

Brrr-Machine Synthesis Specification

Complete Multi-Language Program Analysis Framework

Brrr-Machine Project

January 2026

Abstract

This document presents the complete brrr-machine synthesis specification, establishing the theoretical foundations and architectural principles for multi-language program analysis. The brrr-machine treats programming languages as parameterized instantiations of a universal computational model, enabling uniform analysis across heterogeneous language ecosystems. We present the vision, core thesis, and foundational pillars synthesized from seminal papers in programming language theory: abstract interpretation, program representation, interprocedural dataflow, pointer analysis, effect systems, substructural types, and security analysis.

Contents

| | | |
|-----------|--|-----------|
| I | Preamble: Theoretical Foundations and Architectural Principles | 16 |
| 1 | Introduction and Motivation | 17 |
| 1.1 | Principal Contributions | 17 |
| 2 | The Core Thesis: Languages as Parameterized Type Theories | 17 |
| 2.1 | Language Parameter Configuration Matrix | 18 |
| 3 | Foundational Literature and Theoretical Pillars | 18 |
| 3.1 | Pillar 1: Abstract Interpretation — Semantic Foundation | 18 |
| 3.1.1 | Theoretical Contribution | 18 |
| 3.2 | Pillar 2: Program Representation — Unified Graph Structure | 19 |
| 3.2.1 | Theoretical Contribution | 19 |
| 3.3 | Pillar 3: Interprocedural Dataflow — Algorithmic Framework | 20 |
| 3.3.1 | Theoretical Contribution | 20 |
| 3.4 | Pillar 4: Pointer and Alias Analysis — Precision Infrastructure | 21 |
| 3.4.1 | Theoretical Contribution | 21 |
| 3.5 | Pillar 5: Effect Systems — Computational Behavior Tracking | 21 |
| 3.5.1 | Theoretical Contribution | 21 |
| 3.6 | Pillar 6: Substructural Types and Ownership — Resource Invariants | 22 |
| 3.6.1 | Theoretical Contribution | 22 |
| 3.7 | Pillar 7: Security Analysis — Information Flow and Vulnerability Detection . . . | 23 |
| 3.7.1 | Theoretical Contribution | 23 |
| 3.8 | Cross-Cutting Concerns: Multi-Language Interoperability and Incrementality . . | 24 |
| 3.8.1 | Theoretical Contribution | 24 |
| 4 | Document Organization and Conventions | 24 |
| 4.1 | Section Template | 24 |
| 4.2 | Notational Conventions | 25 |
| II | Theoretical Foundations | 26 |
| 5 | Abstract Interpretation: The Central Dogma | 27 |
| 5.1 | The Concrete and Abstract Worlds | 28 |
| 5.2 | Galois Connections | 28 |
| 5.3 | Complete Lattices | 29 |
| 5.4 | Fixpoint Computation | 29 |
| 5.5 | Widening and Narrowing | 30 |
| 5.6 | Lazy Evaluation and Verification Condition Soundness | 31 |
| 5.7 | Neural Invariant Synthesis (Optional Enhancement) | 33 |
| 5.8 | F* Specification: Abstract Domains | 34 |
| 5.9 | Widening and Narrowing Theoretical Foundation (Cousot 1992) | 37 |
| 5.10 | Probabilistic Abstract Domains (Cousot 2012) | 39 |

