}^\#}(\nu) = \{[0, 2]_\emptyset, [1, 2]_\emptyset, [0, 2, 3]_\emptyset, [1, 2, 3]_\emptyset\}$.

3.9 Chapter Summary

This chapter defined the parametric abstract domain $\mathcal{V}_B^\#$ for R vectors. The domain combines four components: a length interval, a finite known content sequence, a summary for unknown positions, and an attribute abstraction. We introduced an equivalence relation and quotient construction to obtain canonical minimum representatives, defined the partial order and proved it forms a complete lattice, gave the join and meet operators, introduced a well-founded widening operator, and established the Galois connection with the concrete collecting semantics.

The domain design achieves a balance between precision (explicitly tracking known positions) and tractability (summarizing unknown positions compactly). The parametric content domain allows instantiating the framework with different levels of value abstraction, from constant propagation to interval analysis.

Abstract Operators

Transferring concrete R vector operators into the abstract domain requires careful handling of the gaps between concrete collections and their over-approximations. This chapter lifts each operator from Chapter 2 to $\mathcal{V}_B^\sharp$, the abstract domain established in Chapter 3. Formal soundness proofs are provided for recycling and positive selection (Theorem 4.4.1, Appendix B); the remaining abstract operators are defined and their formal soundness proofs are left to future work.

The chapter is organized as follows. Section 4.1 discusses how type coercion is handled implicitly in the abstract domain. Section 4.2 lifts the concrete attribute operators `setAttr` and `getAttr` to `setAttr‡` and the abstract projection, and documents how attributes are threaded through the remaining operators. Section 4.3.2 presents the abstract concatenation operator $\oplus^\sharp$ and the helper functions `Squash` and `SquashExcept`. Section 4.4 covers the abstract recycling operator `Recycle‡` with its helpers $\rho_c^\sharp$ and $\rho_f^\sharp$, culminating in Theorem 4.4.1, the soundness theorem. Section 4.5 introduces shared auxiliary functions used by the abstract selection and update operators. Section 4.6 presents the abstract selection operator `Select‡`, covering positive, negative, and logical indexing. Section 4.7 covers the abstract update operator `Update‡` through the same three indexing modes. Throughout this chapter, we use the notation and domain definitions established in Chapter 3 and the concrete definitions from Chapter 2.

4.1 Abstract Coercion

R is dynamically typed but enforces strict **Type Homogeneity** within vectors. When an operation involves vectors of different types, elements are implicitly coerced to a common type before the operation proceeds.

Abstract type handling is structurally trivial. Since the type hierarchy $\mathcal{T}$ under the coercion order $\text{Bool} \prec \text{Int} \prec \text{Float}$ forms a finite lattice, the full typed abstract domain is the Cartesian product $\mathcal{T} \times \mathcal{V}_B^\sharp$. The type component is a finite chain of height 3: its join is the maximum along the chain, and its lattice properties are immediate. With regard to the value abstraction $\mathcal{V}_B^\sharp$, we rely on the hypothesis that the chosen instantiation of $\mathcal{B}^\sharp$ can abstract values of all three concrete types. In our integer interval instantiation this holds: Boolean values are encoded as singleton intervals (True maps to $[1, 1]$ and False to $[0, 0]$), while integer and floating-point values are represented by intervals with appropriate bounds.

1 Abstract Coercion,	55
2 Abstract Attribute Manipulation,	56
3 Helper Functions,	57
4 Abstract Recycling,	58
5 Helper Functions,	61
6 Abstract Selection,	65
7 Abstract Update,	71
8 Chapter Summary,	79

□ Theorem,	61
------------	----

⊗ Section 4.1,	55
----------------	----

⊗ Section 4.2,	56
----------------	----

⊗ Section 4.3.2,	57
------------------	----

⊗ Section 4.4,	58
----------------	----

□ Theorem,	61
------------	----

⊗ Section 4.5,	61
----------------	----

⊗ Section 4.6,	65
----------------	----

⊗ Section 4.7,	71
----------------	----

⊗ Chapter 3,	41
--------------	----

⊗ Chapter 2,	25
--------------	----

⊗ Type Homogeneity. The invariant that every element of an R vector must share the same atomic type (Bool, Int, or Float in this thesis). When operations combine values of different types, the abstract domain $\mathcal{V}_B^\sharp$ computes the least upper bound type and applies coercion.

4.2 Abstract Attribute Manipulation

The concrete operators `setAttr` and `getAttr` (Section 2.6) modify and inspect the attribute set of a concrete vector. In the abstract domain, the attribute component $a \in \text{Attr}^\sharp$ approximates the set of attributes via the must/may pair abstraction defined in Chapter 3. We now lift the attribute operators to $\mathcal{V}_B^\sharp$.

⇒ Section 2.6,

32

⇒ Chapter 3,

41

Definition 4.2.1

Abstract `setAttr` and `getAttr`

The abstract operator $\text{setAttr}^\sharp : \mathcal{V}_B^\sharp \times \mathcal{P}(\text{Attr}) \rightarrow \mathcal{V}_B^\sharp$ replaces the attribute abstraction of a vector:

$$\text{setAttr}^\sharp([l, u], p, s, a, A) \stackrel{\text{def}}{=} ([l, u], p, s, (A, A))$$

The resulting attribute abstraction (A, A) encodes that the vector definitely possesses exactly the attributes in A and definitely lacks all others: the concretization

$$\gamma_{\text{Attr}^\sharp}((A, A)) = \{A\}$$

contains only the concrete attribute set A .

The abstract $\text{getAttr}^\sharp : \mathcal{V}_B^\sharp \rightarrow \mathcal{V}_B^\sharp$ returns the abstract vector unchanged:

$$\text{getAttr}^\sharp(\nu) \stackrel{\text{def}}{=} \nu$$

Since the attribute component is the fourth element of ν , it can be extracted via projection.

The remaining abstract operators of this chapter thread attributes through as follows:

- Concatenation $\oplus^\sharp$ joins the attribute abstractions of both operands via $\sqcup_{\text{Attr}^\sharp}$ (Definition 4.3.3), mirroring the concrete semantics where $\oplus$ preserves only the “names” attribute from the union (Definition 2.7.1).
- Recycling $\text{Recycle}^\sharp$ preserves the attribute abstraction of each operand unchanged, since concrete recycling does not modify attributes.
- Selection $\text{Select}^\sharp$ inherits the attribute abstraction of the source vector, since the concrete $[]$ operator preserves all attributes of the original.
- Update $\text{Update}^\sharp$ inherits the attribute abstraction of the target vector, since the concrete update modifies only data elements, not attributes.

:= Definition,

57

:= Definition,

32

4.3 Helper Functions

4.3.1 Squash and SquashExcept

Before defining the abstract concatenation, we introduce two auxiliary functions that collapse vector values into a single abstract element.

Definition 4.3.1

Squash

The function $\text{Squash} : \mathcal{V}_B^\# \rightarrow \mathcal{B}^\#$ collapses all possible values of an abstract vector into a single abstract element:

$$\text{Squash}([l, u], \mathbf{p}, s, a) \stackrel{\text{def}}{=} \bigsqcup_{i=1}^{|\mathbf{p}|} \mathcal{B}_i^\# p_i \sqcup_{\mathcal{B}^\#} s$$

This captures all possible values that may appear in any position of the vector.

Definition 4.3.2

SquashExcept

The function $\text{SquashExcept} : \mathcal{V}_B^\# \times \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{B}^\#$ is similar but skips indices in a given set E :

$$\text{SquashExcept}([l, u], \mathbf{p}, s, a, E) \stackrel{\text{def}}{=} \bigsqcup_{i=1, i \notin E}^{|\mathbf{p}|} \mathcal{B}_i^\# p_i \sqcup_{\mathcal{B}^\#} s$$

Positions in E are excluded from the join.

4.3.2 Abstract Concatenation $\oplus^\#$

The abstract operator $\oplus^\# : \mathcal{V}_B^\# \times \mathcal{V}_B^\# \rightarrow \mathcal{V}_B^\#$ lifts $\oplus$ to the abstract domain. It must handle two qualitatively different cases depending on whether the left operand has a finite upper bound.

Definition 4.3.3

Abstract Concatenation

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ where $L_1 = [l_1, u_1]$ and $L_2 = [l_2, u_2]$. Let $k_1 = |\mathbf{p}_1|$ and $k_2 = |\mathbf{p}_2|$. The result $\nu_1 \oplus^\# \nu_2 = (L, \mathbf{p}, s, a)$ is defined as:

$$\nu_1 \oplus^\# \nu_2 \stackrel{\text{def}}{=} \begin{cases} ([l_1 + l_2, u_1 + u_2], \mathbf{p}', s', a_1 \sqcup_{\text{Attr}^\#} a_2) & \text{if } u_1 \neq +\infty \\ ([l_1 + l_2, +\infty], \langle p_{1,1} \dots p_{1,l_1} \rangle, s'', a_1 \sqcup_{\text{Attr}^\#} a_2) & \text{if } u_1 = +\infty \end{cases}$$

In the finite case, given $d = u_1 - l_1$, the known content is first initialized by pure sequence concatenation:

$$\mathbf{p}' = \mathbf{p}_1 \oplus \mathbf{p}_2$$

Then, for each position $i \in [u_1 + 1, u_1 + k_2]$ we propagate the value backward into the preceding d positions, modeling R's behavior of recycling during concatenation:

$$\forall j \in [1, d] : p'_{i-j} \leftarrow p'_{i-j} \sqcup_{\mathcal{B}^\#} p'_i$$

The summary in this case is $s' = s_2$.

In the infinite case, the result retains only the first l_1 positions of ν_1 (the guaranteed minimum), and the summary collapses all remaining information:

$$s'' = s_1 \sqcup_{\mathcal{B}^\#} \text{SquashExcept}(\nu_1, [1, l_1]) \sqcup_{\mathcal{B}^\#} \text{Squash}(\nu_2)$$

The backward propagation in the finite case is necessary when $l_1 \neq u_1$, because the concretizations of ν_1 can have different lengths. Without propagation, concatenating a concretization of ν_2 would yield results where the same position index may belong to ν_1 or to ν_2 , depending on the actual length of ν_1 .

To illustrate, consider the following.

Example

Abstract Concatenation with Propagation

Let $\nu_1 = ([1, 2], \langle [1, 1], [2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $\nu_2 = ([1, 1], \langle [9, 9] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$. The concretizations of ν_1 are $[1]$ and $[1, 2]$; the concretization of ν_2 is $[9]$. The possible concrete concatenations are therefore $[1, 9]$ and $[1, 2, 9]$.

Since $d = u_1 - l_1 = 1$, backward propagation applies:

$$\nu_1 \oplus^\# \nu_2 = ([2, 3], \langle [1, 1], [2, 9], [9, 9] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

The length is $[1 + 1, 2 + 1] = [2, 3]$. Propagation joins p'_3 into p'_2 , yielding $[2, 9]$, because position 2 may come from ν_1 (when $|\nu_1| = 2$) or from ν_2 via recycling (when $|\nu_1| = 1$), and the join captures both possibilities.

4.4 Abstract Recycling

4.4.1 Helpers

To lift recycling to the abstract domain, we first define two helper functions that generate the known content of an abstract vector after recycling to a target length.

Definition 4.4.1

Abstract Cycling $\rho_c^\#$

The function $\rho_c^\# : \mathcal{B}^{\#*} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{B}^{\#*}$ cycles the first $i' = \min(i, |p|)$ elements of its input to reach length h :

$$\rho_c^\sharp(\mathbf{p}, i, h) \stackrel{\text{def}}{=} \underbrace{p_1 \dots p_{i'} \oplus \dots \oplus p_1 \dots p_{i'} \oplus p_1 \dots p_r}_{\lfloor \frac{h}{i'} \rfloor \text{ times}}$$

where $r = h \bmod i'$. In positional terms:

$$\rho_c^\sharp(\mathbf{p}, i, h)_j = \mathbf{p}_{((j-1) \bmod i') + 1} \quad \text{for } j \in [1, h]$$

This is the direct abstract counterpart of concrete ρ_c : it reuses known content blocks cyclically.

Definition 4.4.2

Full Recycling $\rho_f^\sharp$

The function $\rho_f^\sharp : \mathcal{B}^{\sharp*} \times \mathcal{B}^\sharp \times \mathbb{N} \times \mathbb{N} \rightarrow \mathcal{B}^{\sharp*}$ generates known positions for each possible original length by joining $\rho_c^\sharp$ over all $i \in [l, n]$:

$$\rho_f^\sharp(\mathbf{p}, s, l, n) \stackrel{\text{def}}{=} \begin{cases} \bigsqcup_{i=l}^{\min(n, |\mathbf{p}|)} \rho_c^\sharp(\mathbf{p}, i, n) & \text{if } s = \perp_{\mathcal{B}^\sharp} \\ \bigsqcup_{i=l}^n \rho_c^\sharp(\mathbf{p}^i, i, n) & \text{if } s \neq \perp_{\mathcal{B}^\sharp} \end{cases}$$

When $s \neq \perp_{\mathcal{B}^\sharp}$, we pad $\mathbf{p}$ to length i using the summary:

$$\mathbf{p}_j^i = \begin{cases} p_j & \text{if } j \leq |\mathbf{p}| \\ s & \text{if } j > |\mathbf{p}| \end{cases}$$

The two cases of $\rho_f^\sharp$ handle finite and infinite vectors:

- When $s = \perp_{\mathcal{B}^\sharp}$ (finite vector), the original length i ranges only up to $\min(n, |\mathbf{p}|) = \min(n, u)$ since all positions are known.
- When $s \neq \perp_{\mathcal{B}^\sharp}$ (infinite vector), we must pad with s to represent potentially longer original lengths, so i ranges up to n .

For instance:

Example

Abstract Recycling Helpers

Let $\mathbf{p} = \langle [1, 1][2, 2] \rangle$. Then:

$$\begin{aligned} \rho_f^\sharp(\mathbf{p}, \perp_{\text{tv}}, 1, 3) &= \rho_c^\sharp(\mathbf{p}, 1, 3) \sqcup_{\mathcal{B}^{\sharp*}} \rho_c^\sharp(\mathbf{p}, 2, 3) \\ &= \langle [1, 1][1, 1][1, 1] \rangle \sqcup_{\mathcal{B}^{\sharp*}} \langle [1, 1][2, 2][1, 1] \rangle = \langle [1, 1][1, 2][1, 1] \rangle \end{aligned}$$

This captures all possible recycled contents when the original vector could have length 1 or 2 and is recycled to length 3. The join over i ensures soundness for all possible concrete original lengths.

4.4.2 Abstract Recycling Recycle[#]**Definition 4.4.3****Abstract Recycling**

Let $\nu_1 = ([l_1, u_1], \mathbf{p}_1, s_1, a_1)$ with $|\mathbf{p}_1| = m$ and $\nu_2 = ([l_2, u_2], \mathbf{p}_2, s_2, a_2)$ with $|\mathbf{p}_2| = n$. The abstract recycling function $\text{Recycle}^\# : \mathcal{V}_B^\# \times \mathcal{V}_B^\# \rightarrow \mathcal{V}_B^\# \times \mathcal{V}_B^\#$ returns the pair:

$$\text{Recycle}^\#(\nu_1, \nu_2) = (\nu'_1, \nu'_2)$$

where the target length is $[l', u'] = [\max(l_1, l_2), \max(u_1, u_2)]$ and:

$$\nu'_1 = \left([l', u'], \bigsqcup_{d=l'}^{\max(m,n)} \mathcal{B}^\# \rho_f^\#(\mathbf{p}_1, s_1, l_1, d), s'_1, a_1 \right)$$

$$\nu'_2 = \left([l', u'], \bigsqcup_{d=l'}^{\max(m,n)} \mathcal{B}^\# \rho_f^\#(\mathbf{p}_2, s_2, l_2, d), s'_2, a_2 \right)$$

The summary updates depend on whether the target length is infinite:

$$s'_1 = \begin{cases} s_1 & \text{if } u' \neq +\infty \\ s_1 \sqcup_{\mathcal{B}^\#} \bigsqcup_{k=1}^m \mathcal{B}^\# \mathbf{p}_{1,k} & \text{otherwise} \end{cases}$$

The known content for each vector is obtained by joining $\rho_f^\#$ over all possible target lengths $d \in [l', \max(m, n)]$. The summary is only valorized when the target upper bound is infinite ($u' = +\infty$), in which case it joins the original summary with all known content positions to cover positions beyond the known content after recycling.

The following example illustrates the construction.

Example**Abstract Recycling**

Let $\nu_1 = ([1, 2], \langle [1, 1], [2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $\nu_2 = ([1, 3], \langle [3, 3], [4, 4], [5, 5] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$. Both are finite, so summaries are $\perp_{\text{Itv}}$. The target length is $[1, 3]$ with $\max(m, n) = \max(2, 3) = 3$.

The known content of ν'_1 joins $\rho_f^\#$ over $d \in [1, 3]$:

$$\mathbf{p}'_1 = \bigsqcup_{d=1}^3 \mathcal{B}^\# \rho_f^\#(\langle [1, 1], [2, 2] \rangle, \perp_{\text{Itv}}, 1, d)$$

This yields an abstract vector ν'_1 whose known content over-approximates all possible recyclings of ν_1 to lengths 1, 2, or 3. Since the original length of ν_1 may be 1 or 2, and the target may be 1, 2, or 3 (dictated by ν_2), the join captures every combination. The summary $s'_1 = \perp_{\text{Itv}}$ since $u' = 3 \neq +\infty$.

4.4.3 Soundness of Abstract Recycling

We now establish that $\text{Recycle}^\sharp$ soundly abstracts Recycle . The proof is structured as a chain of four lemmas that progressively establish coverage of the recycled vector's length, known content, summary, and attributes. The complete proof is in Section B.1.

↻ Section B.1,

102

Theorem 4.4.1

Soundness of Abstract Recycling

The abstract recycling operator $\text{Recycle}^\sharp$ is a sound abstraction of the concrete recycling operator Recycle : for all $\nu_1, \nu_2 \in \mathcal{V}_B^\sharp$ with $\text{Recycle}^\sharp(\nu_1, \nu_2) = (\nu'_1, \nu'_2)$:

$$\forall x \in \gamma_{\mathcal{V}_B^\sharp}(\nu_1), \forall y \in \gamma_{\mathcal{V}_B^\sharp}(\nu_2) : \text{Recycle}(x, y) \in \gamma_{\mathcal{V}_B^\sharp}(\nu'_1) \times \gamma_{\mathcal{V}_B^\sharp}(\nu'_2)$$

Proof

By symmetry it suffices to show $x_r = \rho_c(x, d) \in \gamma_{\mathcal{V}_B^\sharp}(\nu'_1)$ where $d = \max(|x|, |y|)$. Length soundness follows from interval arithmetic. Known-content soundness uses the padded-covers lemma, cycling soundness, and join soundness to handle the joins over all possible target lengths: the finite case ($s_1 = \perp_{B^\sharp}$) is direct, and the infinite case ($s_1 \neq \perp_{B^\sharp}$) uses padded sequences. Summary soundness is justified by the summary soundness lemma. Attributes are preserved unchanged. The full proof with all four lemmas is in Section B.1.

↻ Section B.1,

102

□

4.5 Helper Functions

Before presenting the abstract selection and update operators, we introduce several helper functions that capture common patterns. The functions *Squash* and *SquashExcept* were already defined in Definition 4.3.1 and Definition 4.3.2; this section adds abstract cardinality, enumerability testing, zero propagation, selector compression, and abstract element access.

4.5.1 Abstract Cardinality and Enumerability

Definition 4.5.1

Abstract Cardinality

We define $\text{card} : \text{Itv} \rightarrow \mathbb{N} \cup \{+\infty\}$, a cardinality function over integer intervals:

$$\text{card}(c) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } c = \perp_{\text{Itv}} \\ |b - a| + 1 & \text{if } c = [a, b] \text{ with } a \\ & b \in \mathbb{Z} \\ +\infty & \text{otherwise} \end{cases}$$

Definition 4.5.2**Enumerable Positions**

An abstract value $c \in \text{Itv}$ is **enumerable** when its cardinality does not exceed a fixed threshold $\theta \in \mathbb{N}$:

$$\text{Enumerable}(c) \stackrel{\text{def}}{=} \text{card}(c) \leq \theta$$

The threshold θ determines whether an abstract index interval can be individually tracked during selection and update operations.

4.5.2 Zero Propagation: Propagate

The function `Propagate` is a recursive helper that propagates abstract values forward through positions containing zero. It is used by `AdjustForZeros` to determine, for each position in a selector vector, which possible abstract value emerges after removing zero-valued positions.

We distinguish between **definite zeros** (where $\gamma_{\text{Itv}}(c_i) = \{0\}$) and **possible zeros** (where $0 \in \gamma_{\text{Itv}}(c_i) \wedge \gamma_{\text{Itv}}(c_i) \neq \{0\}$), using separate counters d (definite) and p (possible).

Definition 4.5.3**Propagate**

The function $\text{Propagate} : \text{Itv}^* \times \text{Itv} \times \mathbb{N} \times \mathbb{N} \rightarrow \text{Itv}$ takes a suffix of known selector content $\langle c_1, \dots, c_n \rangle$, a summary value s , a count d of definite pending zeros, and a count p of possible pending zeros. It is defined recursively:

$$\text{Propagate}(\varepsilon, s, d, p) \stackrel{\text{def}}{=} s$$

$$\text{Propagate}(\langle c_1, \dots, c_n \rangle, s, d, p) \stackrel{\text{def}}{=} \begin{cases} \text{Propagate}(\langle c_2, \dots, c_n \rangle, s, d, p) & \text{if } \gamma_{\text{Itv}}(c_1) = \{0\} \\ \text{Propagate}(\langle c_2, \dots, c_n \rangle, s, d, p) \sqcup_{\text{Itv}} \text{Propagate}(\langle c_2, \dots, c_n \rangle, s, d-1, p) & \text{if } d > 0 \wedge 0 \in \gamma_{\text{Itv}}(c_1) \\ \text{Propagate}(\langle c_2, \dots, c_n \rangle, s, d-1, p) & \text{if } d > 0 \\ c_1 \sqcup_{\text{Itv}} \text{Propagate}(\langle c_2, \dots, c_n \rangle, s, 0, p) & \text{if } 0 \in \gamma_{\text{Itv}}(c_1) \wedge d = 0 \\ c_1 \sqcup_{\text{Itv}} \text{Propagate}(\langle c_2, \dots, c_n \rangle, s, 0, p-1) & \text{if } 0 \notin \gamma_{\text{Itv}}(c_1) \wedge d = 0 \wedge p > 0 \\ c_1 & \text{if } \gamma_{\text{Itv}}(c_1) \neq \{0\} \wedge d = 0 \wedge p = 0 \end{cases}$$

The six cases capture distinct scenarios where zeros alter the effective position of non-zero selectors. A definite zero (case 1) contributes nothing and is simply skipped. When pending definite zeros exist and the current element may be zero (case 2), both possibilities are joined: the element is either skipped (if it is zero) or absorbs one pending zero (if it is not). A non-zero with pending definite zeros

(case 3) absorbs one and recurses. A possible zero with no pending definite zeros (case 4) joins the element itself with the result of skipping it. A non-zero with pending possible zeros (case 5) joins the element with the result that consumes one possible zero. A clean non-zero (case 6) returns the element directly.

A concrete example clarifies the case analysis.

Example

Propagate

Let the known content be $\langle [0, 0], [0, 2], [3, 3] \rangle$ and $s = \perp_{\text{Itv}}$. Calling $\text{Propagate}(\langle [0, 0], [0, 2], [3, 3] \rangle, \perp_{\text{Itv}}, 0, 0)$:

- $[0, 0]$ is a definite zero (case 1), skipped.
- $[0, 2]$ may be zero, and $d = 0$ (case 4): join $[0, 2]$ with the recursive result from $[3, 3]$.
- $[3, 3]$ is a clean non-zero (case 6), returning $[3, 3]$.

The join $[0, 2] \sqcup_{\text{Itv}} [3, 3] = [0, 3]$ captures both possibilities: the first position of the selector could be 0, 1, or 2, i.e., the second position is the first non-zero, or 3, i.e., the second position is zero and the third is reached.

4.5.3 Selector Compression: AdjustForZeros

The function `AdjustForZeros` compresses a selector vector by removing definite zero-valued positions and adjusting the length bounds accordingly. In the concrete semantics, zero indices are silently discarded during selection (Definition 2.9.1) and update (Definition 2.10.1); `AdjustForZeros` performs this compression in the abstract, revealing non-zeros positions of the selector.

:= Definition,

34

:= Definition,

37

For the same reason specified in Section 2.9, we designed these operators to evaluate abstract interval vectors.

⊗ Section 2.9,

34

Definition 4.5.4

AdjustForZeros

The function `AdjustForZeros` maps an abstract vector over the interval domain to a compressed abstract vector:

$$\text{AdjustForZeros}([l, u], \mathbf{p}, s, a) \stackrel{\text{def}}{=} \left([l', u'], \bigoplus_{i=1}^n c_i, s, a \right),$$

where for each position $i \in [1, |\mathbf{p}|]$, we compute:

$$c_i = \text{Propagate}(\langle p_i \cdots p_n \rangle, s, \hat{d}_i, \hat{p}_i), \quad \text{provided } c_i \neq \perp_{\text{Itv}},$$

with $n = |\mathbf{p}|$ and the counters:

- $\hat{d}_i = |\{j < i \mid \gamma_{\text{Itv}}(p_j) = \{0\}\}|$ (definite zeros before i)
- $\hat{p}_i = |\{j < i \mid 0 \in \gamma_{\text{Itv}}(p_j) \wedge \gamma_{\text{Itv}}(p_j) \neq \{0\}\}|$ (possible zeros before i)

The adjusted known content collects only the non- $\perp_{\text{Itv}}$ results. The length bounds are:

$$l' = \max(0, l - \hat{p}_{n+1} - \hat{d}_{n+1}), \quad u' = u - \hat{d}_{n+1},$$

where $\hat{d}_{n+1}$ and $\hat{p}_{n+1}$ are the total counts over the entire known content.

The l adjustment reflects two observations. Definite zeros are guaranteed to be removed, so the upper bound shrinks by their count. Possible zeros may or may not be removed, so the lower bound is penalized by both definite and possible zero counts, covering the worst case where all possible zeros are indeed zero.

As an illustration:

Example

AdjustForZeros

Let $\nu = ([2, 4], \langle [0, 0], [1, 1], [0, 0], [5, 5] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$. The known content has two definite zeros (positions 1 and 3) and no possible zeros, so $\hat{d}_5 = 2, \hat{p}_5 = 0$.

Compressing with Propagate gives the adjusted content $\langle [1, 1], [5, 5] \rangle$: the definite zeros are removed and the remaining non-zero values shift forward. The length becomes $[2 - 2, 4 - 2] = [0, 2]$, representing a selector of 0 to 2 non-zero indices.

With a possible zero at a non-first position, e.g. $\nu = ([1, 3], \langle [0, 0], [0, 2], [3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$, the counters are $\hat{d}_4 = 1, \hat{p}_4 = 1$. Compressing gives the adjusted content $\langle [0, 3], [3, 3] \rangle$: the possible zero introduces non-determinism, i.e., $[0, 2] \sqcup_{\text{Itv}} [3, 3] = [0, 3]$, while the definite zero is consumed by the later non-zero. The length becomes $[\max(0, 1 - (1 + 1)), 3 - 1] = [0, 2]$.

4.5.4 Abstract Element Access

The final helper retrieves the abstract value at a given position from an abstract vector, soundly covering both in-bounds and out-of-bounds access.

Definition 4.5.5

Abstract Element Access

Let $\nu = ([l_\nu, u_\nu], \mathbf{p}_\nu, s_\nu, a_\nu) \in \mathcal{V}_{\mathcal{B}}^\#$. The function $\nu^{*\#}(k)$ retrieves the abstract value at position k :

$$\nu^{*\#}(k) \stackrel{\text{def}}{=} \begin{cases} p_{\nu, k} & \text{if } 1 \leq k \leq l_\nu \\ p_{\nu, k} \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{if } l_\nu < k \leq |\mathbf{p}_\nu| \\ s_\nu \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{if } |\mathbf{p}_\nu| < k \leq u_\nu \\ \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{otherwise} \end{cases}$$

The four cases reflect the certainty about whether position k is within bounds. When $k \leq l_\nu$, all concrete vectors have at least k positions, so the position value is returned directly. When $l_\nu < k \leq |p_\nu|$, the position is within the known content but some concrete vectors may be shorter, so we join with $\alpha_{\mathcal{B}^\#}(\text{NA})$ to cover out-of-bounds access. When beyond the known content, the summary covers the tail. For definitely out-of-bounds positions, a single $\alpha_{\mathcal{B}^\#}(\text{NA})$ is returned.

Remark

The expression $p_{\nu,k} \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA})$ implicitly assumes that $\mathcal{B}^\#$ can represent both ordinary abstract values and the abstraction of NA as distinct elements, so that the least upper bound does not collapse all precision. For instance, instantiating $\mathcal{B}^\#$ with a plain interval domain would force $\alpha_{\mathcal{B}^\#}(\text{NA})$ to equal $\top_{\mathcal{B}^\#}$, and joining any interval with $\top_{\mathcal{B}^\#}$ would lose all positional information. This issue is easily resolved using the Cartesian product $\mathcal{B}^\# \times \{0, 1\}^\perp$ as the generic value domain, where the second component tracks whether the abstraction of NA is absent (0) or present (1), and $\perp, \top$ are the standard extremal elements. The join then operates component wise, preserving the numeric bound in the first component while marking the result as possibly containing NA in the second. We gloss over this construction in what follows, treating $\sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA})$ as a well-defined operation on a suitably instantiated domain.

4.6 Abstract Selection

The polymorphic behavior of the Select operator (Section 2.9) is abstracted by the function:

$$\text{Select}^\# : \mathcal{V}_{\mathcal{B}}^\# \times \mathcal{V}_{\text{ltv}}^\# \rightarrow \mathcal{V}_{\mathcal{B}}^\#.$$

Given the source vector $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$ and the vector of positions to extract $\nu_2 = ([l_2, u_2], p_2, s_2, a_2) \in \mathcal{V}_{\text{ltv}}^\#$, the resulting selection $\nu_r = ([l_r, u_r], p_r, s_r, a_r)$ is obtained through case analysis.

Since the concrete semantics distinguishes three mutually exclusive cases (positive indexing, negative indexing, and logical filtering) and R does not permit mixed positive/negative integer indices, the abstract operator handles these cases separately. For integer selection, we apply **Abstract Filtering** [1]: for each abstract element $p_{2,i} \in p_2$, we retain positions whose concretization falls within the respective sign class:

$$\begin{aligned} p_2^+ &= \langle p \in p_2 \mid \gamma_{\text{ltv}}(p) \subseteq \mathbb{Z}_{\geq 0} \cup \{\text{NA}\} \rangle \\ p_2^- &= \langle p \in p_2 \mid \gamma_{\text{ltv}}(p) \subseteq \mathbb{Z}_{\leq 0} \rangle. \end{aligned}$$

Let $\nu_2^+ = ([l_2, u_2], p_2^+, s_2, a_2)$ and $\nu_2^- = ([l_2, u_2], p_2^-, s_2, a_2)$. Then:

$$\text{Select}^\#(\nu_1, \nu_2) = \text{Select}_{\text{pos}}^\#(\nu_1, \nu_2^+) \sqcup_{\mathcal{V}_{\mathcal{B}}^\#} \text{Select}_{\text{neg}}^\#(\nu_1, \nu_2^-).$$

↻ Section 2.9,

34

▼ **Abstract Filtering.** An abstract interpretation technique that partitions abstract elements by checking whether their concretization satisfies a given condition. In this thesis, it is applied to separate abstract selector vectors into non-positive and non-negative index classes. Elements whose concretization is empty indicate an unreachable state and force the entire selection to $\perp_{\mathcal{V}_{\mathcal{B}}^\#}$.

[1] Cousot et al., 1977
Abstract Interpretation: A
Unified Lattice Model for Static
Analysis of Programs by
Construction or
Approximation of Fixpoints,

If any element of p_2^+ or p_2^- equals $\perp_{\text{Itv}}$, the corresponding selection returns $\perp_{\mathcal{V}_B^\#}$, since $\gamma_{\text{Itv}}(\perp_{\text{Itv}}) = \emptyset$ invalidates the sequence. The empty selector is a special case: $\text{Select}^\#(\nu_1, \alpha_{\mathcal{V}_B^\#}([\])) = \nu_1$, i.e., the source vector is returned unchanged.

4.6.1 Positive Selection

For positive integer selection, we first apply `AdjustForZeros` to handle zero indices in the selector, then proceed based on whether the resulting selector is finite or infinite.

Given $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$ and $\nu_2^+ = ([l_2, u_2], p_2^+, s_2, a_2)$, we compute the compressed selector:

$$([l'_2, u'_2], p'_2, s_2, a_2) = \text{AdjustForZeros}([l_2, u_2], p_2^+, s_2, a_2).$$

If $u'_2 = 0$, the selector contains only zeros and we soundly return $([0, 0], \varepsilon, \perp_{\mathcal{B}^\#}, \text{AttrEmpty})$.

Definition 4.6.1

Abstract Positive Selection, Finite Selector

When $u'_2 \neq +\infty$, the selector has finite length. We enumerate each position in p'_2 that is enumerable, extract the corresponding value from ν_1 , and set the summary to $\perp_{\mathcal{B}^\#}$:

$$\text{Select}_{\text{pos}}^\#(\nu_1, \nu_2^+) = ([l'_2, u'_2], p_r, \perp_{\mathcal{B}^\#}, a_1),$$

where the result content is built position-by-position:

$$p_r = \bigoplus_{i=1}^{|p'_2|} \begin{cases} \bigsqcup_j \nu_1^{*\#}(j) & \text{if Enumerable}(p_{2,i}^+) \\ \text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{otherwise} \end{cases},$$

with $j \in \gamma_{\text{Itv}}(p_{2,i}^+)$, $j \neq 0$.

When a selector position is enumerable, the join over the concretization of $p_{2,i}^+$ extracts precisely those source values that may be selected. When non-enumerable, the position covers too many indices to track individually and the entire source vector is collapsed via `Squash`, with an additional join of $\alpha_{\mathcal{B}^\#}(\text{NA})$ to handle out-of-bounds access.

Definition 4.6.2

Abstract Positive Selection, Infinite Selector

When $u'_2 = +\infty$, the selector has infinite length. The construction mirrors the finite case but uses a non-trivial summary:

$$\text{Select}_{\text{pos}}^\#(\nu_1, \nu_2^+) = ([l'_2, +\infty], p_r, s_r, a_1),$$

where p_r is built as in the finite case. The summary is:

$$s_r = \begin{cases} \text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{if } u_r > l_1 \\ \text{Squash}(\nu_1) & \text{otherwise} \end{cases}$$

where $\text{Squash}(\nu_2^+) = [l, u_r]$.

We illustrate with a simple case.

Example

Positive Selection

Let $\nu_1 = ([2, 2], \langle [10, 10], [20, 20] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $\nu_2^+ = ([1, 1], \langle [1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$. After `AdjustForZeros`, the compressed selector is unchanged (no zeros). The single enumerable position $[1, 1]$ extracts $\nu_1^{\#}(1) = [10, 10]$, yielding $\nu_r = ([1, 1], \langle [10, 10] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$.

If the selector contained $\langle [0, 0], [1, 1] \rangle$, `AdjustForZeros` would remove the zero and compress to $\langle [1, 1] \rangle$ with adjusted length $[0, 1]$.

Soundness is formalized as follows.

Theorem 4.6.1

Soundness of Positive Selection

The abstract positive selection $\text{Select}_{\text{pos}}^\#$ is a sound abstraction of the concrete positive selection: for all $\nu_1 \in \mathcal{V}_{\mathcal{B}^\#}^\#$, $\nu_2 \in \mathcal{V}_{\text{Itv}}^\#$, letting ν_2^+ be the positive-filtered selector and $\nu_r = \text{Select}_{\text{pos}}^\#(\nu_1, \nu_2^+)$:

$$\forall v \in \gamma_{\mathcal{B}^\#}(\nu_1), \forall c \in \gamma_{\text{Itv}}(\nu_2^+) : \text{Select}(v, c) \in \gamma_{\mathcal{B}^\#}(\nu_r).$$

Proof

The proof relies on three components: the soundness of `AdjustForZeros` (which compresses the selector by removing zero indices), the soundness of $\nu^{\#}(j)$ for abstract element access (distinguishing in-bounds, summary, and out-of-bounds cases), and join soundness for the join over all possible concrete indices at each enumerable selector position. Non-enumerable positions are over-approximated by $\text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA})$. The summary is $\perp_{\mathcal{B}^\#}$ for finite selectors and $\text{Squash}(\nu_1)$ (possibly extended with $\alpha_{\mathcal{B}^\#}(\text{NA})$) for infinite selectors. The complete proof is in Proof (of Theorem 4.6.1).

■ Proof,

108

□

4.6.2 Negative Selection

For integer negative selection, we apply `AdjustForZeros` to handle zero indices. Given $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$ and $\nu_2^- = ([l_2, u_2], p_2^-, s_2, a_2)$, we first compute:

$$\nu_2'^- = ([l_2', u_2'], p_2', s_2, a_2) = \text{AdjustForZeros}([l_2, u_2], p_2^-, s_2, a_2).$$

If $u'_2 = 0$, the selector contains only zeros and we return $([0, 0], \varepsilon, \perp_{B^\sharp}, \text{AttrEmpty})$.

We identify three sets of positions relative to ν_1 :

- **MustDeleted**: positions definitely excluded. Each position i satisfies $\text{card}(p_{2,i}^-) = 1$ and the concretization intersects $[-u_1, -1]$, meaning the negative index precisely targets a single valid position.
- **MayDeleted**: positions possibly excluded, including those with $\text{card}(p_{2,i}^-) \geq 1$ from the content whose concretization intersects $[-u_1, -1]$, plus all positions in $\gamma_{\text{ltv}}(s_2)$. If the summary is not enumerable, we conservatively assume all other positions may be deleted.
- **MustNotDeleted**: positions definitely kept, computed as $[1, |p_1|] \setminus \text{MayDeleted}$.

By construction $\text{MustDeleted} \subseteq \text{MayDeleted}$. For any set of positions S , we write S_i to denote $\{u \in S \mid u \leq i\}$.

We distinguish three cases based on the properties of the adjusted selector.

Definition 4.6.3

Abstract Negative Selection, Non-Enumerable Position

When at least one position in p'_2 is not enumerable, we cannot precisely track which positions are excluded. We conservatively assume all positions not in **MustDeleted** may be deleted:

$$\text{Select}_{\text{neg}}^\sharp(\nu_1, \nu_2^-) = ([0, u_r], p_r, s_r, a_1)$$

where $u_r = u_1 - |\text{MustDeleted}|$ and the summary is:

$$s_r = \begin{cases} \perp_{B^\sharp} & \text{if } u_1 \neq +\infty \\ s_1 & \text{otherwise} \end{cases}$$

The content is:

$$p_r = [\text{SquashExcept}(\nu_1, \text{MustDeleted})]_1^{\min(k_1, u_r)},$$

a vector of length $\min(|p_1|, u_r)$ where all positions carry the squashed value of the source, excluding only the definitely deleted positions.

Definition 4.6.4

Abstract Negative Selection, Enumerable, Infinite Selector

When $u'_2 = +\infty$ and all positions in p'_2 are enumerable, the result is $\nu_r = ([l_r, u_r], p_r, s_1, a_1)$, where:

$$l_r = l_1 - |\text{MayDeleted}_{l_1}|, \quad u_r = u_1 - |\text{MustDeleted}_{u_1}|$$

The content p_r has $|p_1| - |\text{MustDeleted}|$ positions, initialized to $\perp_{\mathcal{B}^i}$. We define $\text{CountMustDeleted}(i) = |\{d \in \text{MustDeleted} \mid d < i\}|$. The content is constructed in two phases:

1. For each $i \in \text{MustNotDeleted}$ (ascending):

$$\forall j \in [1, i - \text{CountMustDeleted}(i)] : p_{r,j} \leftarrow p_{r,j} \sqcup_{\mathcal{B}^i} p_{1,i}$$

2. For each $i \in \text{MayDeleted}$ (ascending):

$$\forall j \in [1, i - \text{CountMustDeleted}(i)] : p_{r,j} \leftarrow p_{r,j} \sqcup_{\mathcal{B}^i} p_{1,i}$$

The first phase places definitely kept elements compactly, at all positions they could occupy depending on how many preceding may-deleted elements are actually deleted. The second phase joins in the values of possibly-deleted elements at all positions they might shift into.

The finite case ($u'_2 \neq +\infty$ with all enumerable positions) follows the same construction but restricts MayDeleted to only the non-singleton abstract values from the content (the summary contributes no positions since the selector is finite), giving a finite upper bound.

For instance:

Example

Negative Selection

Let $\nu_1 = ([5, 5], \langle [1, 1], [2, 2], [3, 3], [4, 4], [5, 5] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$ and $\nu_2^- = ([2, 2], \langle [-2, -2], [-4, -5] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$.

After `AdjustForZeros`, we have the compressed selector. Position $[-2, -2]$ has cardinality 1 and concretization $\{-2\}$ which intersects $[-5, -1]$, so $\text{MustDeleted} = \{2\}$. Position $[-4, -5]$ has cardinality 2 and concretization $\{-4, -5\}$ intersecting $[-5, -1]$, so $\text{MayDeleted} = \{2, 4, 5\}$. The definitely kept positions are $\text{MustNotDeleted} = \{1, 3\}$.

Constructing p_r (initialized to length $5 - 1 = 4$ with $\perp_{\text{Itv}}$):

- Process $i = 1$: place $[1, 1]$ at $j \in [1, 1 - 0]$, so $p_{r,1} \leftarrow [1, 1]$.
- Process $i = 3$: place $[3, 3]$ at $j \in [1, 3 - 1]$, so $p_{r,1} \leftarrow [3, 3]$, $p_{r,2} \leftarrow [3, 3]$.
- Process $i = 4$ (may deleted): join $[4, 4]$ at $j \in [1, 4 - 1]$, so $p_{r,1} \leftarrow [4, 4]$, etc.
- Process $i = 5$ (may deleted): join $[5, 5]$ similarly.

Finally, we get $p_r = \langle [1, 4], [3, 4], [4, 5], [5, 5] \rangle$, over-approximating all possible orderings after deletion: each position reflects the join of every source element that could shift into it.

4.6.3 Logical Selection

For logical selection, the selector is a Boolean mask where `True` selects the corresponding element from the source vector, `False` skips

it, and NA is preserved. The mask is recycled to match the source length using $\rho_f^\#$ (Definition 4.4.2), mirroring the concrete semantics in Definition 2.9.3.

:= Definition,
 := Definition,

59
 35

Definition 4.6.5

Abstract Logical Selection

Given $\nu_1 = ([l_1, u_1], \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = ([l_2, u_2], \mathbf{p}_2, s_2, a_2)$, we recycle the mask:

$$\mathbf{p}'_2 = \rho_f^\#(\mathbf{p}_2, s_2, l_2, \max(|\mathbf{p}_1|, |\mathbf{p}_2|)).$$

The resulting content $\mathbf{p}_r$ is constructed position-by-position. For each position i from 1 to $|\mathbf{p}'_2|$ with selector value $p'_{2,i}$, the contribution is determined by the concretization:

$$\mathbf{p}_r = \bigoplus_{i=1}^{|\mathbf{p}'_2|} \begin{cases} \nu_1^{*\#}(i) & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{1\} \\ \nu_1^{*\#}(i) & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{0, 1\} \\ \nu_1^{*\#}(i) \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{1, \text{NA}\} \\ \nu_1^{*\#}(i) \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{0, 1, \text{NA}\} \\ \alpha_{\mathcal{B}^\#}(\text{NA}) & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{\text{NA}\} \\ \varepsilon & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{0\} \end{cases}$$

The first two cases select the element from ν_1 (definitely selected, or selected in some concretizations). The next two cases include NA in the result when the selector may be NA. When the selector is definitely NA, a single NA is emitted. When the selector is definitely False, the position contributes nothing.

Since positions that are definitely False produce no result element, we have $|\mathbf{p}_r| = |\mathbf{p}'_2| - d$, where

$$d = |\{i \in [1, |\mathbf{p}'_2|] \mid \gamma_{\text{ltv}}(p'_{2,i}) = \{0\}\}|.$$

With regard to the resulting lower bound on the length, we count mask positions whose concretization never equals False, but only among the first $\max(l_1, l_2)$ positions, since the recycled selector is guaranteed to have at least that many elements. Formally:

$$l_r = |\{i \in [1, \max(l_1, l_2)] \mid 0 \notin \gamma_{\text{ltv}}(p'_{2,i})\}|.$$

Two cases are distinguished based on whether the selector is finite or infinite.