| | | |
|------------|--|-----------|
| 5.11 | Concrete Abstract Domains | 42 |
| 5.12 | Occurrence Type Domain | 49 |
| 5.13 | DeepPoly: Abstract Domain for Neural Networks | 53 |
| 5.14 | Dual Soundness: Over and Under Approximation | 56 |
| 5.15 | Local Completeness Logic (Bruni et al. 2023) | 57 |
| 5.16 | Unified Constraint Domain Framework | 60 |
| III | Program Representation | 62 |
| 6 | The Code Property Graph | 63 |
| 6.1 | Why CPG? | 63 |
| 6.2 | CPG Components | 63 |
| 6.2.1 | AST (Abstract Syntax Tree) | 63 |
| 6.2.2 | CFG (Control Flow Graph) | 63 |
| 6.2.3 | PDG (Program Dependence Graph) | 64 |
| 6.2.4 | Call Graph | 64 |
| 6.2.5 | Effect Graph (brrr-machine Extension) | 64 |
| 6.3 | Edge Types in Detail | 64 |
| 6.4 | Node Types | 67 |
| 6.5 | CPG Data Structure | 70 |
| 6.6 | CPG Traversal Primitives | 71 |
| 6.6.1 | TNODES: Tree Nodes (AST Subtree Collection) | 72 |
| 6.6.2 | MATCH: Find Nodes Matching Predicate in Subtrees | 72 |
| 6.6.3 | Traversal Combinators | 72 |
| 6.6.4 | Buffer Overflow Query Example | 73 |
| 6.6.5 | Generic Vulnerability Query Structure | 74 |
| 6.6.6 | Basic Traversals | 74 |
| 6.6.7 | Transitive Closures | 75 |
| 6.6.8 | Program Slicing | 75 |
| 6.6.9 | Two-Phase Interprocedural Slicing | 77 |
| 6.6.10 | Taint Analysis Traversals | 78 |
| 6.6.11 | Label Patterns | 78 |
| 6.7 | CPG Construction | 79 |
| 6.7.1 | Control Dependence Construction | 80 |
| 6.7.2 | Data Dependence Construction | 80 |
| 6.7.3 | Reaching Definitions | 81 |
| 6.8 | Neural Program Graph Embeddings (Optional Enhancement) | 82 |
| 6.8.1 | Semantic Identifier Embeddings (DeepBugs) | 84 |
| IV | Analysis Algorithms | 91 |
| 7 | IFDS: Interprocedural Finite Distributive Subset | 93 |
| 7.1 | The Key Insight | 93 |
| 7.2 | The Exploded Supergraph | 94 |
| 7.3 | The Tabulation Algorithm | 94 |
| 7.4 | Common IFDS Problems | 98 |

| | | |
|-----------|---|------------|
| 7.5 | IFDS Implementation Optimizations | 100 |
| 7.5.1 | Representation Relations | 101 |
| 7.5.2 | H-Sparse Optimization | 102 |
| 7.5.3 | Locally Separable Problems (Gen/Kill) | 103 |
| 7.5.4 | Demand-Driven IFDS | 105 |
| 7.5.5 | Summary: IFDS Complexity Variants | 107 |
| 7.6 | Datalog Compilation Strategy (Souffle) | 107 |
| 7.6.1 | Staged Compilation via Futamura Projections | 107 |
| 7.6.2 | Optimal Index Selection (Dilworth's Theorem) | 108 |
| 7.6.3 | Implementation Strategy for brrr-machine | 109 |
| 7.7 | Lattice-Extended Datalog and IDE (Flix) | 109 |
| 7.7.1 | Lattice Predicates and Transfer Functions | 109 |
| 7.7.2 | IDE as Flix with Micro-Function Lattice | 110 |
| 7.7.3 | Semi-Naive Evaluation for Lattices | 111 |
| 8 | CFL-Reachability | 112 |
| 8.1 | The Framework | 112 |
| 8.2 | Demand-Driven Analysis via CFL | 113 |
| 8.3 | Interleaved Dyck and the Combined Context+Field Problem | 114 |
| 8.4 | Multiple Context-Free Language Reachability (MCFL) | 114 |
| 8.5 | Synchronized Pushdown Systems (SPDS) | 116 |
| 8.6 | Cross-Reference: Sparse Value-Flow Analysis (SVF) | 118 |
| 9 | Under-Approximate Analysis (Bug Finding) | 118 |
| 9.1 | The Eval Algorithm (Pulse/Infer) | 118 |
| 9.2 | ISL Triple Semantics | 120 |
| 9.3 | Latent vs Manifest Errors | 121 |
| 9.4 | Integration: IFDS + Eval Hybrid | 123 |
| 10 | Symbolic Execution (Path-Sensitive) | 124 |
| 10.1 | Execution Tree vs CPG | 125 |
| 10.2 | On-Demand Symbolic Execution | 125 |
| 10.3 | Witness Generation | 127 |
| 10.4 | Constraint Solver Optimizations | 128 |
| 10.5 | Concolic Testing Optimizations | 131 |
| 10.6 | Chopped Symbolic Execution | 133 |
| 10.6.1 | The Path Explosion Problem | 133 |
| 10.6.2 | The Four State Types | 134 |
| 10.6.3 | Guiding Constraints | 134 |
| 10.6.4 | The Recovery Mechanism | 135 |
| 10.6.5 | Correctness Properties | 135 |
| 10.6.6 | Integration with Outcome Logic | 136 |
| 10.7 | Optimistic Concolic Execution (QSYM) | 136 |
| 10.7.1 | Soundness Model: Validation-Based | 138 |
| V | Pointer Analysis — The Precision Foundation | 140 |
| 11 | Andersen's Analysis | 142 |