Finite selector. When $u_2 \neq +\infty$, the upper bound is the number of mask positions that may contribute a result element. Definite False

positions produce no result, so they are subtracted from the mask length:

$$\text{Select}_{\log}^{\#}(\nu_1, \nu_2) = ([l_r, |p_r|], p_r, \perp_{\mathcal{B}^{\#}}, a_1).$$

Infinite selector. When $u_2 = +\infty$, the result may be unbounded or finite depending on the selector summary s_2 . The result summary s_r captures values selected from positions beyond the known mask content:

$$s_r = \begin{cases} \perp_{\mathcal{B}^{\#}} & \text{if } \gamma_{\text{ltv}}(s_2) = \{0\} \\ \bigsqcup_{i=|p'_2|+1}^{u_1} (p_{1,i} \sqcup_{\mathcal{B}^{\#}} \alpha_{\mathcal{B}^{\#}}(\text{NA})) & \text{if } 1 \in \gamma_{\text{ltv}}(s_2) \wedge |p'_2| \leq u_1. \\ \alpha_{\mathcal{B}^{\#}}(\text{NA}) & \text{if } 1 \in \gamma_{\text{ltv}}(s_2) \wedge |p'_2| > u_1 \end{cases}$$

The first case applies when all positions are definitely False, yielding the join-identity $\perp_{\mathcal{B}^{\#}}$. The second case joins all source elements from $|p'_2| + 1$ to u_1 with $\alpha_{\mathcal{B}^{\#}}(\text{NA})$, capturing every value that could appear when the mask is true at some position in that range. The third case applies when the mask extends beyond the source, so only NA can be selected.

When $s_r = \perp_{\mathcal{B}^{\#}}$, all positions beyond $|p'_2|$ are definitely false, so the result is finite with upper bound $|p_r|$. Otherwise, the result remains unbounded:

$$\text{Select}_{\log}^{\#}(\nu_1, \nu_2) = \begin{cases} ([l_r, |p_r|], p_r, \perp_{\mathcal{B}^{\#}}, a_1) & \text{if } s_r = \perp_{\mathcal{B}^{\#}} \\ ([l_r, +\infty], p_r, s_r, a_1) & \text{otherwise} \end{cases}.$$

4.7 Abstract Update

The update operator Update (Section 2.10) is abstracted by

⊗ Section 2.10,

36

$$\text{Update}^{\#} : \mathcal{V}_{\mathcal{B}}^{\#} \times \mathcal{V}_{\text{ltv}}^{\#} \times \mathcal{V}_{\mathcal{B}}^{\#} \rightarrow \mathcal{V}_{\mathcal{B}}^{\#}.$$

Given the source vector $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$, the selector $\nu_2 = ([l_2, u_2], p_2, s_2, a_2)$, and the replacement values $\nu_3 = ([l_3, u_3], p_3, s_3, a_3)$, the abstract update handles positive, negative, and logical indexing separately.

For integer indexing, abstract filtering of p_2 yields p_2^+ and p_2^- as defined in Section 4.6. The overall operator is:

⊗ Section 4.6,

65

$$\text{Update}^{\#}(\nu_1, \nu_2, \nu_3) = \text{Update}_{\text{pos}}^{\#}(\nu_1, \nu_2^+, \nu_3) \sqcup_{\mathcal{V}_{\mathcal{B}}^{\#}} \text{Update}_{\text{neg}}^{\#}(\nu_1, \nu_2^-, \nu_3).$$

If any element of p_2^+ or p_2^- equals $\perp_{\text{ltv}}$, the corresponding update returns $\perp_{\mathcal{V}_{\mathcal{B}}^{\#}}$. For the empty selector, we construct a pseudo-selector $\nu'_2 = ([l_1, u_1], \langle [1, 1][2, 2] \dots [u_1, u_1] \rangle, s_1, a_1)$ matching the length of the

source vector and delegate to positive update: $\text{Update}^\#(\nu_1, \langle \rangle, \nu_3) = \text{Update}_{\text{pos}}^\#(\nu_1, \nu'_2, \nu_3)$.

4.7.1 Positive Update

The positive update first applies `AdjustForZeros` to handle zero indices. Three cases are distinguished based on selector length and position enumerability.

First, compress the selector:

$$([l'_2, u'_2], p'_2, s_2, a_2) = \text{AdjustForZeros}([l_2, u_2], p_2^+, s_2, a_2).$$

Definition 4.7.1

Positive Update, Non-Enumerable Positions

When at least one selected position is not enumerable, we cannot track individual positions. The entire vector is collapsed:

$$\text{Update}_{\text{pos}}^\#(\nu_1, \nu_2^+, \nu_3) = ([l_1, +\infty], \varepsilon, v, a_1),$$

where $v = \text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\#} \text{Squash}(\nu_3)$ captures all possible values (original or replacement) that any position may take.

When at least one selector position has a non-enumerable concretization, the analysis cannot track which individual positions of the source vector are affected. The sound over-approximation collapses the entire vector to an infinite-length abstract element whose summary joins the source and replacement values: every position may take any value from either the original or the replacement vector.

Before presenting the remaining cases, we introduce two helper functions for content manipulation.

Definition 4.7.2

InitKnown

The function $\text{InitKnown} : \mathcal{B}^{\#*} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathcal{B}^{\#*}$ initializes a content sequence with a given pattern, preserving the first l elements, joining the next $u - l$ elements with $\alpha_{\mathcal{V}_B^\#}(\text{NA})$, and extending with $\alpha_{\mathcal{V}_B^\#}(\text{NA})$ up to length u_r :

$$\begin{aligned} \text{InitKnown}(p, l, u, u_r) &\stackrel{\text{def}}{=} \langle p_1 \cdots p_l \rangle \\ &\oplus \underbrace{\left(p_{l+1} \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{V}_B^\#}(\text{NA}) \right) \oplus \cdots \oplus \left(p_u \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{V}_B^\#}(\text{NA}) \right)}_{u-l} \\ &\oplus \underbrace{\left\langle \alpha_{\mathcal{V}_B^\#}(\text{NA}) \cdots \alpha_{\mathcal{V}_B^\#}(\text{NA}) \right\rangle}_{u_r - u}, \end{aligned}$$

where $l \leq u \leq u_r$.

Definition 4.7.3**UpdateKnown**

The function $\text{UpdateKnown} : \mathcal{B}^{\#*} \times \text{Itv}^* \times \mathcal{B}^{\#*} \rightarrow \mathcal{B}^{\#*}$ recursively updates positions in a target content based on resolved selector positions and replacement values. Given $p \in \mathcal{B}^{\#}$, $c \in \text{Itv}^* : c = \langle c_1 \dots c_n \rangle$ and $v \in \mathcal{B}^{\#*} : v = \langle v_1 \dots v_m \rangle$, we have:

$$\begin{aligned} \text{UpdateKnown}(p, \varepsilon, _) &\stackrel{\text{def}}{=} p \\ \text{UpdateKnown}(p, c, v) &\stackrel{\text{def}}{=} \text{UpdateKnown}(p', c', v'), \end{aligned}$$

where p' performs the update for position c_1 :

$$p' = \begin{cases} p \left[i \leftarrow \begin{cases} v_1 & \text{if } \text{card}(c_1) = 1 \\ p_i \sqcup_{\mathcal{B}^{\#}} v_1 & \text{otherwise} \end{cases} \right] & \begin{matrix} i \in \gamma_{\text{Itv}}(c_1) \\ \wedge i \neq 0 \\ \wedge i \neq \text{NA} \end{matrix} \text{ if } \text{Enumerable}(c_1) \\ p[i \leftarrow p_i \sqcup_{\mathcal{B}^{\#}} v_1]_{i \in [1, |p|]} & \text{otherwise} \end{cases}$$

and $c' = \langle c_2 \dots c_n \rangle$, $v' = \langle v_2 \dots v_m \rangle$.

When the selector position is enumerable, a strong update is applied if the abstract value is a singleton, i.e., $\text{card}(c_1) = 1$, and a weak update otherwise. When non-enumerable, all positions are weakly updated.

We are now able to introduce the cases differentiation.

Definition 4.7.4**Positive Update,
Infinite Selector,
Enumerable Positions**

When $u'_2 = +\infty$ and all positions in p'_2 are enumerable, we distinguish two sub-cases based on the summary.

Non-enumerable summary. Let

$$[l_r, u_r] = \bigsqcup_{i=1}^{|p'_2|} p'_{2,i}$$

be the join of all selector positions. We initialize the base content preserving the first l_1 elements of p_1 , joining positions $l_1 + 1$ to $|p_1|$ with $\alpha_{\mathcal{V}_B^{\#}}(\text{NA})$, and extending with $\alpha_{\mathcal{V}_B^{\#}}(\text{NA})$ up to $\max(|p_1|, u_r, l_{s_2})$:

$$p'_r = \text{InitKnown}(p_1, l_1, |p_1|, \max(|p_1|, u_r, l_{s_2})).$$

When $u_1 = +\infty$, the source vector is infinite and its summary s_1 describes all positions beyond $|p_1|$. Since `InitKnown` fills those positions with $\alpha_{\mathcal{B}^\#}(\text{NA})$, we overwrite them with the actual information from s_1 via a strong update:

$$p'_r[i \leftarrow s_1]_{i \in [|p_1|+1, \max(|p_1|, u_r, l_{s_2})]}.$$

Recycled replacement values are $p'_3 = \rho_f^\#(p_3, s_3, l_3, u_r)$. We apply:

$$p''_r = \text{UpdateKnown}(p'_r, p'_2, p'_3).$$

Then, when $l_{s_2} \leq \max(|p_1|, u_r)$, the summary covers positions up to $\max(|p_1|, u_r)$, so we weakly update those positions:

$$p_r = p''_r[i \leftarrow p_i \sqcup_{\mathcal{B}^\#} \text{Squash}(\nu_3)]_{i \in [l_{s_2}, \max(|p_1|, u_r)]}.$$

Let l_2^b be the maximum of the lower bounds of the first l'_2 positions of p'_2 , i.e., the largest index guaranteed to be selected by at least one of those positions in every concrete selector of length at least l'_2 . The result has length $[\max(l_1, l_2^b, l_{s_2}), +\infty]$ and summary $s_r = s_1 \sqcup_{\mathcal{B}^\#} \text{Squash}(\nu_3) \sqcup_{\mathcal{B}^\#} \alpha_{\mathcal{B}^\#}(\text{NA})$:

$$\text{Update}_{\text{pos}}^\#(\nu_1, \nu_2^+, \nu_3) = ([\max(l_1, l_2^b, l_{s_2}), +\infty], p_r, s_r, a_1).$$

Enumerable summary. When the summary of the infinite selector is enumerable, the bound of modified positions is finite. Let l_r, u_r be the lower and upper bounds of $\text{Squash}(\nu_2)$. The base content is initialized as in the previous case, with length $\max(|p_1|, u_r)$:

$$p'_r = \text{InitKnown}(p_1, l_1, |p_1|, \max(|p_1|, u_r)).$$

When $u_1 = +\infty$, we overwrite the extended positions with the source summary as above:

$$p'_r[i \leftarrow s_1]_{i \in [|p_1|+1, \max(|p_1|, u_r)]}.$$

We generate the replacement values using $\rho_f^\#$:

$$p'_3 = \rho_f^\#(p_3, s_3, l_3, u'_2).$$

and safely update p'_r :

$$p''_r = \text{UpdateKnown}(p'_r, p'_2, p'_3).$$

Finally, we join positions indicated by the summary with $\text{Squash}(\nu_3)$:

$$p_r = p''_r[i \leftarrow p_i \sqcup_{\mathcal{B}^\#} \text{Squash}(\nu_3)]_{i \in [l_{s_2}, \max(|p_1|, u_r)]}.$$

Let l_2^b be defined as above. We have the resulting length $[\max(l_1, l_2^b, l_{s_2}), \max(u_1, u_r)]$. When $u_1 \neq +\infty$, the source is finite so $s_1 = \perp_{\mathcal{B}^\#}$; when $u_1 = +\infty$, the source is infinite but the selector's enumerable summary constrains its positions to the finite range $[l_r, u_r]$,

so positions beyond this range are never selected and the source summary remains unchanged. In either case the summary is s_1 :

$$\text{Update}_{\text{pos}}^{\#}(\nu_1, \nu_2^+, \nu_3) = ([\max(l_1, l_2^b, l_{s_2}), \max(u_1, u_r)], p_r, s_1, a_1).$$

When the selector is infinite and all positions are enumerable, the construction depends on whether the summary of the selector is itself enumerable. In the non-enumerable summary case, the selector summary covers an unbounded set of positions beyond any finite prefix; the result is therefore always infinite, regardless of the source. The summary joins the source summary, the replacement values, and $\alpha_{\mathcal{B}^+}(\text{NA})$: since the selector's non-enumerable summary may skip positions that lie beyond the source range, those positions are NA and must be accounted for. In the enumerable summary case, the selector's summary bounds the positions to a finite range $[l_r, u_r]$, so the selector never reaches the infinite tail; the finiteness of the result, and its summary, is determined by the source alone.

Example

Lower bound l_2^b

Consider a source vector $\nu_1 = ([2, 3], \langle [1, 1], [2, 2], [3, 3] \rangle, \perp_{\text{ltv}}, \text{AttrEmpty})$ and two infinite selectors after AdjustForZeros:

$$\nu_2'^A = ([2, +\infty], \langle [1, 1], [1, 1], [5, 5], [6, 6] \rangle, [1, +\infty], \text{AttrEmpty}),$$

$$\nu_2'^B = ([2, +\infty], \langle [1, 1], [4, 4], [5, 5], [6, 6] \rangle, [1, +\infty], \text{AttrEmpty}).$$

Both selectors have minimum length $l_2' = 2$, so the first two positions are guaranteed to appear in every concretization, and a non-enumerable summary $s_2 = [1, +\infty]$ with $l_{s_2} = 1$. In selector A , both guaranteed positions are $[1, 1]$, giving $l_2^b = \max(1, 1) = 1$: the largest index forced by the selector is 1, so the result lower bound $\max(l_1, l_2^b, l_{s_2}) = \max(2, 1, 1) = 2$ is unchanged from the source. In selector B , the guaranteed positions are $[1, 1]$ and $[4, 4]$, giving $l_2^b = \max(1, 4) = 4$: the selector forces at least position 4 to be updated, raising the result lower bound to $\max(2, 4, 1) = 4$. This captures the fact that an abstract selector whose minimum concrete length is l_2' still covers at least that many positions, and the maximum lower bound among them is the smallest index that every concretization is guaranteed to select.

Definition 4.7.5

Positive Update, Finite Selector, Enumerable Positions

When $u_2' \neq +\infty$ and all positions are enumerable, the result is:

$$\text{Update}_{\text{pos}}^{\#}(\nu_1, \nu_2^+, \nu_3) = ([\max(l_1, u_r), \max(u_1, u_r)], p_r, s_r, a_1),$$

where $[l_r, u_r] = \bigcup_{i=1}^{|p'_2|} p'_{2,i}$ gives the range of positions targeted by the selector. If $u_r > u_1$, the vector is extended: positions between the original content and the new indices are filled with $\alpha_{\mathcal{V}_B^\#}(\text{NA})$.

Recycled replacement values are $p'_3 = \rho_f^\#(p_3, s_3, l_3, u'_2)$. Zero indices are handled as in the positive selection case; we initialize:

$$p'_r = \begin{cases} p_1 & \text{if } u_r \leq u_1 \\ \text{InitKnown}(p_1, l_r, \min(|p_1|, u_1), \max(u_1, u_r)) & \text{if } l_1 < l_r \leq u_1 \\ \text{InitKnown}(p_1, l_1, \min(|p_1|, u_1), \max(u_1, u_r)) & \text{otherwise} \end{cases}$$

and update via UpdateKnown:

$$p_r = \text{UpdateKnown}(p'_r, p'_2, p'_3).$$

The summary is:

$$s_r = \begin{cases} \perp_{\mathcal{B}^\#} & \text{if } u_1 \neq +\infty \\ s_1 \sqcup_{\mathcal{B}^\#} \text{Squash}(\nu_3) & \text{otherwise} \end{cases}$$

When the selector is finite and all positions are enumerable, the resulting length is bounded by the maximum of the source and selector upper bounds. The three-case InitKnown construction handles: the selector targets only positions within the existing content (no extension needed), the selector extends beyond the original content starting after l_1 , and the general case where the source content is extended with $\alpha_{\mathcal{V}_B^\#}(\text{NA})$. The summary remains $\perp_{\mathcal{B}^\#}$ when the source is finite, and joins with Squash(ν_3) when the source is infinite to cover tail positions.

4.7.2 Negative Update

Given $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$, $\nu_2^- = ([l_2, u_2], p_2^-, s_2, a_2)$, and $\nu_3 = ([l_3, u_3], p_3, s_3, a_3)$, we first compress the selector:

$$\nu_2'^- = ([l'_2, u'_2], p'_2, s_2, a_2) = \text{AdjustForZeros}([l_2, u_2], p_2^-, s_2, a_2).$$

We identify three sets of positions:

- **MustNotUpdated**: positions definitely not updated, i.e., where $\text{card}(p_{2,i}^-) = 1$ and the concretization is a valid positive index in $[-u_1, -1]$. These are positions excluded by precise singleton negative indices.
- **MayNotUpdated**: positions possibly excluded, including those with $\text{card} \geq 1$ whose concretization intersects $[-u_1, -1]$, plus all positions covered by the summary.
- **MustUpdated**: positions definitely updated, computed as $[1, |p_1|] \setminus \text{MayNotUpdated}$.

We distinguish three cases based on the properties of the adjusted selector.

Definition 4.7.6**Negative Update,
Non-Enumerable Position**

When at least one position in p'_2 is not enumerable, we weakly update all positions not in `MustNotUpdated`:

$$\text{Update}_{\text{neg}}^{\#}(\nu_1, \nu_2^-, \nu_3) = ([l_1, u_1], p_r, s_r, a_1),$$

where:

$$p_r = p_1 [i \leftarrow p_{1,i} \sqcup_{\mathcal{B}^{\#}} \text{Squash}(\nu_3)]_{i \in [1, |p_1|] \setminus \text{MustNotUpdated}}$$

$$s_r = \begin{cases} s_1 \sqcup_{\mathcal{B}^{\#}} \text{Squash}(\nu_3) & \text{if } u_1 = +\infty \\ \perp_{\mathcal{B}^{\#}} & \text{otherwise} \end{cases}.$$

When at least one adjusted selector position is non-enumerable, the analysis cannot determine which individual source positions are targeted for deletion. The sound over-approximation weakens every position not in `MustNotUpdated` with the replacement summary while preserving the source length. This is the most conservative case: the only guarantee is that positions in `MustNotUpdated` remain untouched.

When all adjusted selector positions are enumerable, the sets `MustUpdated` and `MustNotUpdated` provide the precision needed for a more refined construction. The following definition handles the case where the selector is unbounded, building a positive-index selector that delegates to the positive update.

Definition 4.7.7**Negative Update,
Infinite Selector,
All Enumerable**

When $u'_2 = +\infty$ and all positions are enumerable, we build a positive-index selector from the position sets and delegate to positive update. Define the selector length $[l_r, u_r] = [| \text{MustUpdated}_{l_1} |, u_1 - | \text{MustNotUpdated}_{u_1} |]$ and a content of length u_1 initialized to $\perp_{\text{Itv}}$, i.e., $p_{\nu_s} = \langle \perp_{\text{Itv}} \cdots \perp_{\text{Itv}} \rangle$.

Let $\text{CountMustNotUpdated}(i) = |\{u \in \text{MustNotUpdated} \mid u < i\}|$. Construct p_{ν_s} :

1. For each $i \in \text{MustUpdated}$ (ascending):

$$\forall j \in [1, i - \text{CountMustNotUpdated}(i)] : p_{\nu_s, j} \leftarrow p_{\nu_s, j} \sqcup_{\text{Itv}} [i, i].$$

2. For each $i \in \text{MayNotUpdated}$ (ascending):

$\forall j \in [1, i - \text{CountMustNotUpdated}(i)] : p_{\nu_s, j} \leftarrow p_{\nu_s, j} \sqcup_{\text{ltv}} [i, i] .$

The selector is $\nu_s = ([l_r, u_r], p_{\nu_s}, s_1, a_1)$. Then:

$$\text{Update}_{\text{neg}}^{\#}(\nu_1, \nu_2^-, \nu_3) = \text{Update}_{\text{pos}}^{\#}(\nu_1, \nu_s, \nu_3).$$

Every index $i \in [1, |p_1|]$ belongs to either `MustUpdated` or `MayNotUpdated`, so the constructed selector p_{ν_s} has no $\perp_{\text{ltv}}$ entries and covers all positions that may be updated.

When the selector is finite and all positions are enumerable, the construction simplifies because the summary s_2 contributes no positions, i.e., the selector targets only a bounded number of positions.

Definition 4.7.8

Negative Update, Finite Selector, Enumerable Positions

When $u'_2 \neq +\infty$ and all positions are enumerable, we proceed as in the infinite case but with `MayNotUpdated` restricted to only the non-singleton abstract values from the content. We build the selector ν_s as in the infinite case and delegate to the positive update:

$$\text{Update}_{\text{neg}}^{\#}(\nu_1, \nu_2^-, \nu_3) = \text{Update}_{\text{pos}}^{\#}(\nu_1, \nu_s, \nu_3).$$

4.7.3 Logical Update

Given $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$, $\nu_2 = ([l_2, u_2], p_2, s_2, a_2)$, and $\nu_3 = ([l_3, u_3], p_3, s_3, a_3)$, the logical update assigns values to positions selected by a Boolean mask.

First, if the selector contains NA, we check a special constraint: if $\alpha_{\mathcal{B}^{\#}}(\text{NA}) \in \gamma_{\mathcal{B}^{\#}}(\text{Squash}(\nu_2))$ and $u_3 \neq 1$, the update is undefined and returns $\perp_{\gamma_{\mathcal{B}^{\#}}}$, since assigning multiple values with an NA-containing selector produces an error in the concrete semantics (Definition 2.10.3). If the constraint holds and the filtered length is $[1, 1]$, the singleton replacement recycles identically regardless of which NA positions are selected.