| | | |
|-----------|---|------------|
| 11.1 | The Constraint Language | 142 |
| 11.2 | Constraint Solving | 142 |
| 11.3 | Limitations from Andersen 1994 | 145 |
| 11.4 | Inter-Procedural Context Sensitivity | 146 |
| 11.5 | Cross-References: Pointer Analysis Dependencies | 146 |
| 12 | Steensgaard’s Analysis | 147 |
| 12.1 | Conditional Join (cjoin) | 149 |
| 12.2 | Data Structure Analysis (DSA) | 150 |
| 12.2.1 | DS Graph Structure | 150 |
| 12.2.2 | The Completeness Flag | 151 |
| 12.2.3 | Three-Phase Algorithm | 151 |
| 12.2.4 | Heap Cloning: Distinguishing Data Structure Instances | 152 |
| 12.2.5 | Engineering Optimizations | 152 |
| 12.2.6 | Precision Comparison with Andersen | 153 |
| 13 | On-the-Fly Call Graph Construction | 153 |
| 13.1 | When to Use Each Approach | 155 |
| 13.2 | Selective Context Sensitivity (ZIPPER) | 155 |
| 13.2.1 | The Uniform Context Sensitivity Problem | 155 |
| 13.2.2 | The Three Precision-Loss Patterns | 156 |
| 13.2.3 | The ZIPPER Algorithm | 156 |
| 13.2.4 | Empirical Results | 157 |
| 13.3 | Python-Specific Call Graph via Type Inference | 157 |
| 13.3.1 | C3 Linearization for Python MRO | 158 |
| 13.3.2 | Call Graph Algorithm Selection | 159 |
| 14 | Shape Analysis via Three-Valued Logic | 159 |
| 14.1 | Three-Valued Foundation | 159 |
| 14.2 | Shape Structures | 160 |
| 14.3 | Embedding Theorem | 160 |
| 14.4 | Focus-Transform-Coerce Algorithm | 160 |
| 14.5 | Integration with Brrr-Machine | 161 |
| 15 | Symbolic Heap Shape Analysis | 161 |
| 15.1 | Symbolic Heap Structure | 161 |
| 15.2 | Heap Predicates | 162 |
| 15.3 | Canonicalization as Widening | 163 |
| 15.4 | Memory Leak Detection via Junk | 163 |
| 15.5 | Frame Rule for Compositional Analysis | 164 |
| 15.6 | Symbolic Execution for Symbolic Heaps | 165 |
| 15.7 | Comparison: TVLA vs Symbolic Heaps | 166 |
| 16 | Sparse Value-Flow Analysis (SVF) | 166 |
| 16.1 | Motivation: Dense vs Sparse Dataflow | 167 |
| 16.2 | Memory SSA | 167 |
| 16.3 | SVFG Construction | 168 |
| 16.4 | Context-Sensitivity via CFL-Reachability | 169 |

| | | |
|------------|---|------------|
| 16.5 | IFDS vs SVF Comparison | 169 |
| 16.6 | When to Use Each Approach | 169 |
| 16.7 | Leak Detection Algorithm | 170 |
| 16.8 | Cross-References | 171 |
| VI | Effect Systems and Coeffects—Tracking Computational Behavior | 172 |
| 17 | Effects as First-Class Citizens | 173 |
| 17.1 | The Moggi Foundation | 173 |
| 17.2 | Algebraic Effects | 173 |
| 17.3 | Row-Polymorphic Effects | 175 |
| 17.4 | F* Specification: Effect System | 175 |
| 18 | Effect Handler Limitations | 181 |
| 18.1 | Production Effect Handler Implementation | 183 |
| 19 | C11/C++ Memory Model | 185 |
| 19.1 | Memory Orderings and Actions | 186 |
| 19.2 | Execution Witnesses and Candidate Executions | 187 |
| 19.3 | Release Sequences | 188 |
| 19.4 | Coherence Axioms | 189 |
| 19.5 | Data Race Detection | 189 |
| 19.6 | Sequential Consistency and SC-DRF | 190 |
| 19.7 | Promising Semantics 2.0 for Sound Relaxed Atomics | 191 |
| 19.8 | Cross-Reference Summary for Memory Models | 193 |
| 19.9 | IMM: Intermediate Memory Model for Compilation Proofs | 193 |
| 20 | Coeffect Systems: Context Requirements | 195 |
| 20.1 | Flat vs Structural Coeffects | 195 |
| 20.2 | Liveness as a Structural Coeffect | 196 |
| 20.3 | Usage Coeffects and Linear Types | 196 |
| 20.4 | Platform Capability Coeffects | 197 |
| 20.5 | F* Specification: Coeffect System | 197 |
| VII | Ownership and Resources | 200 |
| 21 | Resource Algebras from Iris | 203 |
| 21.1 | The Camera Abstraction | 203 |
| 21.1.1 | Ghost State and Invariants | 204 |
| 21.1.2 | View Shifts and Step-Indexing | 204 |
| 21.2 | Separation Logic for Memory | 205 |
| 21.3 | F* Specification: Ownership System | 205 |
| 21.3.1 | Resource Algebra Typeclass | 205 |
| 21.4 | Capability Multiplicities | 208 |
| 22 | Access Permissions and Tpestate | 209 |
| 22.1 | Access Permissions (Bierhoff) | 209 |

| | | |
|-------------|---|------------|
| 22.1.1 | Per-Node Fraction Functions (Bierhoff 2007 Extended) | 210 |
| 22.1.2 | Temporary State in Weak Permissions | 211 |
| 22.2 | MALL Specifications for Method Contracts | 211 |
| 22.3 | Vault Adoption and Focus Mechanisms | 212 |
| 22.4 | Frame-Based OOP Inheritance | 214 |
| 22.4.1 | API Pattern Examples with Adoption and Focus | 215 |
| 22.5 | Rust Verification: RefinedRust and Prusti | 217 |
| 22.5.1 | Place Types and Blocked Types (RefinedRust) | 217 |
| 22.5.2 | Pledge Specifications for Reborrowing (Prusti) | 218 |
| 23 | Linearizability for Concurrent Objects | 218 |
| 23.1 | Progress Properties for Concurrent Objects | 219 |
| 23.2 | Temporal Logic for Liveness | 220 |
| 23.3 | CTL Model Checking | 221 |
| 24 | Frame Rule and Footprint Computation | 222 |
| 24.1 | The Frame Rule | 222 |
| 24.2 | Footprint Computation | 222 |
| 24.3 | Compositional Analysis Algorithm | 223 |
| 24.4 | Magic Wands for Partial Data Structures | 223 |
| 24.5 | Quantified Permissions for Arrays | 223 |
| 24.6 | Automated Proof Search for Separation Logic | 224 |
| 25 | Representation Predicates | 224 |
| 26 | GC-Aware Ownership Analysis | 225 |
| 26.1 | GC-Isomorphism Definition | 225 |
| 26.2 | Ownership in GC Languages | 226 |
| 26.3 | GC at Language Boundaries | 226 |
| 27 | WebAssembly Memory Safety Analysis | 226 |
| 27.1 | WebAssembly Linear Memory Model | 227 |
| 27.2 | MS-Wasm Segment Memory Model | 227 |
| 27.3 | MS-Wasm Memory Safety Properties | 228 |
| 27.4 | Hardware-Accelerated Memory Safety | 228 |
| 27.5 | WebAssembly Analysis Landscape (2024 Survey) | 228 |
| 27.6 | Integration with Brrr-Machine Analysis | 228 |
| VIII | Security Analysis — Information Flow and Taint | 230 |
| 28 | Information Flow and Taint Analysis | 231 |
| 28.1 | Denning’s Lattice Model | 231 |
| 28.1.1 | Taint Mapping | 232 |
| 28.1.2 | The 4-Point Lattice with Integrity | 232 |
| 28.2 | The Taint Analysis Framework | 233 |
| 28.3 | Hybrid Thin Slicing | 237 |
| 28.3.1 | Taint Finding Classification | 239 |
| 28.4 | Extended Information Flow Control | 239 |