:= Definition,

38

Definition 4.7.9

Abstract Logical Update

For the standard case, the resulting content is initialized from the source vector:

$$p_r = \text{InitKnown}(p_1, l_1, |p_1|, \max(|p_1|, |p_2|)).$$

The mask and the replacement values are recycled to the same length:

$$\begin{aligned} p'_2 &= \rho_f^\#(p_2, s_2, l_2, \max(|p_1|, |p_2|)), \\ p'_3 &= \rho_f^\#(p_3, s_3, l_3, \max(|p_1|, |p_2|)). \end{aligned}$$

Writing $p_{r,i}$, $p_{3,i}$, and $p'_{2,i}$ for the i -th elements of p_r , p'_3 , and p'_2 respectively, the per-position update is:

$$p_r = \bigoplus_{i=1}^{|p_r|} \begin{cases} p_r[i \leftarrow p_{3,i}] & \text{if } \gamma_{\text{ltv}}(p'_{2,i}) = \{1\} \\ p_r[i \leftarrow p_{r,i} \sqcup_{B^\#} p_{3,i}] & \text{if } 1 \in \gamma_{\text{ltv}}(p'_{2,i}) \\ p_r & \text{otherwise} \end{cases}$$

A strong update replaces the position with the recycled replacement value when the mask is definitely true. A weak update is applied when the mask may be true. When the mask cannot be true, the position is unchanged.

The minimum number of definitely selected positions is $d_r = |\{i \in [1, l_2] \mid \gamma_{\text{ltv}}(p_{2,i}) = \{1\}\}|$. The overall result is:

$$\text{Update}_{\log}^\#(\nu_1, \nu_2, \nu_3) = \begin{cases} ([\max(l_1, d_r), +\infty], p_r, \text{Squash}(\nu_3), a_1) & \text{if } u_2 = +\infty \\ ([\max(l_1, d_r), |p_r|], p_r, \perp_{B^\#}, a_1) & \text{if } u_2 \neq +\infty \end{cases}$$

When the selector is infinite, the result is unbounded and the summary captures all replacement values for positions beyond the known content.

4.8 Chapter Summary

This chapter lifted the concrete R vector operators defined in Chapter 2 to the abstract domain $\mathcal{V}_B^\#$. We covered the abstract counterparts of the operators defined in Chapter 2: abstract type coercion (Section 4.1), abstract attribute manipulation (Section 4.2) via $\text{setAttr}^\#$, abstract concatenation (Section 4.3.2) with the helper functions Squash and SquashExcept , and abstract vector recycling (Section 4.4) with the helpers $\rho_c^\#$ and $\rho_f^\#$. We proved Theorem 4.4.1, establishing that $\text{Recycle}^\#$ soundly abstracts Recycle , through a chain of four lemmas addressing padding, cycling, join preservation, and summary coverage.

Building on these foundations, we introduced a set of shared helper functions (Section 4.5): abstract cardinality card , the enumerability test Enumerable , zero propagation Propagate , selector compression AdjustForZeros , and abstract element access $\nu^{*\#}(k)$. These cover common patterns across both selection and update.

We then presented the abstract selection operator $\text{Select}^\#$ (Section 4.6) and the abstract update operator $\text{Update}^\#$ (Section 4.7), covering positive, negative, and logical indexing modes for each. Soundness proofs for the recycling and positive selection operators are in Appendix B.

↻ Chapter 2,	25
↻ Chapter 2,	25
↻ Section 4.1,	55
↻ Section 4.2,	56
↻ Section 4.3.2,	57
↻ Section 4.4,	58
□ Theorem,	61
↻ Section 4.5,	61
↻ Section 4.6,	65
↻ Section 4.7,	71
↻ Appendix B.,	102

Together, these operators form the static analysis framework. The overall design decomposes each abstraction into helper functions for specific concerns: zero handling, position enumeration, content initialization.

From Abstract Operators to Program Analysis

With the individual abstract operators established in Chapter 4, we now assemble them into a static analysis that over-approximates the behavior of μR programs. We define the abstract expression and statement semantics that lift the collecting semantics of Chapter 2 to the abstract domain $\mathcal{V}_B^\#$, then describe the realization of these semantics within the flowR static analysis framework [29]. The chapter closes with an instantiation of the parametric abstract domain, an experimental validation against a suite of test programs, and a discussion of limitations.

The chapter proceeds as follows. We first integrate the individual abstract operators into expression evaluation in Section 5.1, then extend the analysis to statements with fixpoint computation via widening in Section 5.2. The pipeline-level architecture of the flowR realization is described in Section 5.3, mapping R constructs to abstract operators. Section 5.4 instantiates the parametric $\mathcal{V}_B^\#$ with the interval domain *Itv* and details the NA-aware wrapping that preserves precision. Finally, Section 5.5 presents the experimental validation across 61 handcrafted test cases and discusses the results.

5.1 Abstract Expression Semantics

To evaluate μR expressions under abstraction, we combine the operators defined in the previous chapters. We define the abstract environment $\mathcal{E}^\# : \text{Var} \rightarrow \mathcal{V}_B^\#$ as a partial function from variables to abstract vectors, mirroring the concrete environment $\mathcal{E}$ defined in Chapter 2. An abstract expression denotes a function that takes an abstract environment and returns an updated environment together with an abstract vector.

The abstract expression semantics $\mathbb{E}^\#[\cdot] : (E \times \mathcal{E}^\#) \rightarrow (\mathcal{E}^\# \times \mathcal{V}_B^\#)$ is defined in Table 4¹ by case analysis on the expression syntax of μR , using the abstract operators of Chapter 4.

1 Abstract Expression Semantics,	81
2 Abstract Statement Semantics,	83
3 Realization in flowR,	83
4 Domain Instantiation,	84
5 Validation, Chapter 4,	85 55

Chapter 2,	25
------------	----

[29] Sihler et al., 2025
*Statically Analyzing the
Dataflow of R Programs*,

Section 5.1,	81
--------------	----

Section 5.2,	83
--------------	----

Section 5.3,	83
--------------	----

Section 5.4,	84
--------------	----

Section 5.5,	85
--------------	----

Chapter 2,	25
------------	----

1. For readability, multi-character code symbols are rendered as ligatures in the graphical output; the actual language tokens are the corresponding ASCII sequences.

Chapter 4,	55
------------	----

$$\mathbb{E}^\# \llbracket e \rrbracket : (E \times \mathcal{E}^\#) \rightarrow (\mathcal{E}^\# \times \mathcal{V}_{\mathcal{B}}^\#), \quad \text{where } a \in \text{Atm}, x \in \text{Var}, e \in E, s \in \text{String}$$

$$\begin{aligned} \mathbb{E}^\# \llbracket a \rrbracket \rho^\# &\stackrel{\text{def}}{=} \begin{cases} (\rho^\#, \rho^\#(a)) & \text{if } a \in \text{Var} \\ (\rho^\#, ([1, 1], \langle \alpha_{\mathcal{B}^\#}(a) \rangle, \perp_{\mathcal{B}^\#}, \text{AttrEmpty})) & \text{if } a \in \text{Float} \cup \text{Int} \cup \text{Bool} \\ (\rho^\#, ([0, 0], \varepsilon, \perp_{\mathcal{B}^\#}, \text{AttrEmpty})) & \text{if } a \equiv \text{NULL} \end{cases} \\ \mathbb{E}^\# \llbracket -e \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho', ([l, u], \langle -_{\mathcal{B}^\#} p_1 \cdots -_{\mathcal{B}^\#} p_k \rangle, -_{\mathcal{B}^\#} s, \text{AttrEmpty})) \\ &\quad \text{where } (\rho', \nu) = \mathbb{E}^\# \llbracket e \rrbracket \rho^\#, \nu = ([l, u], \langle p_1 \cdots p_k \rangle, s, \text{AttrEmpty}) \\ \mathbb{E}^\# \llbracket !e \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho', ([l, u], \langle \overline{\text{ToBool}^\#}(p_1) \cdots \overline{\text{ToBool}^\#}(p_k) \rangle, \overline{\text{ToBool}^\#}(s), \text{AttrEmpty})) \\ &\quad \text{where } (\rho', \nu) = \mathbb{E}^\# \llbracket e \rrbracket \rho^\#, \nu = ([l, u], \langle p_1 \cdots p_k \rangle, s, \text{AttrEmpty}) \\ \mathbb{E}^\# \llbracket e_1 \diamond e_2 \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho_2, ([l', u'], \langle p'_{1,1} \diamond_{\mathcal{B}^\#} p'_{2,1} \cdots p'_{1,n} \diamond_{\mathcal{B}^\#} p'_{2,n} \rangle, s'_1 \diamond_{\mathcal{B}^\#} s'_2, \text{AttrEmpty})) \\ &\quad \text{where } (\rho_1, \nu_1) = \mathbb{E}^\# \llbracket e_1 \rrbracket \rho^\# \\ &\quad (\rho_2, \nu_2) = \mathbb{E}^\# \llbracket e_2 \rrbracket \rho_1' \\ &\quad \text{Recycle}^\#(\nu_1, \nu_2) = (\nu'_1, \nu'_2) \\ \mathbb{E}^\# \llbracket c() \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho^\#, ([0, 0], \varepsilon, \perp_{\mathcal{B}^\#}, \text{AttrEmpty})) \\ \mathbb{E}^\# \llbracket c(e_1, \dots, e_n) \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho_n, \nu_1 \oplus^\# \cdots \oplus^\# \nu_n) \\ &\quad \text{where } (\rho_i, \nu_i) = \mathbb{E}^\# \llbracket e_i \rrbracket \rho_{i-1} \text{ with } \rho_0 = \rho^\# \\ \mathbb{E}^\# \llbracket x \leftarrow e \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho' [x \mapsto \nu], \nu) \\ &\quad \text{where } (\rho', \nu) = \mathbb{E}^\# \llbracket e \rrbracket \rho^\# \\ \mathbb{E}^\# \llbracket x[e_{\text{idx}}] \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho', \text{Select}^\#(\nu, \nu_{\text{idx}})) \\ &\quad \text{where } \nu = \rho'(x), (\rho', \nu_{\text{idx}}) = \mathbb{E}^\# \llbracket e_{\text{idx}} \rrbracket \rho^\# \\ \mathbb{E}^\# \llbracket x[[e_{\text{idx}}]] \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho', \text{Select}^\#(\nu_v, \nu_{\text{idx}})) \\ &\quad \text{where } (\rho', \nu_{\text{idx}}) = \mathbb{E}^\# \llbracket e_{\text{idx}} \rrbracket \rho^\#, [l_{\nu_{\text{idx}}}, u_{\nu_{\text{idx}}}] = [1, 1] \\ \mathbb{E}^\# \llbracket x[e_{\text{idx}}] \leftarrow e \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho_2 [x \mapsto \text{Update}^\#(\nu, \nu_{\text{idx}}, \nu_{\text{val}})], \nu_{\text{val}}) \\ &\quad \text{where } (\rho_1, \nu_{\text{val}}) = \mathbb{E}^\# \llbracket e \rrbracket \rho^\# \\ &\quad (\rho_2, \nu_{\text{idx}}) = \mathbb{E}^\# \llbracket e_{\text{idx}} \rrbracket \rho_1 \\ &\quad \nu = \rho_2(x) \\ \mathbb{E}^\# \llbracket \text{getAttr}(x) \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho^\#, \nu) \\ &\quad \text{where } \nu = \rho^\#(x) \\ \mathbb{E}^\# \llbracket \text{setAttr}(x, \{s_1, \dots, s_n\}) \rrbracket \rho^\# &\stackrel{\text{def}}{=} (\rho^\# [x \mapsto \nu'], \nu') \\ &\quad \text{where } \nu = \rho^\#(x), \nu' = \text{setAttr}^\#(\nu, \{s_1, \dots, s_n\}) \end{aligned}$$

Table 4: Abstract semantics of μR expressions.

The atoms case handles three sub-cases. A variable $a \in \text{Var}$ is looked up in the abstract environment $\rho^\#$, returning the abstract vector stored there. Numeric and Boolean literals $a \in \text{Float} \cup \text{Int} \cup \text{Bool}$ are abstracted to a vector of length $[1, 1]$ with the abstract value $\alpha_{\mathcal{B}^\#}(a)$ at position 1, the summary $\perp_{\mathcal{B}^\#}$ (a finite-length vector needs no

summary), and the empty attribute abstraction `AttrEmpty`. The null literal `NULL` yields the empty vector $([0, 0], \varepsilon, \perp_{\mathcal{B}^\#}, \text{AttrEmpty})$, consistent with its concrete interpretation as a vector of length zero.

Unary operators negate the content. The arithmetic negation $-$ maps each known position p_i to $-\mathcal{B}^\# p_i$ and the summary s to $-\mathcal{B}^\# s$, using the abstract negation operator $-\mathcal{B}^\#$ of $\mathcal{B}^\#$. The logical negation $!$ applies $\overline{\text{ToBool}}^\#$ element-wise and to the summary, mirroring the concrete $\overline{\text{ToBool}}$.

Binary operators $e_1 \diamond e_2$ first recycle the operand vectors via $\text{Recycle}^\#$ to a common length, producing ν'_1 and ν'_2 . The result is then obtained by applying the abstract binary operator $\diamond_{\mathcal{B}^\#}$ position-wise and to the summaries. This mirrors the concrete pattern of recycling followed by element-wise operation, established in Chapter 4.

Chapter 4,

55

Concatenation $c(e_1, \dots, e_n)$ evaluates each sub-expression in sequence, threading the environment through, and joins the resulting abstract vectors via iterated $\oplus^\#$. The zero-argument form $c()$ returns the empty vector directly. Variable assignment $x \leftarrow e$ updates the abstract environment at x with the result of evaluating e .

Selection $x[e_{idx}]$ evaluates the selector and applies $\text{Select}^\#$ to the vector bound to x . The double-bracket extraction $x[[e_{idx}]]$ additionally constrains the selector to a single positive index, using the same $\text{Select}^\#$ operator but restricted to a singleton result. Update $x[e_{idx}] \leftarrow e$ evaluates the right-hand side and the selector, then applies $\text{Update}^\#$ to the vector bound to x , updating the environment accordingly. Attribute manipulation uses $\text{setAttr}^\#$ to replace the attribute abstraction. Retrieving attributes returns the abstract vector itself, since the $\text{Attr}^\#$ component is its fourth element and can be extracted by the caller.

5.2 Abstract Statement Semantics

The abstract statement semantics $\mathcal{S}^\#[\cdot]$ lifts the concrete statement semantics to abstract environments over $\mathcal{V}_{\mathcal{B}}^\#$, reusing the operators of Chapter 4 unchanged. The expression statement, sequence, and conditional are standard: they mirror the collecting semantics, with the conditional replacing the collecting union by a pointwise join over Var . The while loop requires widening: since $\mathcal{V}_{\mathcal{B}}^\#$ admits infinite ascending chains, naive fixpoint iteration may not terminate. We replace it with a widened iteration using $\nabla_{\mathcal{V}_{\mathcal{B}}^\#}$ (Chapter 3), which computes a sound post-fixpoint over-approximating the collecting fixpoint.

Chapter 3,

41

5.3 Realization in flowR

The abstract domain and operators are realized as an extension of flowR [29], a static analysis framework for the R programming language.² This section describes the integration at the pipeline level, avoiding code-level detail. The purpose is to show that the abstract domain is not merely a theoretical construction but can be implemented and executed on concrete R programs.

[29] Sihler et al., 2025
Statically Analyzing the
Dataflow of R Programs,
2. <https://github.com/manueldiagostino/flowr>

5.3.1 Analysis Pipeline

flowR processes R source code through a sequence of stages. The R source is parsed into an abstract syntax tree and normalized into a canonical form that desugars syntactic variants. flowR then performs dataflow analysis to construct both a dataflow graph and a **CFG** simultaneously; unlike traditional compilers, the CFG is not constructed before dataflow analysis but emerges from it. The CFG is then traversed by an abstract interpretation visitor that propagates abstract states through the vertices according to the semantics defined in this chapter.

The pipeline is designed to handle the full R language at the syntactic level, though the abstract interpretation visitor only supports the μR subset. When the visitor encounters R constructs outside μR (such as function definitions, closures, or non-standard evaluation), it falls back to $\top_{\mathcal{V}_B^\#}$, the least precise abstract element. This design allows the analysis to be applied to real R programs while clearly delimiting its soundness guarantee: the analysis is sound for the μR subset and conservatively over-approximates all other constructs.

5.3.2 The Vector Inference Visitor

The core of the implementation is the `VectorInferenceVisitor`, which extends flowR's generic `AbstractInterpretationVisitor`. The generic visitor provides the fixpoint iteration algorithm: forward **CFG** visitation via a **Worklist Algorithm**, join at merge points, delayed widening at loop headers, and fixpoint detection. The `VectorInferenceVisitor` specializes this process for the vector domain.

The visitor decomposes the abstract state into the five sub-domains described in Section 5.4 and dispatches each CFG vertex to a handler that applies the corresponding abstract operator. Table 5 summarizes this mapping.

R Construct	Abstract Operator	Reference
<code>c()</code>	$\oplus^\#$	Section 4.3.2
<code>v $\diamond$ w</code>	$\text{Recycle}^\#, \diamond_{\mathcal{B}^\#}$	Section 4.4
<code>-v</code>	$\neg_{\mathcal{B}^\#}$	Section 5.1
<code>!v</code>	$\overline{\text{ToBool}}^\#$	Section 5.1
<code>v[i]</code>	$\text{Select}^\#$	Section 4.6
<code>v[[i]]</code>	$\text{Select}^\#$	Section 4.6
<code>v[i] $\leftarrow$ x</code>	$\text{Update}^\#$	Section 4.7

Table 5: Mapping from R constructs to abstract operators.

5.4 Domain Instantiation

Chapter 3 defined $\mathcal{V}_B^\#$ parametrically over an abstract value domain $\mathcal{B}^\#$. This section instantiates that parameter for the experimental evaluation in Section 5.5.

CFG, Control Flow Graph. A directed graph representation of a program where nodes correspond to basic blocks (straight-line sequences of instructions) and edges represent possible transfers of control. The CFG is the standard data structure underlying data-flow analyses and abstract interpretation: analysis information propagates along edges from one node to the next.

Worklist Algorithm. The standard fixpoint iteration engine used in abstract interpretation. Maintains a queue of program points whose outgoing abstract state has changed; at each step, a point is dequeued, its transfer function is re-evaluated, and successors whose state changed are added back to the queue. The algorithm terminates when the worklist is empty, indicating that a fixpoint has been reached. Combined with widening at loop headers, it guarantees termination even for lattices with infinite ascending chains.

5.4.1 Scalar Domain

We instantiate $\mathcal{B}^\#$ as Itv , the interval domain introduced in Section 1.4.4. An abstract value is an interval $[a, b]$ with $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$ and $a \leq b$, denoting the integers between a and b . Booleans are encoded as integers (True to $[1, 1]$, False to $[0, 0]$). Floating-point numbers are not handled in this evaluation for simplicity; extending to floating-point abstract intervals is straightforward.

↻ Section 1.4.4,

16

As noted in Remark, abstracting NA directly as an interval is problematic. Representing NA as $\top_{\mathcal{B}^\#}$ would cause any join with NA to collapse all value information to $\top_{\mathcal{B}^\#}$. To avoid this, the content domain is the product $\text{Itv} \times \{0, 1\}^\top$, where the second component is an NA-presence flag. An element (I, n) pairs an interval I with a flag n : $n = 0$ means NA is absent, $n = 1$ means NA is present. The abstraction of NA is $(\perp_{\text{Itv}}, 1)$; a concrete value c abstracts to $(\alpha_{\text{Itv}}(c), 0)$. Lattice operations are component-wise: the join propagates the NA flag via max, the meet via min. This separation ensures that value tracking and NA tracking remain independent throughout the analysis.

* Remark,

65

5.4.2 Domain Components

The vector abstract domain is realized as a product of five subdomains, composed using flowR's product domain facility:

- **length**: `PosIntervalDomain`, the vector length as a positive integer interval $[l, u]$.
- **known**: `KnownInitialPositionsDomain`, a finite sequence of NA-aware scalar domain elements, each an abstract value at a specific tracked position.
- **summary**: `NAAwareDomain`, a single NA-aware element covering all positions beyond the known content.
- **attributes**: `VectorAttrDomain`, a must/may pair of attribute sets, as described in Chapter 3.
- **type**: `RVectorTypeDomain`, an element of the coercion lattice $\mathcal{T}$, tracking the vector's R type.

↻ Chapter 3,

41

The known content and summary together implement the dual representation of $\mathcal{V}_{\mathcal{B}}^\#$: individual positions are tracked precisely where possible, while the summary collapses all remaining positions into a single conservative element.

5.5 Validation

The original paper validates the correctness and precision of the abstract domain against a suite of handcrafted test programs. This section describes the validation methodology, presents representative test cases, and discusses the results and limitations.

5.5.1 Methodology

We adopt a dual verification approach. Each test case consists of an R code snippet and its expected abstract value, specified as a tuple $(L, \mathbf{p}, s, a)$ in the interval instantiation of $\mathcal{V}_{\mathcal{B}}^\#$. Verification employs two complementary checks, applied together where possible:

- The abstract interpreter computes the abstract value at the program point of interest, and we verify that the computed value equals the expected one under the partial order $\sqsubseteq_{\gamma_{\text{ltv}}^\#}$. This confirms that the implementation matches the formal definitions of Chapters 3 and 4.
- Where a meaningful concrete counterpart exists, we execute the R code in the R interpreter and verify that the concrete result v satisfies $v \in \gamma_{\text{ltv}}^\#(\nu_{\text{computed}})$. This confirms that the abstraction over-approximates actual runtime behavior.

Both checks are combined for tests involving deterministic operations on bounded-length vectors. For tests involving unbounded vectors (loops with widening, unknown-length sources), only the first check is used, since no single concrete execution can meaningfully validate an infinite abstract value. The expected abstract values are derived manually from the formal definitions, serving as an independent oracle that cross-validates both the implementation and the theory.

5.5.2 Test Corpus

The test suite consists of 61 test cases organized into six categories, each targeting a specific semantic feature. The tests are minimal and representative, designed to isolate individual operations rather than simulate real-world programs. This design choice enables precise verification: each test exercises exactly one abstract operator or composition, making it possible to pinpoint the source of any precision loss or soundness violation.

The six categories are:

- **Creation** (14 tests): scalar values, vector construction via `c()`, conditional branches and attribute manipulation.
- **Recycling on Arithmetic** (9 tests): vector addition with recycling, mismatched lengths, NA propagation and uncertain branching contexts.
- **Select** (12 tests): subset selection with positive, negative and Boolean indices, double-bracket extraction and out-of-bounds access.
- **Update** (12 tests): assignment with positive, negative and Boolean indices, NA in selectors and out-of-bounds extension.
- **Uncertain Vectors** (8 tests): unbounded selectors, top source vectors and the interaction between the summary field and subsequent operations.
- **Loops** (6 tests): while loops with growing vectors via concatenation, fixpoint computation with widening and nested loops.

In the following table, the symbol $*$ denotes a statically unknown value which evaluates to $\top_{\gamma_{\text{ltv}}^\#}$.