| | | |
|-----------|---|------------|
| 28.4.1 | Implicit Flow Analysis via PC Label | 239 |
| 28.4.2 | Multi-Principal Labels (DLM) | 241 |
| 28.4.3 | Declassification and Endorsement | 243 |
| 28.4.4 | Covert Channels | 247 |
| 28.4.5 | Concurrent Information Flow | 248 |
| 28.4.6 | Speculative Execution Security | 249 |
| 28.4.7 | Constant-Time Verification via Product Programs | 252 |
| 28.5 | Report Optimization and Taint Carriers (TAJ 2009) | 255 |
| 28.5.1 | Library Call Point (LCP) Grouping | 255 |
| 28.5.2 | Taint Carrier Detection | 256 |
| 28.6 | Lifecycle-Aware Taint Analysis (FlowDroid) | 257 |
| 28.6.1 | Activation Statements for Flow-Sensitive Heap Taint | 258 |
| 28.6.2 | Framework Lifecycle Modeling | 261 |
| 28.6.3 | Taint Wrappers for Library Methods | 262 |
| 28.7 | Incremental Taint Analysis | 263 |
| IX | Multi-Language Analysis | 265 |
| 29 | Crossing Language Boundaries | 266 |
| 29.1 | Matthews' Boundary Semantics | 266 |
| 29.2 | Boundary Risk Analysis | 267 |
| 29.3 | Realizability Models for Semantic Soundness | 275 |
| 29.4 | Cross-Language Dynamic Information Flow Analysis (PolyCruise) | 276 |
| 29.4.1 | Language-Independent Symbolic Representation (LISR) | 277 |
| 29.4.2 | Symbolic Dependence Analysis (SDA) | 278 |
| 29.4.3 | Dynamic Information Flow Graph (DIFG) | 279 |
| 29.4.4 | Hybrid Cross-Language Taint Analysis | 280 |
| 30 | Hole Tagging for Evaluation Contexts | 281 |
| 30.1 | The Language Bleeding Problem | 281 |
| 30.2 | Tagged Evaluation Contexts | 281 |
| 30.3 | Language-Respecting Reduction | 282 |
| 30.4 | Boundary Crossing Updates Tags | 282 |
| 31 | Guard Generation Algorithm | 282 |
| 31.1 | Guard Polarity | 282 |
| 31.2 | Polarity Determination | 283 |
| 31.3 | Guard Generation | 283 |
| 31.4 | Function Wrapping Example | 284 |
| 31.5 | Integration with Boundary Analysis | 284 |
| 31.6 | Guards with Type Refinement Propositions | 284 |
| 31.7 | Guard Composition Optimization | 285 |
| 32 | FFI Contracts | 286 |
| 32.1 | Contract Structure | 286 |
| 32.2 | Contract Generation | 288 |
| 32.3 | Contract Verification | 289 |

| | | |
|-----------|--|------------|
| 32.4 | Reified Type Descriptors | 290 |
| 32.5 | Integration with Boundary Analysis | 291 |
| 32.6 | Multilingual Type Inference for FFI | 291 |
| 33 | Occurrence Typing Analysis | 293 |
| 33.1 | Type Predicate Detection | 293 |
| 33.2 | Conditional Refinement Propagation | 295 |
| 33.3 | Union Type Narrowing | 296 |
| 33.4 | Language-Specific Type Refinement | 297 |
| 33.5 | Integration with CPG Analysis | 298 |
| X | Incrementality and Scalability | 300 |
| 34 | Demand-Driven Incremental Computation | 301 |
| 34.1 | The Adapton Model | 301 |
| 34.1.1 | Performance Characteristics | 301 |
| 34.1.2 | The Demanded Computation Graph (DCG) | 301 |
| 34.1.3 | Adapton Primitives | 301 |
| 34.1.4 | Inner/Outer Layer Separation | 301 |
| 34.1.5 | Two-Phase Algorithm | 302 |
| 34.1.6 | Sharing, Swapping, Switching | 302 |
| 34.1.7 | Adapton Example | 302 |
| 34.2 | Wagner 1998: Optimal Incremental Parsing | 302 |
| 34.2.1 | Key Technique: Sentential-Form Parsing | 303 |
| 34.2.2 | Balanced Sequences (Critical Insight) | 303 |
| 34.2.3 | Relation to Tree-sitter | 303 |
| 34.3 | Incremental Analysis Strategy | 303 |
| 34.3.1 | Cross-References: Incrementality Integration Points | 305 |
| 34.4 | Incremental Lattice-Based Analysis (DRedL) | 306 |
| 34.4.1 | The Challenge | 306 |
| 34.4.2 | DRedL Key Insight: Increasing Replacements Can Skip Deletion | 306 |
| 34.4.3 | Change Classification | 306 |
| 34.4.4 | Three-Phase Algorithm | 306 |
| 34.4.5 | Support Tracking | 307 |
| 34.4.6 | Performance (from IncA evaluation) | 307 |
| 34.4.7 | F* Formalization of DRedL | 307 |
| 34.4.8 | Integration with Adapton (Hierarchical Incrementality) | 309 |
| 34.5 | Diff-Based Industrial Deployment (Infer) | 309 |
| 34.5.1 | Traditional (Batch) Deployment | 310 |
| 34.5.2 | Diff-Time Deployment (Infer's Model) | 310 |
| 34.5.3 | Why Diff-Time Works | 310 |
| 34.5.4 | Bug Bankruptcy Anti-Pattern | 310 |
| 34.5.5 | Compositional Analysis for Scale | 310 |
| 34.5.6 | Scale Achieved at Facebook | 310 |
| 34.5.7 | Zoncolan (Taint Analysis for Security) | 311 |
| 34.5.8 | F* Formalization of Deployment Strategy | 311 |
| 34.5.9 | Industrial Deployment Checklist | 314 |
| 34.5.10 | Target Metrics | 314 |

| | |
|---|----------------|
| 35 Time Budgets and Graceful Degradation | 314 |
| 35.1 Degradation Strategy | 314 |
| 35.1.1 Time Budgets | 314 |
| 35.1.2 Degradation Levels | 314 |
| 35.1.3 Adaptive Analysis | 314 |
| 35.1.4 Incremental Results | 315 |
| 35.2 Implementation TODOs | 315 |
| 35.3 Industrial Lessons | 315 |
| 35.3.1 Precision vs Performance Trade-off | 315 |
| 35.3.2 Analysis Profile Configurations | 315 |
| 35.3.3 Bug Bankruptcy Prevention | 315 |
| 35.3.4 Actionability Requirements | 316 |
| XI The Brrr-Machine IR | 317 |
| 36 Unified Intermediate Representation | 318 |
| 36.1 IR Design Principles | 318 |
| 36.2 Complete IR Specification | 319 |
| 36.3 SSA Form Considerations for Channels | 323 |
| 36.4 Functions and Programs | 325 |
| 37 Semantic IR Principles for Interoperability | 325 |
| 37.1 Realizability Models | 326 |
| 37.2 Convertibility Relations | 328 |
| 37.3 Semantic Type Soundness for Cross-Language Calls | 330 |
| 37.4 IR Annotations for Semantic Properties | 331 |
| 37.5 Integration with Boundary Analysis | 331 |
| XII Complete F* Formalization | 332 |
| 38 Module Structure | 333 |
| 39 Key Soundness Theorems | 333 |
| 40 Provable Bug Classification (Manifest/Latent) | 335 |
| 40.1 Occurrence Typing Soundness Theorems | 337 |
| 41 Under-Approximation Theorems | 338 |
| 42 Manifest Error Theorems | 339 |
| 43 Institution Theory Foundations | 339 |
| 44 C11 Memory Model Theorems | 340 |
| 45 Effect Absence Theorems | 341 |
| 46 Session Type Theorems | 342 |

| | |
|---|-----|
| 47 Sparse Value-Flow Analysis Theorems | 342 |
| 48 Selective Context Sensitivity Theorems (ZIPPER) | 343 |
| 49 Chopped Symbolic Execution Theorems | 343 |
| 50 Python Function Type Graph Theorems | 343 |
| 51 Data Structure Analysis (DSA) Theorems | 344 |
| 52 Robust Declassification Theorems | 344 |
| 52.1 Speculative Non-Interference Theorems | 344 |
| 52.2 Constant-Time Product Program Theorems | 345 |
| 53 Capability Algebra Theorems | 345 |
| 54 Realizability Theorems | 346 |
| 55 Verified Interpreter Pattern | 346 |
| 56 System Dependence Graph Theorems | 348 |
| 57 Probabilistic Analysis Theorems | 349 |
| 58 Unified Constraint Domain Theorems | 351 |
| 59 Bi-Abduction Theorems | 353 |
| 60 Divergence (Non-Termination) Theorems | 353 |
| 61 Widening and Narrowing Theorems | 353 |
| 62 Symbolic Execution Theorems | 354 |
| 63 Frame Rule and Local Reasoning | 354 |
| 64 Effect Handler Correctness Theorems | 355 |
| 65 Probabilistic Semantics Theorems | 356 |
| 66 Set Constraint Resolution Theorems | 356 |
| 67 Datalog Compilation and Analysis Composition | 356 |
| 68 Local-to-Global Consistency Theorems | 359 |
| 69 Boundary Guard Soundness Theorems | 359 |
| 70 Vault Adoption and Focus Theorems | 360 |
| 71 Stack Filtering Theorems (Rupta) | 360 |