Code Snippet	Expected Abstract Value
<pre> r ← if (*) { c(1, NA, 3) } else { c(0, NA, 5, NA) } </pre>	$r := ([3, 4], \langle [0, 1] \alpha_{\text{ltv}}(\text{NA}) [3, 5] \alpha_{\text{ltv}}(\text{NA}) \rangle, \perp_{\text{ltv}})$
<pre> v ← 1 if (*) { v ← c(1, 2, 3) } else { v ← c(0, 1) } r ← v + c(9, 10, 11) </pre>	$r := ([3, 3], \langle [9, 10] [11, 12] [11, 14] \rangle, \perp_{\text{ltv}})$
<pre> v ← c(1, 2, 3, 4, 5) r ← v[c(1, 3)] </pre>	$r := ([2, 2], \langle [1, 1] [3, 3] \rangle, \perp_{\text{ltv}})$
<pre> r ← c(1, 2, 3, 4, 5) r[10] ← 42 </pre>	$r := ([10, 10], \langle [1, 1] \dots [5, 5] \alpha_{\text{ltv}}(\text{NA}) \dots \alpha_{\text{ltv}}(\text{NA}) [42, 42] \rangle, \perp_{\text{ltv}})$
<pre> v ← * s ← c(1, 2, 3) x ← s[v] y ← x[c(1, 2)] </pre>	$x := ([0, +\infty], \langle \rangle, [1, 3] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}))$ $y := ([2, 2], \langle [1, 3] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}) \dots [1, 3] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}) \rangle, \perp_{\text{ltv}})$
<pre> x ← c(1, 2, 3) i ← 1 while (*) { x ← c(x, i) i ← i + 1 } r ← x </pre>	$r := ([3, +\infty], \langle [1, 1] [2, 2] [3, 3] \rangle, [1, +\infty] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}))$

Table 6: Representative test cases from the evaluation corpus.

5.5.3 Results and Discussion

All 61 test cases pass their verification checks. For the 47 tests on bounded vectors, the abstract interpreter computes the expected abstract values, and the concretizations of those abstract values contain all observed concrete results. For the 14 tests on unbounded vectors, only the abstract-value check is applied, since no single concrete execution can meaningfully validate an infinite abstract value. We discuss the precision characteristics of each category.

Creation. The analysis tracks exact lengths and values for literal vectors. In conditional construction (Test 1), the join of the two branches produces an abstract vector with length $[3, 4]$ and known content that preserves the fact that position 1 is in $[0, 1]$, position 2 is definitely NA, and positions 3 and 4 (the latter only present in the else branch) are $[3, 5]$ and NA respectively.

Recycling. The analysis successfully handles the interaction between control-flow uncertainty and recycling. In Test 2, the vector being added has length 3, while the operand length is uncertain (2 or 3 depending on the branch). The recycling operator $\text{Recycle}^\#$ produces known content whose entries are intervals that capture all possible

recycling outcomes, yielding precise bounds $[9, 10]$, $[11, 12]$, $[11, 14]$ for the three result positions.

Select. Positional selection with concrete indices preserves the exact values of the selected positions. In Test 3, selecting positions 1 and 3 from a vector of five known values yields a vector of length $[2, 2]$ with the known content $\langle [1, 1], [3, 3] \rangle$, exactly as expected. The abstract positive selection operator of Section 4.6.1 correctly filters the known content.

Section 4.6.1,

66

Update. Out-of-bounds update extends the vector and fills intermediate positions with NA. In Test 4, assigning to position 10 of a vector of length 5 yields a vector of length $[10, 10]$. The known content tracks positions 1 through 5 precisely, the intermediate positions 6 through 9 as $\alpha_{\text{ltv}}(\text{NA})$, and position 10 as $[42, 42]$. The `InitKnown` helper correctly initializes the newly created positions.

Uncertain Vectors. The summary field is essential here because the unknown selector forces all position-level tracking to collapse. In Test 5, a vector s of known values $[1, 2, 3]$ is selected by an unknown selector v (modeled as $\top_{\mathcal{V}_{\text{ltv}}^{\#}}$). Because the selector is unknown, no individual positions of the result can be tracked: the known content is empty and the summary field captures that all elements come from the range $[1, 3]$ with possible NA from out-of-bounds accesses. When a subsequent selection extracts the first two positions, the summary field provides the information that each extracted position lies in the interval $[1, 3]$ with possible NA, restoring a finite vector of length $[2, 2]$. This shows that the summary field preserves information even when enumeration is impossible, enabling downstream operations to recover precision.

Loops. The while loop in Test 6 repeatedly concatenates a counter to an accumulating vector. The widening operator $\nabla_{\mathcal{V}_{\text{ltv}}^{\#}}$ ensures termination by stabilizing the length to $[3, +\infty]$ and the summary to $[1, +\infty]$ (joined with $\alpha_{\text{ltv}}(\text{NA})$). The known content preserves the three positions that existed before the loop entry: $[1, 1]$, $[2, 2]$, $[3, 3]$. The summary field captures all subsequent growth, since positions beyond the third, whose values are the loop counter i , range from 1 to arbitrarily large integers.

5.5.4 Limitations

The validation demonstrates the feasibility and correctness of the abstract domain but has several limitations.

First, the test suite is minimal and representative, not a collection of real-world programs. This is by design: our goal is to verify the abstract operators individually, and real programs combine multiple semantic features, making it difficult to isolate the cause of precision loss.

Second, the evaluation is confined to μR , the core calculus introduced in Section 1.6. The analysis does not handle closures, environments, non-standard evaluation, S3/S4 dispatch, or the library

ecosystem. Extending the evaluation to full R programs requires the extensions outlined in Chapter 6.

↻ Chapter 6,

90

Third, the evaluation is limited to the interval instantiation of $\mathcal{B}^\sharp$. While the parametric design supports other instantiations, we have not experimentally evaluated them. A comparison with, for instance, a sign-domain instantiation could illuminate the precision-cost trade-off inherent in the parametric design, but this is left as future work.

Conclusions

1	Summary of Results,	90
2	Limitations,	91
3	Future Work,	92

This thesis developed a parametric abstract domain for static reasoning about R vectors, the first of its kind to combine a complete lattice structure, formal soundness proofs, and a working implementation. This chapter summarizes the findings, assesses the four contributions enumerated in Chapter 1, discusses the limitations of the work, and outlines directions for future investigation.

6.1 Summary of Results

This thesis developed a parametric abstract domain for static reasoning about R vectors. We defined a core calculus, constructed a complete lattice of abstract vectors, and lifted every calculus operator to an abstract counterpart with soundness proofs for recycling and positive selection. A proof-of-concept implementation then validated the entire framework. The domain is the first of its kind: it provides a formal lattice structure, a widening operator that ensures termination of fixpoint iteration, soundness proofs for recycling and positive selection, and a working implementation whose results have been checked against concrete R semantics. We now summarize the four contributions and confirm their delivery.

Chapter 2 defined μR , a minimal core calculus capturing the essential vector operations of the language: construction, concatenation, type coercion, recycling, selection, and update. The calculus provides a clean formal concrete semantics that serves as the reference for all soundness proofs in the thesis. μR isolates vector-level semantics from R's other complexities: lazy evaluation [14], first-class environments [16], and metaprogramming. This makes the abstraction problem tractable.

Chapter 3 presented the abstract vector domain $\mathcal{V}_B^\sharp$, a complete lattice that tracks four dimensions of a vector simultaneously. These dimensions are: its length as an integer interval, a known content sequence of individually tracked element values, a summary approximation for unbounded positions, and the attribute metadata attached to the vector. The domain is parametric in the element abstract domain $\mathcal{B}^\sharp$: any lattice that provides arithmetic, relational, and logical operations over scalar values can be plugged in, allowing the analysis to be tuned for different precision requirements. We proved three properties of the construction. The quotient under the equivalence relation on abstract vectors yields a partial order. The domain forms a complete lattice with explicit join, meet, and widening operators. And a

μR CALCULUS

[14] Goel et al., 2019
On the Design,
Implementation, and Use of
Laziness in R,

[16] Flückiger et al., 2019
R Melts Brains: An IR for
First-Class Environments and
Lazy Effectful Arguments,

PARAMETRIC ABSTRACT DOMAIN

formal Galois connection links $\mathcal{V}_B^\sharp$ to the concrete powerset domain of R vectors. The widening operator $\nabla_{\mathcal{V}_B^\sharp}$ guarantees that fixpoint iteration in the analysis of while loops terminates, a non-trivial property given that the concrete domain admits infinite ascending chains. Theorem 3.8.2 establishes the Galois connection and its soundness. The Galois connection provides the framework for proving any abstract operator sound, once its concrete counterpart is shown to respect the concretization function.

□ Theorem, 54

Chapter 4 lifted every μR operator to an abstract counterpart over $\mathcal{V}_B^\sharp$. For each concrete operator defined in Chapter 2, we defined an abstract version. The operators covered include coercion, attribute manipulation, concatenation, recycling, and selection and update through positive, negative, and logical indices. Full soundness proofs are provided for the recycling operator (Theorem 4.4.1) and the positive selection operator (Theorem 4.6.1), both in the chapter body and in Appendix B. The recycling proof proceeds through four auxiliary lemmas covering padding, cycling, join preservation, and summary coverage, then assembles them into the main result. The remaining operators, negative selection, logical selection, and all three update modes, have their abstract semantics defined and followed in the implementation, but their formal soundness proofs are left to future work.

SOUND ABSTRACT OPERATORS

⊗ Chapter 2, 25

□ Theorem, 61

□ Theorem, 67

Chapter 5 realized the domain as a proof-of-concept on `flowR`. Across 61 handcrafted μR test programs, all applicable verification checks passed: the abstract interpreter produces the expected abstract values in every case, and concretizations soundly contain the R interpreter output for all bounded tests.

IMPLEMENTATION AND VALIDATION

To our knowledge, this is the first parametric abstract domain for R vectors with a complete lattice structure, a widening operator for termination, formal soundness proofs for recycling and positive selection, and a working implementation validated against concrete R semantics. The thesis demonstrates that abstract interpretation can bring precise, computable reasoning to a language whose vector semantics have long resisted static analysis. Where prior work on R has focused on optimizing interpreters, runtime type feedback, or unsound linting heuristics, this thesis provides a foundation for sound static reasoning about the language's central data structure.

POSITIONING

6.2 Limitations

The work presented here is a first step, and several limitations define the agenda for the next phase of research. We state them explicitly, as design boundaries rather than failures, so that future work can target them directly.

The calculus intentionally excludes closures, environments, lazy evaluation, non-standard evaluation through `eval()` and `substitute()`, and S3/S4 generic function dispatch. Extending the fragment of analyzable R to cover more language features is engineering work; the vector domain itself does not need restructuring.

μR SCOPE

Setting any non-empty attribute collapses the attribute component to its top element, so the domain cannot distinguish a plain numeric vector from a data frame, a matrix, or a named vector. This precludes modeling S3 method dispatch. It also means name-based access is unsupported: `df$column` and `x[“name”]` require tracking a mapping from character strings to positions, which the current attribute component does not provide. The domain handles positional indexing only.

ATTRIBUTES AND
NAMED ACCESS

The test suite systematically covers all six categories of vector operations in μR , but it does not exercise CRAN packages, data-science pipelines, or any codebase of realistic scale. Scaling to real R code would require a configurable domain that accounts for the pervasive operator redefinitions in the R package ecosystem: via S3 dispatch, packages routinely override base operators like `+`, `[`, and `print`, and many define custom coercions and class hierarchies that alter the semantics of vector operations. Developing such a configuration mechanism, together with function summaries for the library ecosystem, is a substantial engineering task orthogonal to the core domain design.

TEST COVERAGE

The test suite is small enough that analysis time is negligible. Scaling to programs with thousands of lines of code would require measuring analysis time, memory consumption, and the behavior of the widening operator under large iteration counts. Such evaluation is premature before library summaries enable analysis of real programs, but it remains a necessary step toward production use.

PERFORMANCE

Formal soundness proofs are complete for the recycling operator and the positive selection operator. The abstract semantics of negative selection, logical selection, and all three update modes (positive, negative, logical) are defined in Chapter 4 and followed by the implementation, but their formal connection to the concrete semantics via the Galois connection remains to be established.

PARTIAL SOUNDNESS
PROOFS

☞ Chapter 4,

55

6.3 Future Work

Each limitation opens a research thread. The following are the most immediate and consequential.

The highest-priority extension is a refined attribute domain that tracks `class`, `dim`, and `names` individually instead of collapsing all attributes to a single top element. This would enable S3 dispatch modeling, name-based access, and data frame analysis, the feature most needed for non-trivial R programs. S4 dispatch, with its formal class hierarchies and multiple dispatch, would also become approachable once the attribute domain tracks `class` values.

S3 ATTRIBUTES

A data frame is a vector with `class`, `dim`, and `names` attributes. Once the attribute domain distinguishes these, the vector domain provides abstract values that a data frame shape analysis can consume directly. Such an analysis already exists within `flowR`: Gerstl [41] tracks column names, column count, and row count on over 33,000 real-world R scripts. The two domains form a natural vertical stack:

DATA FRAMES

[41] Gerstl, 2025
*Tracking the Shape of Data
Frames in R Programs Using
Abstract Interpretation,*

vector domain for element-level semantics, data frame domain for structural semantics.

The proofs for recycling and positive selection establish a reusable methodology: decomposition into helper lemmas, composition through the lattice structure, and final assembly via the Galois connection. The same methodology applies directly to negative selection, logical selection, and all three update modes. Completing these proofs is mechanical work following the pattern demonstrated in Appendix B.

COMPLETING
SOUNDNESS PROOFS

Widening the calculus to include closures, basic environments, lazy evaluation, and a larger fragment of R's surface syntax would broaden the class of analyzable programs without restructuring the vector domain. The domain's operators are defined per-vector; extending the expression and statement language that produces them is orthogonal engineering.

EXTENDED μR

Analyzing real R programs requires abstract summaries for functions in the library ecosystem: for each call, the summary maps abstract input vectors to abstract output vectors over $\mathcal{V}_B^\sharp$. Developing these summaries is primarily an engineering effort, but they form the necessary bridge between the domain's theoretical soundness and its practical applicability.

LIBRARY SUMMARIES

The domain and its proofs show that abstract interpretation can bring formal, sound reasoning to R vector semantics, a problem that has resisted static analysis for decades. The work ahead is substantial but clearly scoped: each limitation maps to a concrete extension, and the theoretical foundation is in place.

Formal Proofs: Domain Structure

This appendix contains the full proofs for the theorems stated in Chapter 3. The proofs establish the validity of the equivalence-class construction, the lattice structure of the abstract domain, and the soundness of the concretization function.

Chapter 3,

41

A.1 Equivalence Relation and Minimum Representative

Proof

of Lemma 3.4.1

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$, $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$, and $\nu_3 = (L_3, \mathbf{p}_3, s_3, a_3)$ with $k_i = |\mathbf{p}_i|$ and $\mathbf{p}_i = \langle p_{i,1} \dots p_{i,k_i} \rangle$.

Reflexivity. For any $\nu = (L, \mathbf{p}, s, a)$, we have $L = L$, $s = s$, $a = a$, and $k = k$ with $\forall i \in [1, k] : p_i = p_i$. Hence $\nu \equiv \nu$.

Symmetry. If $\nu_1 \equiv \nu_2$, then $L_1 = L_2$, $s_1 = s_2$, $a_1 = a_2$, and the known contents agree. The three cases are symmetric: if $k_1 = k_2$ the condition is identical in both directions; if $k_1 > k_2$ the condition for $\nu_2 \equiv \nu_1$ is exactly the $k_2 < k_1$ case of the definition. Hence $\nu_2 \equiv \nu_1$.

Transitivity. Suppose $\nu_1 \equiv \nu_2$ and $\nu_2 \equiv \nu_3$. Then $L_1 = L_2 = L_3$, $s_1 = s_2 = s_3$, and $a_1 = a_2 = a_3$. We write s for the common summary. It remains to show that $\mathbf{p}_1$ and $\mathbf{p}_3$ agree. The key case is when $k_1 < k_2 < k_3$: from $\nu_1 \equiv \nu_2$ we have $\forall i \in [1, k_1] : p_{1,i} = p_{2,i}$ and $\forall j \in [k_1 + 1, k_2] : p_{2,j} = s$; from $\nu_2 \equiv \nu_3$ we have $\forall i \in [1, k_2] : p_{2,i} = p_{3,i}$ and $\forall j \in [k_2 + 1, k_3] : p_{3,j} = s$. Combining these, $\forall i \in [1, k_1] : p_{1,i} = p_{3,i}$ and $\forall j \in [k_1 + 1, k_3] : p_{3,j} = s$, which is exactly the $k_1 < k_3$ case. The other combinations of relative lengths are similar or trivial. Hence $\nu_1 \equiv \nu_3$.

□

Lemma 1.1.1

Minimum Representative

For every $\nu \in \tilde{\mathcal{V}}_B^\sharp$ with $\nu = (L, \mathbf{p}, s, a)$ and $k = |\mathbf{p}|$:

1. $\nu^\downarrow \in \tilde{\mathcal{V}}_B^\sharp$;
2. $\nu^\downarrow \equiv \nu$;
3. if $\nu' \equiv \nu$ then $\nu'^\downarrow = \nu^\downarrow$;
4. $\gamma_{\mathcal{V}_B^\sharp}(\nu^\downarrow) = \gamma_{\mathcal{V}_B^\sharp}(\nu)$.

Proof

of Lemma 1.1.1

Recall that, given $\mathcal{K} = \{i \in [0, k] \mid \forall j > i : p_j = s\} \cup \{k\}$, then $k^\downarrow \stackrel{\text{def}}{=} \max\{\min \mathcal{K}, l\}$, and $\nu^\downarrow = (L, \langle p_1 \cdots p_{k^\downarrow} \rangle, s, a)$.

1. By construction $l \leq k^\downarrow \leq k$. Since $k \leq u$, we have $l \leq k^\downarrow \leq u$. If $u \neq +\infty$, then $s = \perp_{\mathcal{B}^\#}$ and $k = u$, so no position equals s and $k^\downarrow = k = u$, preserving the finite-vector invariant. Hence $\nu^\downarrow \in \tilde{\mathcal{V}}_{\mathcal{B}}^\#$.
2. The tuple $\nu^\downarrow$ has the same L , s , and a as ν , and its known content is a prefix of $\mathbf{p}$ of length $k^\downarrow$. All truncated positions $j \in [k^\downarrow + 1, k]$ satisfy $p_j = s$ by definition of $k^\downarrow$. Hence $\nu^\downarrow \equiv \nu$.
3. Let $\nu' = (L, \mathbf{p}', s, a)$ with $k' = |\mathbf{p}'|$ and $\nu' \equiv \nu$. Then $\mathbf{p}'$ agrees with $\mathbf{p}$ on all positions up to $\min(k, k')$, and any padding beyond the shorter length equals s . The truncation point $k'^\downarrow$ is computed identically: positions beyond $\min(k, k')$ that equal s are removed in both cases, and positions within the common prefix are the same. Hence $\nu'^\downarrow = \nu^\downarrow$.
4. By construction, ν and $\nu^\downarrow$ share known positions in $[1, k^\downarrow]$. Clearly $\forall i \in [1, k^\downarrow] : \gamma_{\mathcal{B}^\#}(p_i^\downarrow) = \gamma_{\mathcal{B}^\#}(p_i)$. By definition, $\forall i \in [k^\downarrow + 1, k] : p_i = s$, that implies $\forall i \in [k^\downarrow + 1, k] : \gamma_{\mathcal{B}^\#}(p_i) = \gamma_{\mathcal{B}^\#}(s)$. It naturally follows that $\gamma_{\mathcal{V}_{\mathcal{B}}^\#}(\nu^\downarrow) = \gamma_{\mathcal{V}_{\mathcal{B}}^\#}(\nu)$.

□

A.2 Partial Order

Proof

of Theorem 3.5.1

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$, $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$, and $\nu_3 = (L_3, \mathbf{p}_3, s_3, a_3)$ with $k_i = |\mathbf{p}_i|$ and $\mathbf{p}_i = \langle p_{i,1} \cdots p_{i,k_i} \rangle$.

Reflexivity. For any $\nu = (L, \mathbf{p}, s, a)$, we have $L \sqsubseteq_{\text{Itv}_0} L$, $s \sqsubseteq_{\mathcal{B}^\#} s$, and $a \sqsubseteq_{\text{Attr}^\#} a$. For all $i \in [1, k]$, we have $p_i \sqsubseteq_{\mathcal{B}^\#} p_i$. Hence $\nu \sqsubseteq_{\mathcal{V}_{\mathcal{B}}^\#} \nu$.

Anti-Symmetry. Suppose $\nu_1 \sqsubseteq_{\mathcal{V}_{\mathcal{B}}^\#} \nu_2$ and $\nu_2 \sqsubseteq_{\mathcal{V}_{\mathcal{B}}^\#} \nu_1$. Then $L_1 = L_2$ (by anti-symmetry of $\sqsubseteq_{\text{Itv}_0}$), $s_1 = s_2$ (by anti-symmetry of $\sqsubseteq_{\mathcal{B}^\#}$), and $a_1 = a_2$. We distinguish two cases based on the finiteness of the length:

If the length is finite, then $k_1 = k_2 = u$ (where u is the upper bound of the length interval). The first case of the order definition applies for both directions, so for all $i \in [1, k_1]$: $p_{1,i} \sqsubseteq_{\mathcal{B}^\#} p_{2,i}$ and $p_{2,i} \sqsubseteq_{\mathcal{B}^\#} p_{1,i}$, implying $p_{1,i} = p_{2,i}$. Hence $\mathbf{p}_1 = \mathbf{p}_2$.

If the length is infinite ($u = +\infty$), we have potentially $k_1 \neq k_2$. Without loss of generality, assume $k_1 \leq k_2$. From $\nu_1 \sqsubseteq_{\mathcal{V}_{\mathcal{B}}^\#} \nu_2$, the second case in the definition of K applies and we have $\forall i \in [1, k_1] : p_{1,i} \sqsubseteq_{\mathcal{B}^\#} p_{2,i}$ and $\forall j \in [k_1 + 1, k_2] : s \sqsubseteq_{\mathcal{B}^\#} p_{2,j}$. From $\nu_2 \sqsubseteq_{\mathcal{V}_{\mathcal{B}}^\#} \nu_1$, the third case of the definition of K applies and we have $\forall i \in [1, k_1] : p_{2,i} \sqsubseteq_{\mathcal{B}^\#} p_{1,i}$ and $\forall j \in [k_1 + 1, k_2] : p_{2,j} \sqsubseteq_{\mathcal{B}^\#} s$. This implies $\forall i \in [1, k_1] : p_{1,i} = p_{2,i}$ and $\forall j \in [k_1 + 1, k_2] : p_{2,j} = s$. Since ν_2 is a minimum representative and all positions $j \in [k_1 + 1, k_2]$ equal the summary s_2 , these trailing positions would be removed by minimization, implying $k_2 \leq k_1$.