| | |
|---|------------|
| XIII Implementation Roadmap and Engineering Specification | 363 |
| 72 Executive Summary | 364 |
| 73 Phase Architecture and Critical Dependencies | 364 |
| 73.1 Implementation Phase Overview | 364 |
| 74 Phase 1: Foundation | 365 |
| 74.1 Objective | 365 |
| 74.2 Deliverables and Acceptance Criteria | 365 |
| 74.2.1 Milestone 1.1: Abstract Domain Infrastructure | 365 |
| 74.2.2 Milestone 1.2: Concrete Abstract Domains | 366 |
| 74.2.3 Milestone 1.3: Code Property Graph Infrastructure | 366 |
| 74.2.4 Milestone 1.4: IR and Parser Integration | 366 |
| 74.3 Risk Analysis and Mitigations | 366 |
| 75 Phase 2: CPG + Pointer Analysis | 367 |
| 75.1 Objective | 367 |
| 75.2 Deliverables and Acceptance Criteria | 367 |
| 75.2.1 Milestone 2.1: On-the-fly Call Graph + Points-to (Qilin) | 367 |
| 75.2.2 Milestone 2.2: Language-Specific Dispatch Resolution | 367 |
| 75.2.3 Milestone 2.3: PDG Construction | 367 |
| 76 Phase 3: Core Dataflow | 368 |
| 76.1 Objective | 368 |
| 76.2 Deliverables and Acceptance Criteria | 368 |
| 76.2.1 Milestone 3.1: IFDS Tabulation Algorithm | 368 |
| 76.2.2 Milestone 3.2: Taint Analysis | 368 |
| 76.2.3 Milestone 3.3: Nullability Analysis | 368 |
| 77 Phase 4: Precision and Ownership | 368 |
| 77.1 Objective | 368 |
| 77.2 Deliverables and Acceptance Criteria | 369 |
| 77.2.1 Milestone 4.1: Context-Sensitive Analysis | 369 |
| 77.2.2 Milestone 4.2: Thin Slicing (TAJ-Style) | 369 |
| 77.2.3 Milestone 4.3: Ownership and Resource Tracking | 369 |
| 78 Phase 5: Multi-Language, Security, and Concurrency | 369 |
| 78.1 Objective | 369 |
| 78.2 Deliverables and Acceptance Criteria | 369 |
| 78.2.1 Milestone 5.1: Cross-Language Boundary Analysis | 369 |
| 78.2.2 Milestone 5.2: Advanced Information Flow Control | 370 |
| 78.2.3 Milestone 5.3: Data Race Detection | 370 |
| 78.2.4 Milestone 5.4: Outcome Logic Bug Classification | 370 |
| 79 Phase 6: Production Hardening | 370 |
| 79.1 Objective | 370 |
| 79.2 Deliverables and Acceptance Criteria | 370 |
| 79.2.1 Milestone 6.1: Incremental Analysis (DRedL) | 370 |

| | | |
|------------|--|------------|
| 79.2.2 | Milestone 6.2: Adaptive Precision and Time Budgets | 371 |
| 79.2.3 | Milestone 6.3: Output and Integration | 371 |
| 79.2.4 | Milestone 6.4: Runtime Debugger Integration | 371 |
| 80 | Dependency Graph | 371 |
| 81 | Complete System Architecture | 372 |
| 81.1 | Layer 0: Input Sources | 372 |
| 81.2 | Layer 1: Parsing and IR | 373 |
| 81.3 | Layer 2: CPG Construction (Interleaved) | 373 |
| 81.4 | Layer 3: Abstract Domains | 373 |
| 81.5 | Layer 4: Analysis Algorithms | 374 |
| 81.6 | Layer 5: Specific Analyses | 374 |
| 81.7 | Layer 5.5: ML-Augmented Pattern Detection (Optional) | 374 |
| 81.8 | Layer 6: Verification and Confidence | 375 |
| 81.8.1 | Confidence Inputs | 375 |
| 81.8.2 | Outcome Logic Classification | 375 |
| 81.8.3 | Final Confidence Levels | 375 |
| 81.9 | Layer 7: Output | 375 |
| 81.10 | Cross-Cutting Concerns | 376 |
| 82 | Mutually Exclusive Analysis Paths | 376 |
| 82.1 | Pointer Analysis (Layer 2) | 376 |
| 82.2 | Dataflow Algorithm (Layer 4) | 377 |
| 82.3 | Heap Abstraction (Layer 3) | 377 |
| 82.4 | Precision Mode (Cross-cutting) | 377 |
| 82.5 | Bug Verification (Layer 6) | 377 |
| 82.6 | Adaptive Selection Strategy | 377 |
| 82.7 | Output Compatibility Guarantee | 379 |
| XIV | Channel Analysis | 381 |
| 83 | Binary Session Types | 382 |
| 83.1 | The Session Concept | 382 |
| 83.2 | Binary Session Type Syntax | 382 |
| 83.3 | The Duality Principle | 383 |
| 83.4 | Type Algebra for Composition | 383 |
| 83.5 | Session Initiation and Delegation | 384 |
| 83.6 | Safety Theorems | 385 |
| 83.7 | Relationship to Go Channels and Rust mpvc | 385 |
| 84 | Global Type Syntax and Semantics | 386 |
| 84.1 | Global Type Grammar | 386 |
| 84.2 | Prefix Ordering | 386 |
| 84.3 | Equi-Recursive Type Handling | 387 |