Combined with $k_1 \leq k_2$, we conclude $k_1 = k_2$ and $[k_1 + 1, k_2] = \emptyset$. The case $k_2 < k_1$ is analogous.

In both cases $p_1 = p_2$, hence $\nu_1 = \nu_2$.

Transitivity. Suppose $\nu_1 \sqsubseteq_{\mathcal{V}_B^\#} \nu_2$ and $\nu_2 \sqsubseteq_{\mathcal{V}_B^\#} \nu_3$. Then $L_1 \sqsubseteq_{\text{Itv}_0} L_3$, $s_1 \sqsubseteq_{\mathcal{B}^\#} s_3$, and $a_1 \sqsubseteq_{\text{Attr}^\#} a_3$. For each position $i \in [1, \min(k_1, k_3)]$, we distinguish two cases based on the relative position with respect to k_2 :

If $i \leq k_2$, then by transitivity of $\sqsubseteq_{\mathcal{B}^\#}$ we have $p_{1,i} \sqsubseteq_{\mathcal{B}^\#} p_{2,i} \sqsubseteq_{\mathcal{B}^\#} p_{3,i}$, which implies $p_{1,i} \sqsubseteq_{\mathcal{B}^\#} p_{3,i}$.

If $i > k_2$, since $\nu_1 \sqsubseteq_{\mathcal{V}_B^\#} \nu_2$ we are in the third case of the definition of K and $p_{1,i} \sqsubseteq_{\mathcal{B}^\#} s_2$; from $\nu_2 \sqsubseteq_{\mathcal{V}_B^\#} \nu_3$, we are in the second case and $s_2 \sqsubseteq_{\mathcal{B}^\#} p_{3,i}$. By transitivity of $\sqsubseteq_{\mathcal{B}^\#}$, $p_{1,i} \sqsubseteq_{\mathcal{B}^\#} p_{3,i}$.

In both cases $p_{1,i} \sqsubseteq_{\mathcal{B}^\#} p_{3,i}$ for all $i \in [1, \min(k_1, k_3)]$, establishing the common content condition. It remains to verify the conditions for positions beyond $\min(k_1, k_3)$. The definition of K imposes additional constraints only when the shorter vector has unknown length ($u_1 = +\infty$ if $k_1 \leq k_3$, or $u_3 = +\infty$ if $k_1 > k_3$); in all other cases the common content condition is sufficient. We thus distinguish the two non-trivial cases:

If $k_1 \leq k_3$ and $u_1 = +\infty$: for $j \in [k_1 + 1, k_3]$, if $j \leq k_2$, then from $\nu_1 \sqsubseteq_{\mathcal{V}_B^\#} \nu_2$ we have $s_1 \sqsubseteq_{\mathcal{B}^\#} p_{2,j}$ while from $\nu_2 \sqsubseteq_{\mathcal{V}_B^\#} \nu_3$ we have $p_{2,j} \sqsubseteq_{\mathcal{B}^\#} p_{3,j}$. Otherwise, if $j > k_2$, then from $\nu_1 \sqsubseteq_{\mathcal{V}_B^\#} \nu_2$ we have $s_1 \sqsubseteq_{\mathcal{B}^\#} s_2$, while from $\nu_2 \sqsubseteq_{\mathcal{V}_B^\#} \nu_3$ we have $s_2 \sqsubseteq_{\mathcal{B}^\#} p_{3,j}$.

If $k_1 > k_3$ and $u_3 = +\infty$: for $j \in [k_3 + 1, k_1]$, if $j \leq k_2$ then, as in the previous case, $p_{1,j} \sqsubseteq_{\mathcal{B}^\#} s_2 \sqsubseteq_{\mathcal{B}^\#} s_3$; if $j > k_2$ then $p_{1,j} \sqsubseteq_{\mathcal{B}^\#} s_2 \sqsubseteq_{\mathcal{B}^\#} s_3$.

In all cases the conditions of $K(\nu_1, \nu_3)$ are satisfied. Hence $\nu_1 \sqsubseteq_{\mathcal{V}_B^\#} \nu_3$.

□

A.3 Meet Preservation

Lemma 1.3.1

Bottom-Meet Length Bound

Let $\nu_1, \nu_2 \in \mathcal{V}_B^\#$ and define p'_i as in the meet operator. If $p'_i = \perp_{\mathcal{B}^\#}$ for some $i \geq 1$, then every vector $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$ has length $n < i$.

Proof

of Lemma 1.3.1

Let $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$ have length n . For every position $j \in [1, n]$ we have $c_j \in \gamma_{\mathcal{B}^\#}(p'_{1,j}) \cap \gamma_{\mathcal{B}^\#}(p'_{2,j}) = \gamma_{\mathcal{B}^\#}(p'_j)$ by meet-preservation of $\gamma_{\mathcal{B}^\#}$. If $i \leq n$, then $c_i \in \gamma_{\mathcal{B}^\#}(\perp_{\mathcal{B}^\#}) = \emptyset$, a contradiction. Hence $n < i$.

□

Lemma 1.3.2**Attribute Concretization is Meet-Preserving**

The attribute concretization function $\gamma_{\text{Attr}^\#}$ is meet-preserving, i.e., for all $a_1, a_2 \in \text{Attr}^\#$:

$$\gamma_{\text{Attr}^\#}(a_1 \sqcap_{\text{Attr}^\#} a_2) = \gamma_{\text{Attr}^\#}(a_1) \cap \gamma_{\text{Attr}^\#}(a_2)$$

Proof**of Lemma 1.3.2**

We distinguish cases on the shape of a_1 and a_2 .

If $a_1 = \perp_{\text{Attr}^\#}$ or $a_2 = \perp_{\text{Attr}^\#}$, then $a_1 \sqcap_{\text{Attr}^\#} a_2 = \perp_{\text{Attr}^\#}$, so $\gamma_{\text{Attr}^\#}(a_1 \sqcap_{\text{Attr}^\#} a_2) = \emptyset = \gamma_{\text{Attr}^\#}(a_1) \cap \gamma_{\text{Attr}^\#}(a_2)$.

Otherwise $a_1 = (\text{MustHave}_1, \text{MayHave}_1)$ and $a_2 = (\text{MustHave}_2, \text{MayHave}_2)$. Their concretizations are:

$$\gamma_{\text{Attr}^\#}(a_j) = \{\mathcal{A} \subseteq \text{Attr} \mid \text{MustHave}_j \subseteq \mathcal{A} \subseteq \text{MayHave}_j\}, \quad j = 1, 2$$

The intersection is:

$$\gamma_{\text{Attr}^\#}(a_1) \cap \gamma_{\text{Attr}^\#}(a_2) = \left\{ \mathcal{A} \mid \begin{array}{l} \text{MustHave}_1 \cup \text{MustHave}_2 \\ \subseteq \mathcal{A} \\ \subseteq \text{MayHave}_1 \cap \text{MayHave}_2 \end{array} \right\}$$

If $\text{MustHave}_1 \cup \text{MustHave}_2 \not\subseteq \text{MayHave}_1 \cap \text{MayHave}_2$, the set is empty and $a_1 \sqcap_{\text{Attr}^\#} a_2 = \perp_{\text{Attr}^\#}$, hence $\gamma_{\text{Attr}^\#}(a_1 \sqcap_{\text{Attr}^\#} a_2) = \emptyset$. Otherwise $a_1 \sqcap_{\text{Attr}^\#} a_2 = (\text{MustHave}_1 \cup \text{MustHave}_2, \text{MayHave}_1 \cap \text{MayHave}_2)$, and:

$$\gamma_{\text{Attr}^\#}(a_1 \sqcap_{\text{Attr}^\#} a_2) = \left\{ \mathcal{A} \mid \begin{array}{l} \text{MustHave}_1 \cup \text{MustHave}_2 \\ \subseteq \mathcal{A} \\ \subseteq \text{MayHave}_1 \cap \text{MayHave}_2 \end{array} \right\}$$

which is exactly $\gamma_{\text{Attr}^\#}(a_1) \cap \gamma_{\text{Attr}^\#}(a_2)$.

□

Proof**of Theorem 3.8.1**

Let $\gamma_{\mathcal{B}^\#}$ be the meet-preserving concretization function of the supporting value domain. By Lemma 1.3.2, $\gamma_{\text{Attr}^\#}$ is meet-preserving. Let $\nu_1 = ([l_1, u_1], \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = ([l_2, u_2], \mathbf{p}_2, s_2, a_2)$ with $k_1 = |\mathbf{p}_1|$, $k_2 = |\mathbf{p}_2|$, and $M = \max(k_1, k_2)$. Recall the auxiliary elements used by the meet operator:

△ Lemma,

97

$$p'_{1,i} = \begin{cases} p_{1,i} & \text{if } i \leq k_1 \\ s_1 & \text{if } i > k_1 \end{cases} \quad p'_{2,i} = \begin{cases} p_{2,i} & \text{if } i \leq k_2 \\ s_2 & \text{if } i > k_2 \end{cases}$$

and $p'_i = p'_{1,i} \sqcap_{\mathcal{B}^\#} p'_{2,i}$. Let:

$$u' = \max\{i \in [0, \max(l_1, l_2, M)] \mid \forall j \in [1, i] : p'_j \neq \perp_{\mathcal{B}^\#}\}$$

We show that a concrete vector $v = [c_1 \dots c_n]_{\mathcal{A}}$ belongs to $\gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$ iff it belongs to $\gamma_{\mathcal{V}_B^\#}(\nu_1 \sqcap_{\mathcal{V}_B^\#} \nu_2)$.

Empty meet. If $\max(l_1, l_2) > u'$, then $\nu_1 \sqcap_{\mathcal{V}_B^\#} \nu_2 = \perp_{\mathcal{V}_B^\#}$ and $\gamma_{\mathcal{V}_B^\#}(\perp_{\mathcal{V}_B^\#}) = \emptyset$. Any $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$ must satisfy $n \geq \max(l_1, l_2) > u'$. Thus, by the definition of the concretization function:

$$\forall i \in [1, \min(k_1, k_2)] : c_i \in \gamma_{\mathcal{B}^\#}(p_{1,i}) \cap \gamma_{\mathcal{B}^\#}(p_{2,i}) = \gamma_{\mathcal{B}^\#}(p_{1,i} \sqcap_{\mathcal{B}^\#} p_{2,i})$$

but by hypothesis $\forall j \in [u' + 1, \max(l_1, l_2)] : p_{1,j} \sqcap_{\mathcal{B}^\#} p_{2,j} = \perp_{\mathcal{B}^\#}$, so it is impossible such vector exists. Hence $\gamma_{\mathcal{V}_B^\#}(\nu_1 \sqcap_{\mathcal{V}_B^\#} \nu_2) = \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2) = \emptyset$.

Non-empty meet. Assume $\max(l_1, l_2) \leq u'$ and let $\nu' = \nu_1 \sqcap_{\mathcal{V}_B^\#} \nu_2$. If $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$, then $n \geq \max(l_1, l_2)$. Moreover, we have:

$$\forall i \in [1, n] : c_i \in \gamma_{\mathcal{B}^\#}(p'_{1,i}) \wedge c_i \in \gamma_{\mathcal{B}^\#}(p'_{2,i}) = c_i \in \gamma_{\mathcal{B}^\#}(p'_i)$$

with $p'_i \neq \perp_{\mathcal{B}^\#}$ for all positions in $[1, u']$.

We now distinguish the two shapes of ν' .

Finite meet. In the finite case $\nu' = ([\max(l_1, l_2), u'], \mathbf{p}', \perp_{\mathcal{B}^\#}, a_1 \sqcap_{\text{Attr}^\#} a_2)$ with $|\mathbf{p}'| = u'$ (i.e., $\mathbf{p}' = \langle p'_1, \dots, p'_{u'} \rangle$). By definition of u' , either $u' = M$ and $s_1 \sqcap_{\mathcal{B}^\#} s_2 = \perp_{\mathcal{B}^\#}$, or $u' < M$ and $p'_{u'+1} = \perp_{\mathcal{B}^\#}$. In both situations there exists $j \leq M + 1$ with $p'_j = \perp_{\mathcal{B}^\#}$. By Lemma 1.3.1, every vector in $\gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$ satisfies $n < j \leq u' + 1$, hence $n \leq u'$. Conversely, if $\max(l_1, l_2) \leq n \leq u'$, then $p'_i \neq \perp_{\mathcal{B}^\#}$ for all $i \in [1, n]$, and a concrete vector satisfying both operands can be built from the non-empty sets $\gamma_{\mathcal{B}^\#}(p'_i)$. Hence the length constraints of ν' are exactly $n \in [\max(l_1, l_2), u']$.

△ Lemma,

96

For the content, let $v \in \gamma_{\mathcal{V}_B^\#}(\nu')$. For each $i \in [1, n]$ we have:

$$c_i \in \gamma_{\mathcal{B}^\#}(p'_i) = \gamma_{\mathcal{B}^\#}(p'_{1,i}) \cap \gamma_{\mathcal{B}^\#}(p'_{2,i})$$

therefore v satisfies the content constraints of both ν_1 and ν_2 . For the attributes, $\mathcal{A} \in \gamma_{\text{Attr}^\#}(a_1) \cap \gamma_{\text{Attr}^\#}(a_2) = \gamma_{\text{Attr}^\#}(a_1 \sqcap_{\text{Attr}^\#} a_2)$. Thus $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$.

Conversely, let $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$. We have shown $n \in [\max(l_1, l_2), u']$. Since $|\mathbf{p}'| = u' \geq n$, every position of v is covered by $\mathbf{p}'$. For each $i \in [1, n]$:

$$c_i \in \gamma_{\mathcal{B}^\#}(p'_{1,i}) \cap \gamma_{\mathcal{B}^\#}(p'_{2,i}) = \gamma_{\mathcal{B}^\#}(p'_i)$$

which is exactly the condition required by $\gamma_{\mathcal{V}_B^\#}(\nu')$. The attribute condition is identical. Hence $v \in \gamma_{\mathcal{V}_B^\#}(\nu')$.

Infinite meet. In the infinite case $\nu' = ([\max(l_1, l_2), +\infty], \mathbf{p}', s_1 \sqcap_{\mathcal{B}^\#} s_2, a_1 \sqcap_{\text{Attr}^\#} a_2)$ with $|\mathbf{p}'| = M$ (i.e., $\mathbf{p}' = \langle p'_1, \dots, p'_M \rangle$) and $u' = M$. The feasible lengths are $n \in [\max(l_1, l_2), +\infty]$.

Let $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$. For each $i \in [1, n]$: if $i \leq M$, then:

$$c_i \in \gamma_{\mathcal{B}^\#}(p'_{1,i}) \cap \gamma_{\mathcal{B}^\#}(p'_{2,i}) = \gamma_{\mathcal{B}^\#}(p'_i)$$

which matches the known content of ν' ; if $i > M$, then $i > k_1$ and $i > k_2$, so:

$$c_i \in \gamma_{\mathcal{B}^\#}(s_1) \cap \gamma_{\mathcal{B}^\#}(s_2) = \gamma_{\mathcal{B}^\#}(s_1 \sqcap_{\mathcal{B}^\#} s_2)$$

which matches the summary of ν' . Since $u_1 = u_2 = +\infty$, the length satisfies $n \geq \max(l_1, l_2)$ with no upper bound, so $n \in [\max(l_1, l_2), +\infty]$. Hence $v \in \gamma_{\mathcal{V}_B^\#}(\nu')$.

Conversely, let $v \in \gamma_{\mathcal{V}_B^\#}(\nu')$. For each $i \in [1, n]$: if $i \leq M$, then $c_i \in \gamma_{\mathcal{B}^\#}(p'_i) \subseteq \gamma_{\mathcal{B}^\#}(p'_{1,i}) \cap \gamma_{\mathcal{B}^\#}(p'_{2,i})$; if $i > M$, then:

$$c_i \in \gamma_{\mathcal{B}^\#}(s_1 \sqcap_{\mathcal{B}^\#} s_2) = \gamma_{\mathcal{B}^\#}(s_1) \cap \gamma_{\mathcal{B}^\#}(s_2)$$

Thus v satisfies the content constraints of both ν_1 and ν_2 . Moreover $n \in [\max(l_1, l_2), +\infty] \subseteq [l_1, +\infty] \cap [l_2, +\infty]$. Therefore $v \in \gamma_{\mathcal{V}_B^\#}(\nu_1) \cap \gamma_{\mathcal{V}_B^\#}(\nu_2)$.

□

A.4 Widening

Proof

of Theorem 3.7.1

Let $\nu_1 = (L_1, \mathbf{p}_1, s_1, a_1)$ and $\nu_2 = (L_2, \mathbf{p}_2, s_2, a_2)$ with $k_i = |\mathbf{p}_i|$, $m = \min(k_1, k_2)$, $M = \max(k_1, k_2)$, and $\nu' = \nu_1 \nabla_{\mathcal{V}_B^\#} \nu_2 = ([l', u'], \mathbf{p}', s', a_1 \sqcup_{\text{Attr}^\#} a_2)$. Recall that $[l', u'] = L_1 \nabla_{\text{ltv}_0} L_2$,

$$\mathbf{p}' = \begin{cases} \langle p'_1 \dots p'_M \rangle & \text{if } u' \neq +\infty, \\ & \vee (u_1 \neq +\infty \wedge u_2 \neq +\infty) \\ \langle p_{1,1} \nabla_{\mathcal{B}^\#} p_{2,1} \dots p_{1,m} \nabla_{\mathcal{B}^\#} p_{2,m} \rangle & \text{otherwise} \end{cases}$$

with $p'_j = p_{1,j} \nabla_{\mathcal{B}^\#} p_{2,j}$ for $j \leq m$ and $p'_j = p_{i,j}$ for the longer vector i when $m < j \leq M$, and:

$$s' = \begin{cases} \perp_{\mathcal{B}^\#} & \text{if } u' \neq +\infty \\ s_1 \nabla_{\mathcal{B}^\#} \left(s_2 \sqcup_{\mathcal{B}^\#} \bigsqcup_{j=m+1}^{k_1} p_{1,j} \sqcup_{\mathcal{B}^\#} \bigsqcup_{j=m+1}^{k_2} p_{2,j} \right) & \text{otherwise} \end{cases}$$

Upper bound. We show $\nu_1 \sqsubseteq_{\mathcal{V}_B^\#} \nu'$; the argument for ν_2 is symmetric.

Length. $L_1 \sqsubseteq_{\text{ltv}_0} L_1 \nabla_{\text{ltv}_0} L_2 = L'$ by the upper-bound property of interval widening.

Attributes. $a_1 \sqsubseteq_{\text{Attr}^\#} a_1 \sqcup_{\text{Attr}^\#} a_2 = a'$ because $\sqcup_{\text{Attr}^\#}$ is a least upper bound.

Summary. If $u' \neq +\infty$, then $s' = \perp_{B^\#}$ and $s_1 \sqsubseteq_{B^\#} \perp_{B^\#}$ holds trivially. If $u' = +\infty$, then $s' = s_1 \nabla_{B^\#}(\dots)$, so $s_1 \sqsubseteq_{B^\#} s'$ by the upper-bound property of value widening.

Known content. We verify $K(\nu_1, \nu')$; the argument for ν_2 is symmetric. We distinguish the two cases of the definition of p' .

Case 1: $u' \neq +\infty \vee (u_1 \neq +\infty \wedge u_2 \neq +\infty)$. In this case $p' = \langle p'_1 \dots p'_M \rangle$ with $k' = M = \max(k_1, k_2) \geq k_1$. Note that $u' \neq +\infty$ implies $u_1 \neq +\infty$ (since interval widening with $+\infty$ yields $+\infty$), so in both disjuncts we have $u_1 \neq +\infty$. Consequently only the first case of $K(\nu_1, \nu')$ can apply ($k_1 \leq k'$ and $u_1 \neq +\infty$), and we must show $\forall j \in [1, k_1] : p_{1,j} \sqsubseteq_{B^\#} p'_j$. We further distinguish:

- If $k_1 \leq k_2$, then $m = k_1$ and $p'_j = p_{1,j} \nabla_{B^\#} p_{2,j}$ for all $j \in [1, m]$. Hence $p_{1,j} \sqsubseteq_{B^\#} p'_j$ by the upper-bound property of value widening.
- Otherwise $k_1 > k_2$, so $m = k_2$ and $M = k_1$. For $j \in [1, m]$ we have $p'_j = p_{1,j} \nabla_{B^\#} p_{2,j}$, so $p_{1,j} \sqsubseteq_{B^\#} p'_j$ by widening. For $j \in [m+1, M]$ we have $p'_j = p_{1,j}$, so $p_{1,j} \sqsubseteq_{B^\#} p_{1,j}$ trivially.

Case 2: $u' = +\infty$ and at least one of u_1, u_2 is $+\infty$. Here $k' = m = \min(k_1, k_2)$ and $u' = +\infty$. We distinguish according to the relative order of k_1 and k' .

- If $k_1 \leq k'$, then $k_1 = m = \min(k_1, k_2)$, i.e., $k_1 \leq k_2$ and $k' = k_1$. If $u_1 \neq +\infty$, the first case of $K(\nu_1, \nu')$ applies and we need $\forall j \in [1, k_1] : p_{1,j} \sqsubseteq_{B^\#} p'_j$, which holds since $p'_j = p_{1,j} \nabla_{B^\#} p_{2,j}$. If $u_1 = +\infty$, the second case of $K(\nu_1, \nu')$ applies: we need $\forall j \in [1, k_1] : p_{1,j} \sqsubseteq_{B^\#} p'_j$ (holds by widening) and $\forall j \in [k_1+1, k'] : s_1 \sqsubseteq_{B^\#} p'_j$. The latter range is empty because $k' = k_1$, so the condition holds vacuously.
- If $k_1 > k'$, then $k_1 > k_2$ and $k' = k_2 = m$. Since $u' = +\infty$, the third case of $K(\nu_1, \nu')$ applies. We need $\forall j \in [1, k'] : p_{1,j} \sqsubseteq_{B^\#} p'_j$ and $\forall j \in [k'+1, k_1] : p_{1,j} \sqsubseteq_{B^\#} s'$.

The first part holds because $p'_j = p_{1,j} \nabla_{B^\#} p_{2,j}$. For the second part, let:

$$X = s_2 \sqcup_{B^\#} \bigsqcup_{i=m+1}^{k_1} p_{1,i} \sqcup_{B^\#} \bigsqcup_{i=m+1}^{k_2} p_{2,i}$$

For any $j \in [k'+1, k_1] = [m+1, k_1]$ we have $p_{1,j} \sqsubseteq_{B^\#} \bigsqcup_{i=m+1}^{k_1} p_{1,i} \sqsubseteq_{B^\#} X$, and $X \sqsubseteq_{B^\#} s_1 \nabla_{B^\#} X = s'$ by the upper-bound property of widening. Hence $p_{1,j} \sqsubseteq_{B^\#} s'$.

Convergence. Let $(\nu_i)_{i \in \mathbb{N}}$ be an arbitrary sequence and define $x_0 = \nu_0, x_{i+1} = x_i \nabla_{\mathcal{V}_B^\#} \nu_{i+1}$. Write $x_i = (L_i, p_i, s_i, a_i)$ with $L_i = [l_i, u_i]$

and $k_i = |p_i|$. For the elements of the input sequence, write $\nu_i = (L_i^{\text{in}}, p_i^{\text{in}}, s_i^{\text{in}}, a_i^{\text{in}})$ with $L_i^{\text{in}} = [l_i^{\text{in}}, u_i^{\text{in}}]$ and $k_i^{\text{in}} = |p_i^{\text{in}}|$.

Length convergence. The length intervals L_i form a sequence defined by interval widening, which stabilises after finitely many steps by the well-foundedness of interval widening. Let $L^* = [l^*, u^*]$ be the stable length and let N_1 be such that $L_i = L^*$ for all $i \geq N_1$.

Attribute convergence. The attributes a_i form an ascending chain in the finite lattice $\text{Attr}^\sharp$, so they stabilise after at most $|\text{Attr}^\sharp|$ steps.

Known content convergence. Consider the known content lengths k_i .

If $u^* \neq +\infty$, then for all $i \geq N_1$ we have $u_{i+1}^{\text{in}} \leq u^*$ (otherwise u' would become $+\infty$). Hence every widening step falls into the first case for p' and $k_{i+1} = \max(k_i, k_{i+1}^{\text{in}}) \leq u^*$. For each position $j \in [1, u^*]$, once it appears in p_i it either stays unchanged or is widened with $\nabla_{\mathcal{B}^\sharp}$ at the next step; since value widening stabilises, each position stabilises after finitely many steps. Because there are at most u^* positions, the whole known content stabilises after finitely many steps.

If $u^* = +\infty$, let $i_0 \geq N_1$ be the first index where $u_{i_0} = +\infty$. For every $i \geq i_0$, at least one operand has infinite length, so the widening step uses the second case for p' and $k_{i+1} = \min(k_i, k_{i+1}^{\text{in}}) \leq k_i$. Thus $(k_i)_{i \geq i_0}$ is a non-increasing sequence of natural numbers and stabilises to some k^* . Once $k_i = k^*$, every subsequent input has $k_{i+1}^{\text{in}} \geq k^*$, so each position $j \in [1, k^*]$ evolves by value widening and stabilises.

Summary convergence. Once the length and the known content length have stabilised, the summary evolves only when $u^* = +\infty$. In that case, after the known content length stabilises to k^* , the summary update becomes:

$$s_{i+1} = s_i \nabla_{\mathcal{B}^\sharp} \left(s_{i+1}^{\text{in}} \sqcup_{\mathcal{B}^\sharp} \bigsqcup_{j=k^*+1}^{k_{i+1}^{\text{in}}} p_{(i+1),j}^{\text{in}} \right)$$

which is a value-widening sequence and therefore stabilises by assumption.

Since all components stabilise after finitely many steps, the sequence (x_i) stabilises.

□

Formal Proofs: Abstract Operators

This appendix contains the complete proofs for the soundness theorems stated in Chapter 4. The proof sketches in the chapter body reference these full proofs.

Chapter 4,

55

B.1 Soundness of Abstract Recycling

The soundness of $\text{Recycle}^\sharp$ (Theorem 4.4.1) is established through four auxiliary lemmas. The lemmas are stated and proved here, then composed in the main proof.

Theorem,

61

Lemma 2.1.1

Padded Covers

Let $\nu = ([l, u], \mathbf{p}, s, a) \in \mathcal{V}_B^\sharp$ with $|\mathbf{p}| = k$, and let $x \in \gamma_{\mathcal{V}_B^\sharp}(\nu)$ with $|x| = n$. Define the padded sequence $\mathbf{p}^n$ of length n by setting $\mathbf{p}_j^n = p_j$ when $j \leq k$ and $\mathbf{p}_j^n = s$ when $j > k$.

Then for all $j \in [1, n]$: $x_j \in \gamma_{B^\sharp}(\mathbf{p}_j^n)$.

Proof

By definition of $\gamma_{\mathcal{V}_B^\sharp}$ (Definition 3.8.1), for each $j \in [1, n]$: if $j \leq k$, then $x_j \in \gamma_{B^\sharp}(\mathbf{p}_j) = \gamma_{B^\sharp}(\mathbf{p}_j^n)$; if $j > k$, then $x_j \in \gamma_{B^\sharp}(s) = \gamma_{B^\sharp}(\mathbf{p}_j^n)$. Both cases follow directly from the definition of the concretization function.

:= Definition,

53

□

Lemma 2.1.2

Cycling Soundness

Let $\mathbf{p}$ be a sequence of abstract values with $|\mathbf{p}| \geq n$, and let x be a concrete vector with $|x| = n$ such that for all $j \in [1, n]$: $x_j \in \gamma_{B^\sharp}(\mathbf{p}_j)$. For any target length $h \geq 1$, let $x_r = \rho_c(x, h)$. Then for all $j \in [1, h]$: $x_{r,j} \in \gamma_{B^\sharp}(\rho_c^\sharp(\mathbf{p}, n, h)_j)$.

Proof

By definition, $\rho_c(x, h)$ concatenates $\lfloor \frac{h}{n} \rfloor$ full copies of x followed by the first $h \bmod n$ elements. The abstract cycling $\rho_c^\sharp(\mathbf{p}, n, h)$ mirrors this structure: position j of the result contains $\mathbf{p}_{((j-1) \bmod n)+1}$.

For any $j \in [1, h]$, let $i = ((j-1) \bmod n) + 1 \in [1, n]$. Then $x_{r,j} = x_i$ by the definition of ρ_c , and $\rho_c^\#(\mathbf{p}, n, h)_j = \mathbf{p}_i$ by the positional definition of $\rho_c^\#$. By hypothesis, $x_i \in \gamma_{\mathcal{B}^\#}(\mathbf{p}_i)$. Hence $x_{r,j} \in \gamma_{\mathcal{B}^\#}(\rho_c^\#(\mathbf{p}, n, h)_j)$.

□

Lemma 2.1.3**Join Soundness**

Let I be a finite non-empty set of indices and for each $i \in I$ let $\mathbf{p}_i \in \mathcal{B}^\#$. Let $\mathbf{p} = \bigsqcup_{i \in I} \mathcal{B}^\# \mathbf{p}_i$ be their pointwise join. Let x be a concrete vector such that for some $i_0 \in I$ with $|\mathbf{p}_{i_0}| \geq |x|$, we have $x_j \in \gamma_{\mathcal{B}^\#}(\mathbf{p}_{i_0,j})$ for all $j \in [1, |x|]$. Then for all $j \in [1, |x|]$: $x_j \in \gamma_{\mathcal{B}^\#}(\mathbf{p}_j)$.

Proof

For any $A, B \in \mathcal{B}^\#$, join is an upper bound: $A \sqsubseteq_{\mathcal{B}^\#} A \sqcup_{\mathcal{B}^\#} B$ and $B \sqsubseteq_{\mathcal{B}^\#} A \sqcup_{\mathcal{B}^\#} B$. By monotonicity of $\gamma_{\mathcal{B}^\#}$ (from the Galois connection, Theorem 3.8.2), we have $\gamma_{\mathcal{B}^\#}(A \sqcup_{\mathcal{B}^\#} B) \supseteq \gamma_{\mathcal{B}^\#}(A) \cup \gamma_{\mathcal{B}^\#}(B)$.

□ Theorem,

54

The pointwise join $\mathbf{p}$ is defined as $\mathbf{p}_j = \bigsqcup_{i \in I} \mathcal{B}^\# \mathbf{p}_{i,j}$ for each position j . By induction on $|I|$, for each position j :

$$\gamma_{\mathcal{B}^\#}(\mathbf{p}_j) \supseteq \bigsqcup_{i \in I} \gamma_{\mathcal{B}^\#}(\mathbf{p}_{i,j})$$

In particular, $\gamma_{\mathcal{B}^\#}(\mathbf{p}_j) \supseteq \gamma_{\mathcal{B}^\#}(\mathbf{p}_{i_0,j})$. Since $x_j \in \gamma_{\mathcal{B}^\#}(\mathbf{p}_{i_0,j})$ by hypothesis, we conclude $x_j \in \gamma_{\mathcal{B}^\#}(\mathbf{p}_j)$ for all $j \in [1, |x|]$.

□

Lemma 2.1.4**Summary Soundness**

Let $\nu = ([l, +\infty], \mathbf{p}, s, a) \in \mathcal{V}_{\mathcal{B}^\#}^\#$ with $|\mathbf{p}| = k$, and let $s' = s \sqcup_{\mathcal{B}^\#} \bigsqcup_{j=1}^k \mathcal{B}^\# \mathbf{p}_j$. For any concrete vector $x \in \gamma_{\mathcal{V}_{\mathcal{B}^\#}^\#}(\nu)$ and any target length $h \geq 1$, every position of the recycled vector $\rho_c(x, h)$ belongs to $\gamma_{\mathcal{B}^\#}(s')$.

Proof

By definition, $\rho_c(x, h)_j = x_{((j-1) \bmod |x|) + 1}$. Since $x \in \gamma_{\mathcal{V}_{\mathcal{B}^\#}^\#}(\nu)$, each position of x falls into one of two categories:

- If the position index $j' \leq k$, then $x_{j'} \in \gamma_{\mathcal{B}^\#}(\mathbf{p}_{j'})$.
- If $j' > k$, then $x_{j'} \in \gamma_{\mathcal{B}^\#}(s)$.

By construction, $s' = s \sqcup_{\mathcal{B}^\#} \bigsqcup_{j=1}^k \mathcal{B}^\# \mathbf{p}_j$ is an upper bound:

$$s \sqsubseteq_{\mathcal{B}^\#} s' \quad \wedge \quad \mathbf{p}_j \sqsubseteq_{\mathcal{B}^\#} s'$$

for all $j \in [1, k]$

By monotonicity of $\gamma_{\mathcal{B}^\#}$, we have $\gamma_{\mathcal{B}^\#}(s) \subset \gamma_{\mathcal{B}^\#}(s')$ and $\gamma_{\mathcal{B}^\#}(\mathbf{p}_j) \subset \gamma_{\mathcal{B}^\#}(s')$ for all $j \in [1, k]$. Therefore, for every index j , the value $x_{r,j} = x_{((j-1) \bmod |x|)+1}$ belongs to $\gamma_{\mathcal{B}^\#}(s')$. This covers all positions of $\rho_c(x, h)$ uniformly, including tail positions beyond the known content of ν'_1 .

□

Proof**of Theorem 4.4.1**

Let $\nu_1 = ([l_1, u_1], \mathbf{p}_1, s_1, a_1)$ with $|\mathbf{p}_1| = m$, and $\nu_2 = ([l_2, u_2], \mathbf{p}_2, s_2, a_2)$ with $|\mathbf{p}_2| = n$. Let the result of abstract recycling be:

$$\text{Recycle}^\#(\nu_1, \nu_2) = (\nu'_1, \nu'_2)$$

with $\nu'_1 = ([l', u'], \mathbf{p}'_1, s'_1, a'_1)$ and $\nu'_2 = ([l', u'], \mathbf{p}'_2, s'_2, a'_2)$ where $[l', u'] = [\max(l_1, l_2), \max(u_1, u_2)]$.

Let $x \in \gamma_{\mathcal{B}^\#}(\nu_1)$ with $|x| \in [l_1, u_1]$ and $y \in \gamma_{\mathcal{B}^\#}(\nu_2)$ with $|y| \in [l_2, u_2]$. Let $d = \max(|x|, |y|)$ and $(x_r, y_r) = \text{Recycle}(x, y)$, so $x_r = \rho_c(x, d)$. By symmetry it suffices to show $x_r \in \gamma_{\mathcal{B}^\#}(\nu'_1)$.

Length. We have $d = \max(|x|, |y|)$. Since $|x| \in [l_1, u_1]$ and $|y| \in [l_2, u_2]$, we have $|x| \geq l_1 \geq l' = \max(l_1, l_2)$ and $|y| \geq l_2 \geq l'$. Hence $d \geq l'$. Similarly, $d \leq \max(u_1, u_2) = u'$. Thus $d \in [l', u']$, satisfying the length constraint of $\gamma_{\mathcal{B}^\#}(\nu'_1)$.

Known content. We distinguish two cases based on whether ν_1 is finite ($s_1 = \perp_{\mathcal{B}^\#}$) or infinite ($s_1 \neq \perp_{\mathcal{B}^\#}$).

- **Finite case** ($s_1 = \perp_{\mathcal{B}^\#}$). Then $u_1 \neq +\infty$, so $|x| \leq u_1 \leq m$ (since for finite vectors $u_1 = m$ by the domain invariant). The known content of ν'_1 is:

$$\mathbf{p}'_1 = \bigsqcup_{d=l'}^{\max(m,n)} \bigsqcup_{\mathcal{B}^\#} \rho_f^\#(\mathbf{p}_1, \perp_{\mathcal{B}^\#}, l_1, d)$$

Since $d \in [l', \max(m, n)]$, the outer join includes the term for d . For this term:

$$\rho_f^\#(\mathbf{p}_1, \perp_{\mathcal{B}^\#}, l_1, d) = \bigsqcup_{i=l_1}^{\min(d,m)} \bigsqcup_{\mathcal{B}^\#} \rho_c^\#(\mathbf{p}_1, i, d)$$

Since $|x| \in [l_1, \min(d, m)]$ (because $|x| \leq m$ and $l_1 \leq |x| \leq d$ by construction), the inner join includes $i = |x|$. By Lemma 2.1.1, x is covered by $\mathbf{p}_1$ (since $|x| \leq m$, no padding is needed, the padded sequence coincides with $\mathbf{p}_1$). By Lemma 2.1.2, $\rho_c(x, d)$ is covered by $\rho_c^\#(\mathbf{p}_1, |x|, d)$. By Lemma 2.1.3 applied twice (once for the inner join over i , once for the outer join over d), x_r is covered by $\mathbf{p}'_1$.

- **Infinite case** ($s_1 \neq \perp_{\mathcal{B}^\#}$). Then $u_1 = +\infty$, and:

$$p'_1 = \bigsqcup_{d=l'}^{\max(m,n)} \mathcal{B}^\# \rho_f^\#(p_1, s_1, l_1, d)$$

Again $d \in [l', \max(m, n)]$. For this d :

$$\rho_f^\#(p_1, s_1, l_1, d) = \bigsqcup_{i=l_1}^d \mathcal{B}^\# \rho_c^\#(p_1^i, i, d)$$

where $p_1^{i_j} = p_{1,j}$ when $j \leq m$ and $p_1^{i_j} = s_1$ when $j > m$. Since $|x| \in [l_1, d]$, the join includes $i = |x|$. By Lemma 2.1.1, x is covered by the padded sequence $p_1^{|x|}$. By Lemma 2.1.2, $\rho_c(x, d)$ is covered by $\rho_c^\#(p_1^{|x|}, |x|, d)$. By Lemma 2.1.3, x_r is covered by p'_1 .

Summary. Two sub-cases:

- If $u' \neq +\infty$, then by definition $s'_1 = \perp_{\mathcal{B}^\#}$. Since $|p'_1| = \max(m, n) \geq d$ (the join of sequences of lengths up to $\max(m, n)$ has that length), all positions of x_r are covered by the known content and no positions beyond the known content exist. Hence $s'_1 = \perp_{\mathcal{B}^\#}$ is trivially satisfied (the condition on summary is vacuous).
- If $u' = +\infty$, then by definition:

$$s'_1 = s_1 \sqcup_{\mathcal{B}^\#} \bigsqcup_{k=1}^m \mathcal{B}^\# p_{1,k}$$

By Lemma 2.1.4, every position $x_{r,j}$ satisfies $x_{r,j} \in \gamma_{\mathcal{B}^\#}(s'_1)$. In particular, for any $j > |p'_1|$, we have $x_{r,j} \in \gamma_{\mathcal{B}^\#}(s'_1)$. Thus all positions, both those covered by the known content and those beyond, satisfy the concretization constraints.

Attributes. By definition of concrete Recycle (Definition 2.8.1), $x_r.\mathcal{A} = x.\mathcal{A}$. Abstract recycling gives $a'_1 = a_1$. Since $x \in \gamma_{\mathcal{V}_B^\#}(\nu_1)$ implies $x.\mathcal{A} \in \gamma_{\text{Attr}^\#}(a_1)$, we have $x_r.\mathcal{A} \in \gamma_{\text{Attr}^\#}(a_1) = \gamma_{\text{Attr}^\#}(a'_1)$, satisfying the attribute constraint.

:= Definition,

33

All four components (length, known content, summary, attributes) satisfy the conditions required by $\gamma_{\mathcal{V}_B^\#}$ (Definition 3.8.1). Therefore $x_r \in \gamma_{\mathcal{V}_B^\#}(\nu'_1)$.

:= Definition,

53

By symmetry, the same argument applied to ν_2 yields $y_r \in \gamma_{\mathcal{V}_B^\#}(\nu'_2)$. Consequently:

$$\text{Recycle}(x, y) = (x_r, y_r) \in \gamma_{\mathcal{V}_B^\#}(\nu'_1) \times \gamma_{\mathcal{V}_B^\#}(\nu'_2)$$

which completes the proof.

□

Lemma 2.1.5

AdjustForZeros Soundness

Let the vectors $\nu = ([l, u], \mathbf{p}, s, a) \in \mathcal{V}_{\mathcal{B}}^{\sharp}$ and $\nu' = \text{AdjustForZeros}(\nu) = ([l', u'], \mathbf{p}', s, a)$. For any concrete interval vector $c \in \gamma_{\mathcal{V}_{\text{Itv}}^{\sharp}}(\nu)$, let c' be c with all zero elements removed. Then:

1. $|c'| \in [l', u']$
2. For all $j \in [1, \min(|c'|, |\mathbf{p}'|)]$: $c'(j) \in \gamma_{\mathcal{B}^{\sharp}}(\mathbf{p}'_j)$

Proof

Let $n = |\mathbf{p}|$ and $c \in \gamma_{\mathcal{V}_{\text{Itv}}^{\sharp}}(\nu)$. Let c' be c with all zero elements removed, and let z denote the total number of zero elements in c .

Length bound. Let $d = \hat{d}_{n+1}$ be the number of definite zeros among the n known positions, and $p = \hat{p}_{n+1}$ the number of possible zeros among them. Any concrete c has at least d zeros and at most $d + p$ zeros in its known prefix; beyond n , elements come from $\gamma_{\mathcal{B}^{\sharp}}(s)$ and may also be zero. Since $|c| \in [l, u]$, the compressed length satisfies $|c'| = |c| - z \in [l - d - p, u - d] = [l', u']$.

Content coverage. Let $q_i = \text{Propagate}(\langle p_i, \dots, p_n \rangle, s, \hat{d}_i, \hat{p}_i)$ for each $i \in [1, n]$, and let $i_1 < i_2 < \dots < i_k$ be the positions where $q_i \neq \perp_{\mathcal{B}^{\sharp}}$, so that $\mathbf{p}' = \langle q_{i_1}, \dots, q_{i_k} \rangle$. For a fixed concrete c , let $r_1 < \dots < r_m$ be the positions of non-zero elements of c at indices in $[1, n]$. We claim that for each $j \in [1, \min(m, k)]$, the j -th non-zero element $c(r_j)$ is covered by the j -th result $q_{i_j} \neq \perp_{\mathcal{B}^{\sharp}}$. First observe that the non- $\perp_{\mathcal{B}^{\sharp}}$ positions form an initial prefix $\{1, \dots, k\}$, hence $i_j = j$ for all $j \leq k$: if $s \neq \perp_{\mathcal{B}^{\sharp}}$, the base case of *Propagate* returns $s \neq \perp_{\mathcal{B}^{\sharp}}$, so no q_i can evaluate to $\perp_{\mathcal{B}^{\sharp}}$ (even when all suffix positions are consumed by pending zeros); hence $k = n$ and the property holds trivially. If $s = \perp_{\mathcal{B}^{\sharp}}$, we show $q_t = \perp_{\mathcal{B}^{\sharp}} \implies q_{t+1} = \perp_{\mathcal{B}^{\sharp}}$ for all t (the non- $\perp_{\mathcal{B}^{\sharp}}$ positions have no gaps). The recursion of *Propagate* is monotone in the counters: more pending zeros (definite or possible) can only absorb more non-zero positions and never create a non- $\perp_{\mathcal{B}^{\sharp}}$ result where none existed. For a definite zero p_t (Case 1 of the *Propagate* definition in Definition 4.5.3), q_{t+1} receives one additional pending definite zero versus q_t and can only be $\perp_{\mathcal{B}^{\sharp}}$. For a definite non-zero p_t with $\hat{d}_t > 0$ (Case 3), the same monotonicity argument applies: q_t uses counters $(\hat{d}_t - 1, \hat{p}_t)$ while q_{t+1} uses $(\hat{d}_t, \hat{p}_t)$. For a possible zero with $\hat{d}_t > 0$ (Case 2), if both branches are $\perp_{\mathcal{B}^{\sharp}}$, then the suffix from $t + 1$ has no surviving non-zero even with the smaller counters $(\hat{d}_t - 1, \hat{p}_t)$; the extra possible zero added to $\hat{p}_{t+1}$ cannot change this. The remaining cases (p_t non-zero with $\hat{d}_t = 0$, or p_t possible zero with $\hat{d}_t = 0$) include p_t directly in the join (Cases 4, 5, 6 of Definition 4.5.3), so $q_t \neq \perp_{\mathcal{B}^{\sharp}}$ and the implication holds vacuously. Thus it suffices to prove $c(r_j) \in \gamma_{\mathcal{B}^{\sharp}}(q_j)$ for each $j \leq \min(m, k)$.

Fix $j \in [1, \min(m, k)]$. Let $b = |\{t < j \mid c(t) \neq 0\}|$ be the number of non-zero elements of c strictly before position j , and $z = |\{t < j \mid c(t) = 0\}|$ the number of zeros before j . Since positions before j partition into non-zeros and zeros, $j = b + z + 1$, hence $z = j - b - 1$.

Because $c \in \gamma_{\mathcal{V}_{\text{lv}}}^{\#}(\nu)$, the number of actual zeros before j satisfies $z \in [\hat{d}_j, \hat{d}_j + \hat{p}_j]$; write $z = \hat{d}_j + t$ with $t \in [0, \hat{p}_j]$.

Consider the execution of $\text{Propagate}(\langle p_j, \dots, p_n \rangle, s, \hat{d}_j, \hat{p}_j)$ that follows the concrete vector c . At each position:

1. If $c(i) = 0$, it takes the branch that treats p_i as zero (Case 1 for definite zeros; the zero-preserving recursive branch of Case 2 or 4 for possible zeros); counters unchanged.
2. If $c(i) \neq 0$ and counters remain, it absorbs one pending zero (Case 3 when $\hat{d}_t > 0$; the recursive branch of Case 5 when $\hat{d}_t = 0$, $\hat{p}_t > 0$).
3. If $c(i) \neq 0$ and no counters remain, it returns p_i directly (Case 6, or the value-branch of Case 5).

Between j and $r_j - 1$ there are exactly $j - b - 1 = z$ non-zero concrete positions, namely $r_{b+1}, \dots, r_{j-1}$. Each absorbs one pending zero. The $\hat{d}_j$ definite zeros are consumed first (Case 3), then t of the $\hat{p}_j$ possible zeros (recursive branch of Case 5). After processing r_{j-1} , exactly $\hat{d}_j + t = z$ pending zeros have been absorbed, so the counters reach $(0, \hat{p}_j - t)$. At r_j , since no pending definite zeros remain, the behaviour depends on the possible zeros: if $\hat{p}_j = t$ (all possible zeros were real), Case 6 returns p_{r_j} ; if $\hat{p}_j > t$, Case 5 (the value branch) joins p_{r_j} with further propagation. In either case, p_{r_j} is a term in the join, and $c(r_j) \in \gamma_{\mathcal{B}^{\#}}(p_{r_j})$ holds because $c \in \gamma_{\mathcal{V}_{\text{lv}}}^{\#}(\nu)$. This execution path contributes one term to the join that defines q_j ; therefore $c(r_j) \in \gamma_{\mathcal{B}^{\#}}(q_j)$.

□

Lemma 2.1.6**Abstract Access Soundness**

For any $\nu \in \mathcal{V}_{\mathcal{B}^{\#}}^{\#}$ and any $v \in \gamma_{\mathcal{B}^{\#}}^{\#}(\nu)$:

$$\forall k \in \mathbb{N} \cup \{\text{NA}\} : v^*(k) \in \gamma_{\mathcal{B}^{\#}}^{\#}(\nu^{*\#}(k))$$

where $v^*(\text{NA}) = \text{NA}$ and $\nu^{*\#}(\text{NA}) = \alpha_{\mathcal{B}^{\#}}(\text{NA})$.

Proof

Let $\nu = ([l_{\nu}, u_{\nu}], p_{\nu}, s_{\nu}, a_{\nu})$. By definition of v^* , for $k \in \mathbb{N}$: if $1 \leq k \leq |v|$, then $v^*(k) = v(k)$; otherwise $v^*(k) = \text{NA}$.

By Definition 4.5.5, $\nu^{*\#}(k)$ is defined as:

$$\nu^{*\#}(k) = \begin{cases} p_{\nu, k} & \text{if } 1 \leq k \leq l_{\nu} \\ p_{\nu, k} \sqcup_{\mathcal{B}^{\#}} \alpha_{\mathcal{B}^{\#}}(\text{NA}) & \text{if } l_{\nu} < k \leq |p_{\nu}| \\ s_{\nu} \sqcup_{\mathcal{B}^{\#}} \alpha_{\mathcal{B}^{\#}}(\text{NA}) & \text{if } |p_{\nu}| < k \leq u_{\nu} \\ \alpha_{\mathcal{B}^{\#}}(\text{NA}) & \text{otherwise} \end{cases}$$

Case $1 \leq k \leq l_\nu$. Then $\nu^{*\sharp}(k) = p_{\nu,k}$. Since $v \in \gamma_{\mathcal{V}_B^\sharp}(\nu)$, we have $l_\nu \leq |v|$, hence $k \leq |v|$ and $v^*(k) = v(k)$. By concretization of v , $v(k) \in \gamma_{\mathcal{B}^\sharp}(p_{\nu,k})$.

Case $l_\nu < k \leq |p_\nu|$. Then $\nu^{*\sharp}(k) = p_{\nu,k} \sqcup_{\mathcal{B}^\sharp} \alpha_{\mathcal{B}^\sharp}(\text{NA})$. If $k \leq |v|$, then $v^*(k) = v(k)$ and $v(k) \in \gamma_{\mathcal{B}^\sharp}(p_{\nu,k})$ by concretization. If $k > |v|$, then $v^*(k) = \text{NA}$ and $\text{NA} \in \gamma_{\mathcal{B}^\sharp}(\alpha_{\mathcal{B}^\sharp}(\text{NA}))$. In both subcases, the join covers the concrete value.

Case $|p_\nu| < k \leq u_\nu$. Then $\nu^{*\sharp}(k) = s_\nu \sqcup_{\mathcal{B}^\sharp} \alpha_{\mathcal{B}^\sharp}(\text{NA})$. Note that $u_\nu = +\infty$ (otherwise $u_\nu = |p_\nu|$ by the domain invariant, contradicting $|p_\nu| < u_\nu$). If $k \leq |v|$, then $v^*(k) = v(k)$ and, since $k > |p_\nu|$, concretization gives $v(k) \in \gamma_{\mathcal{B}^\sharp}(s_\nu)$. If $k > |v|$, then $v^*(k) = \text{NA}$ and $\text{NA} \in \gamma_{\mathcal{B}^\sharp}(\alpha_{\mathcal{B}^\sharp}(\text{NA}))$. Again the join covers both possibilities.

Case otherwise ($k > u_\nu$, $k < 1$, or $k = \text{NA}$). Then $\nu^{*\sharp}(k) = \alpha_{\mathcal{B}^\sharp}(\text{NA})$. In each situation $v^*(k) = \text{NA}$, since k is either out of bounds, below the first position, or NA itself. And $\text{NA} \in \gamma_{\mathcal{B}^\sharp}(\alpha_{\mathcal{B}^\sharp}(\text{NA}))$ holds by definition of the abstraction.

□

B.2 Soundness of Abstract Positive Selection

Proof

of Theorem 4.6.1

Let $\nu_1 = ([l_1, u_1], p_1, s_1, a_1)$ with $|p_1| = k_1$ and $\nu_2^+ = ([l_2, u_2], p_2^+, s_2, a_2)$ with $|p_2^+| = k_2$. Let $v \in \gamma_{\mathcal{V}_B^\sharp}(\nu_1)$ and $c \in \gamma_{\mathcal{V}_{\text{lv}}^\sharp}(\nu_2^+)$ with $|c| = n$.

First, apply `AdjustForZeros` to the selector: $([l'_2, u'_2], p'_2, s_2, a_2) = \text{AdjustForZeros}([l_2, u_2], p_2^+, s_2, a_2)$. The correctness of `AdjustForZeros` follows from the soundness of `Propagate`: each non-zero selector position is correctly shifted forward past preceding zeros, and zeros are removed from the content. The length adjustments (l'_2, u'_2) correctly account for definite and possible zero removal.

By the concrete semantics of `Select` (Definition 2.9.1):

:= Definition,

34

$$\text{Select}(v, c) = \bigoplus_{i=1}^n \sigma_i$$

where σ_i is $[v^*(c(i))]$ if $c(i) > 0$, $[]$ if $c(i) = 0$, and $[\text{NA}]$ if $c(i) = \text{NA}$.

For the case $u'_2 = 0$, the selector contains only zeros, so $\text{Select}(v, c) = []$. The abstract result is $([0, 0], \varepsilon, \perp_{\mathcal{B}^\sharp}, \text{AttrEmpty})$, and $[] \in \gamma_{\mathcal{V}_B^\sharp}(([0, 0], \varepsilon, \perp_{\mathcal{B}^\sharp}, \text{AttrEmpty}))$ holds.

For $u'_2 \neq +\infty$, the abstract result is $\nu_r = ([l'_2, u'_2], p_r, \perp_{\mathcal{B}^\sharp}, a_1)$ where:

$$p_r = ()_{i=1}^{|p'_2|} \oplus \begin{cases} \bigsqcup_{j \in \mathcal{B}^\sharp} \nu_1^{*\sharp}(j) & \text{if } \text{Enumerable}(p_{2,i}^+) \\ \text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\sharp} \alpha_{\mathcal{B}^\sharp}(\text{NA}) & \text{otherwise} \end{cases}$$

with $j \in \gamma_{\text{ltv}}(p_{2,i}^+), j \neq 0$.

For each concrete index $c(i) > 0$, let $j' = c(i)$ be the extracted position. By the soundness of $\nu_1^{*\sharp}(j')$:

- If $1 \leq j' \leq l_1$: $v(j') \in \gamma_{\mathcal{B}^\sharp}(p_{1,j'}) = \gamma_{\mathcal{B}^\sharp}(\nu_1^{*\sharp}(j'))$.
- If $l_1 < j' \leq k_1$: $v(j') \in \gamma_{\mathcal{B}^\sharp}(p_{1,j'})$ and the join with $\alpha_{\mathcal{B}^\sharp}(\text{NA})$ is harmless since $v(j') \neq \text{Null}$.
- If $k_1 < j' \leq u_1$: $v(j') \in \gamma_{\mathcal{B}^\sharp}(s_1)$, covered by $\nu_1^{*\sharp}(j') = s_1 \sqcup_{\mathcal{B}^\sharp} \alpha_{\mathcal{B}^\sharp}(\text{NA})$.
- If $j' > u_1$: $v^*(j') = \text{NA} \in \gamma_{\mathcal{B}^\sharp}(\alpha_{\mathcal{B}^\sharp}(\text{NA}))$.

When the abstract selector position $p_{2,i}^+$ is enumerable, the join over $j \in \gamma_{\text{ltv}}(p_{2,i}^+)$ includes j' , and by Lemma 2.1.3 the concretization of the join covers $v^*(c(i))$. When non-enumerable, $\text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\sharp} \alpha_{\mathcal{B}^\sharp}(\text{NA})$ over-approximates all possible values by construction: $\text{Squash}(\nu_1)$ collapses all known and summary values of ν_1 , and the $\alpha_{\mathcal{B}^\sharp}(\text{NA})$ term covers out-of-bounds access.

The length $[l'_2, u'_2]$ is correct: the concrete result has length equal to the number of non-zero selector entries (after zero discarding), which AdjustForZeros bounds between l'_2 and u'_2 . The summary $\perp_{\mathcal{B}^\sharp}$ is correct because $u'_2 \neq +\infty$ means all selected positions are explicitly tracked in p_r .

The infinite case ($u'_2 = +\infty$) differs only in the summary: if $u_r > l_1$, where $\text{Squash}(\nu_2^+) = [l, u_r]$, then $s_r = \text{Squash}(\nu_1) \sqcup_{\mathcal{B}^\sharp} \alpha_{\mathcal{B}^\sharp}(\text{NA})$ covers possible out-of-bounds access; otherwise $s_r = \text{Squash}(\nu_1)$ suffices since all selected positions fall within the definite bounds of ν_1 .

Attributes. The abstract selection preserves a_1 from the data vector ν_1 , matching the concrete behavior where the result inherits the attributes of v . Since $v.\mathcal{A} \in \gamma_{\text{Attr}^\sharp}(a_1)$, we have $r.\mathcal{A} \in \gamma_{\text{Attr}^\sharp}(a_1)$.

Thus $\text{Select}(v, c) \in \gamma_{\mathcal{V}_{\mathcal{B}^\sharp}^*}(\nu_r)$, completing the proof.

□

Bibliography

- [1] Patrick Cousot and Radhia 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* (1977), pages 238–252. DOI: [10.1145/512950.512973](https://doi.org/10.1145/512950.512973).
- [2] R Core Team. *R Language Definition*. Manual. R Foundation for Statistical Computing. 2026. URL: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.
- [3] Robert C. Gentleman, Vincent J. Carey, Douglas M. Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, Kurt Hornik, Torsten Hothorn, Wolfgang Huber, Stefano Iacus, Rafael Irizarry, Friedrich Leisch, Cheng Li, Martin Maechler, Anthony J. Rossini, Gunther Sawitzki, Colin Smith, Gordon Smyth, Luke Tierney, Jean Y. H. Yang, and Jianhua Zhang. “Bioconductor: Open Software Development for Computational Biology and Bioinformatics”. In: *Genome Biology* 5.10 (2004), page R80. DOI: [10.1186/gb-2004-5-10-r80](https://doi.org/10.1186/gb-2004-5-10-r80).
- [4] John P. Nolan. “R as a Tool in Computational Finance”. In: *Handbook of Computational Finance*. Edited by Jin-Chuan Duan, Wolfgang Härdle, and James E. Gentle. Berlin, Heidelberg: Springer, 2012. Pages 675–693. DOI: [10.1007/978-3-642-17254-0_29](https://doi.org/10.1007/978-3-642-17254-0_29).
- [5] CRAN. *Contributed Packages*. Accessed: 2026-06-17, lists 23,938 available packages. 2026. URL: <https://cran.r-project.org/web/packages/>.
- [6] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, and Matei Zaharia. “SparkR: Scaling R Programs with Spark”. In: *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 2016. Pages 1099–1104. DOI: [10.1145/2882903.2903740](https://doi.org/10.1145/2882903.2903740).
- [7] Greta Dolcetti, Vincenzo Arceri, Antonella Mensi, Enea Zaffanella, Caterina Urban, and Agostino Cortesi.

- “Introducing Pyra: A High-Level Linter for Data Science Software”. In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2025)*. Springer, 2025. DOI: [10.1007/978-3-031-83842-8_29](https://doi.org/10.1007/978-3-031-83842-8_29).
- [8] John M. Chambers. *Programming with Data: A Guide to the S Language*. 1st edition. New York, NY: Springer, 1998. DOI: [10.1007/978-0-387-98503-9](https://doi.org/10.1007/978-0-387-98503-9).
- [9] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole Computer Science Series. Pacific Grove, CA: Wadsworth & Brooks/Cole Advanced Books & Software, 1988.
- [10] Ross Ihaka and Robert Gentleman. “R: A Language for Data Analysis and Graphics”. In: *Journal of Computational and Graphical Statistics* 5.3 (1996), pages 299–314. DOI: [10.1080/10618600.1996.10474713](https://doi.org/10.1080/10618600.1996.10474713).
- [11] John Fox. “Aspects of the Social Organization and Trajectory of the R Project”. In: *The R Journal* 1.2 (2009), pages 5–13. DOI: [10.32614/RJ-2009-014](https://doi.org/10.32614/RJ-2009-014).
- [12] Florian Sihler, Lukas Pietzschmann, Raphael Straub, Matthias Tichy, Andor Diera, and Abdelhalim Dahou. “On the Anatomy of Real-World R Code for Static Analysis”. In: *Proc. 21st Int. Conf. Mining Software Repositories (MSR)*. ACM, 2024. DOI: [10.1145/3643991.3644911](https://doi.org/10.1145/3643991.3644911).
- [13] Hadley Wickham. *Advanced R*. 2nd. Chapter 3: Vectors. Boca Raton, FL: CRC Press, 2019. URL: <https://adv-r.hadley.nz/vectors-chap.html>.
- [14] Aviral Goel and Jan Vitek. “On the Design, Implementation, and Use of Laziness in R”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), pages 153:1–153:27. DOI: [10.1145/3360579](https://doi.org/10.1145/3360579).
- [15] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. “Evaluating the Design of the R Language: Objects and Functions for Data Analysis”. In: *26th European Conference on Object-Oriented Programming (ECOOP)*. Volume 7313. LNCS. Springer, 2012. Pages 104–131. DOI: [10.1007/978-3-642-31057-7_6](https://doi.org/10.1007/978-3-642-31057-7_6).
- [16] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. “R Melts Brains: An IR for First-Class Environments and Lazy Effective Arguments”. In: *Proc. 15th ACM SIGPLAN Int. Symp. Dynamic Languages (DLS)*. ACM, 2019. Pages 55–66. DOI: [10.1145/3359619.3359744](https://doi.org/10.1145/3359619.3359744).

- [17] Martin Bodin, Tomás Diaz, and Éric Tanter. “A Trustworthy Mechanized Formalization of R”. In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018*. ACM, 2018. Pages 13–24. DOI: [10.1145/3276945.3276946](https://doi.org/10.1145/3276945.3276946).
- [18] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pages 358–366. DOI: [10.1090/S0002-9947-1953-0053041-6](https://doi.org/10.1090/S0002-9947-1953-0053041-6).
- [19] Alfred Tarski. “A Lattice-Theoretical Fixpoint Theorem and Its Applications”. In: *Pacific Journal of Mathematics* 5.2 (1955), pages 285–309. DOI: [10.2140/pjm.1955.5.285](https://doi.org/10.2140/pjm.1955.5.285).
- [20] Patrick Cousot. *Principles of Abstract Interpretation*. Cambridge, MA: The MIT Press, 2021.
- [21] Antoine Miné. “The Octagon Abstract Domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pages 31–100. DOI: [10.1007/s10990-006-8609-1](https://doi.org/10.1007/s10990-006-8609-1).
- [22] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *Proc. 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 1978. Pages 84–96. DOI: [10.1145/512760.512770](https://doi.org/10.1145/512760.512770).
- [23] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. “A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis”. In: *SIGPLAN Notices* 46.1 (2011), pages 105–118. DOI: [10.1145/1925844.1926399](https://doi.org/10.1145/1925844.1926399).
- [24] Rathijit Sen, Jianqiao Zhu, Jignesh M. Patel, and Somesh Jha. “ROSA: R Optimizations with Static Analysis”. In: *RIOT 2017*. ArXiv, 2017. URL: <https://www.microsoft.com/en-us/research/publication/rosa-r-optimizations-with-static-analysis/>.
- [25] Luke Tierney. *codetools: Code Analysis Tools for R*. R Foundation for Statistical Computing. Vienna, Austria, 2024. DOI: [10.32614/CRAN.package.codetools](https://doi.org/10.32614/CRAN.package.codetools). URL: <https://CRAN.R-project.org/package=codetools>.
- [26] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. “A Fast Abstract Syntax Tree Interpreter for R”. In: *ACM SIGPLAN Notices* 49.7 (2014), pages 89–102.
- [27] Radford M. Neal. *Speed Improvements in pqR: Current Status and Future Plans*. 2014. URL: <https://glizen.com/radfordneal/ftp/pqR-dsc.pdf>.
- [28] Olivier Flückiger, Andreas Wälchli, Sebastián Krynski, and Jan Vitek. “Sampling Optimized Code for Type Feedback”. In:

- Proc. 16th ACM SIGPLAN Int. Symp. Dynamic Languages (DLS)*. ACM, 2020. Pages 99–111.
- [29] Florian Sihler and Matthias Tichy. “[Statically Analyzing the Dataflow of R Programs](#)”. In: *Proc. ACM Program. Lang.* 9.OOPSLA2 (2025), pages 1034–1062. DOI: [10.1145/3763087](#).
 - [30] Simon Holm Jensen, Anders Møller, and Peter Thiemann. “[Type Analysis for JavaScript](#)”. In: *Proc. 16th Int. Symp. Static Analysis (SAS)*. Volume 5673. LNCS. Springer, 2009. Pages 238–255. DOI: [10.1007/978-3-642-03237-0_17](#).
 - [31] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. “[JSAl: A Static Analysis Platform for JavaScript](#)”. In: *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*. ACM, 2014. Pages 121–132. DOI: [10.1145/2635868.2635904](#).
 - [32] Vincenzo Arceri, Michele Pasqua, and Isabella Mastroeni. “[An Abstract Domain for Objects in Dynamic Programming Languages](#)”. In: *FM 2019 International Workshops*. Volume 12233. LNCS. Springer, 2020. Pages 136–151. DOI: [10.1007/978-3-030-54997-8_9](#).
 - [33] Etienne Kneuss, Philippe Suter, and Viktor Kunčák. “[Phantm: PHP Analyzer for Type Mismatch](#)”. In: *Proc. 18th ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE)*. ACM, 2010. Pages 373–374. DOI: [10.1145/1882291.1882355](#).
 - [34] David Hauzar and Jan Kofroň. “[Framework for Static Analysis of PHP Applications](#)”. In: *29th European Conference on Object-Oriented Programming (ECOOP)*. Volume 37. LIPIcs. Schloss Dagstuhl, 2015. Pages 689–711. DOI: [10.4230/LIPIcs.ECOOP.2015.689](#).
 - [35] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. “[Static Type Analysis by Abstract Interpretation of Python Programs](#)”. In: *34th European Conference on Object-Oriented Programming (ECOOP)*. Volume 166. LIPIcs. Schloss Dagstuhl, 2020. Pages 17:1–17:29. DOI: [10.4230/LIPIcs.ECOOP.2020.17](#).
 - [36] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “[A Static Analyzer for Large Safety-Critical Software](#)”. In: *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2003. Pages 196–207. DOI: [10.1145/781131.781153](#).
 - [37] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. “[Array-Data Flow Analysis and Its Use in Array](#)

- [Privatization](#)". In: *Proc. 20th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*. ACM, 1993. Pages 2–15. DOI: [10.1145/158511.158515](#).
- [38] Josselin Giet, Félix Ridoux, and Xavier Rival. "[A Product of Shape and Sequence Abstractions](#)". In: *Static Analysis — 30th International Symposium, SAS 2023*. Volume 14284. Lecture Notes in Computer Science. Springer Nature Switzerland, 2023. Pages 310–342. DOI: [10.1007/978-3-031-44245-2_15](#).
- [39] Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. "[Statically Inferring Complex Heap, Array, and Numeric Invariants](#)". In: *Static Analysis – 17th International Symposium, SAS 2010*. LNCS. Springer, 2010. Pages 71–99. DOI: [10.1007/978-3-642-15769-1_6](#).
- [40] Luca Negrini, Guruprerana Shabadi, and Caterina Urban. "[Static Analysis of Data Transformations in Jupyter Notebooks](#)". In: *Proc. 12th ACM SIGPLAN Int. Workshop on the State Of the Art in Program Analysis (SOAP)*. ACM, 2023. Pages 8–13. DOI: [10.1145/3589250.3596145](#).
- [41] Oliver Gerstl. "[Tracking the Shape of Data Frames in R Programs Using Abstract Interpretation](#)". Master's thesis, Universität Ulm, 2025. DOI: [10.18725/OPARU-58621](#).