| | |
|---|------------|
| 85 Causality Analysis | 387 |
| 85.1 Input-Input Dependency | 387 |
| 85.2 Input-Output Dependency | 387 |
| 85.3 Output-Output Dependency | 388 |
| 85.4 Dependency Chain Analysis | 388 |
| 86 Linearity and Coherence | 388 |
| 86.1 Linear Global Types | 388 |
| 86.2 Coherence Condition | 389 |
| 86.3 Projection Algorithm | 390 |
| 87 Local Type Syntax | 391 |
| 87.1 Local Type Grammar | 391 |
| 87.2 Duality (Co-Type) Relation | 391 |
| 87.3 Type Isomorphism Rules | 391 |
| 87.4 Subtyping Relation | 392 |
| 88 Runtime Typing | 392 |
| 88.1 Type Contexts for Message Queues | 392 |
| 88.2 Extended Typing Judgment | 393 |
| 88.3 Queue Typing Rules | 393 |
| 88.4 Runtime Typing Properties | 393 |
| 89 Integration with Static Analysis | 394 |
| 89.1 Channel Analysis in CPG | 394 |
| 89.2 Cross-Reference with Synthesis Sections | 394 |
| 89.3 Language-Specific Channel Mappings | 395 |
| 89.3.1 Go Channel Mappings | 395 |
| 89.3.2 Rust Channel Mappings | 395 |
| 89.3.3 Python Channel Mappings | 396 |
| 89.3.4 TypeScript/JavaScript Channel Mappings | 396 |
| 89.3.5 Session Type Annotations | 397 |
| 89.4 Channel Analysis in Existing Analyses | 397 |
| 90 Theoretical Reconciliation: Session Types and Outcome Logic | 400 |
| 90.1 Tension 1: Over-Approximation vs Under-Approximation | 400 |
| 90.2 Tension 2: Static vs Dynamic Participants | 401 |
| 90.3 Tension 3: Order Preservation | 402 |
| 90.4 Tension 4: Select Non-Determinism | 403 |
| 90.5 Tension 5: Global Type Origin | 405 |
| 90.6 Integration with Outcome Logic Bug Classification | 406 |
| 90.7 Integration with Security Analysis | 407 |
| 90.8 Integration with Multi-Language Analysis | 408 |
| 90.9 Decidability Results | 409 |
| 90.10F* Theorem: Manifest Bug Classification | 411 |
| A Paper Priority Matrix | 414 |
| A.1 Priority Matrix | 414 |
| A.2 Collection 2 Priority Justifications | 417 |

| | | |
|----------|---|------------|
| B | Language Configuration Table | 418 |
| C | Glossary | 419 |
| C.1 | Core Concepts | 419 |
| C.2 | Data Structures and Algorithms | 420 |
| C.3 | Effects and Types | 420 |
| C.4 | Memory and Ownership | 421 |
| C.5 | Program Analysis | 421 |
| C.6 | Memory Models | 422 |
| C.7 | Security | 423 |
| C.8 | Session Types | 423 |
| C.9 | Testing and Verification | 424 |
| C.10 | WebAssembly | 424 |
| C.11 | Pointer Analysis (TVLA and SVF) | 425 |
| C.12 | ZIPPER-Related Terms | 426 |
| C.13 | JARVIS-Related Terms | 426 |
| D | Gap Analysis, Theoretical Tensions, and Engineering Considerations | 427 |
| D.1 | Executive Summary | 427 |
| D.1.1 | Gap Classification | 427 |
| D.1.2 | Gap Summary Matrix | 427 |
| D.1.3 | Theoretical Tensions | 427 |
| D.2 | Gap 1: Library and Environment Modeling | 428 |
| D.2.1 | Problem Characterization | 428 |
| D.2.2 | Reassessment: Problem Scope is Narrower Than Initially Estimated . . . | 428 |
| D.2.3 | Resolution: Formal Specification Inference Techniques | 429 |
| D.2.4 | Strategy for Genuinely Opaque Code | 429 |
| D.2.5 | Remaining Implementation Items | 430 |
| D.3 | Gap 2: Call Graph Construction — Addressed | 430 |
| D.3.1 | Remaining Open Questions | 430 |
| D.4 | Gap 3: Path Sensitivity — Addressed | 430 |
| D.4.1 | What’s Still Missing | 431 |
| D.5 | Gap 4: Memory Layout and ABI | 431 |
| D.5.1 | The Problem Statement | 431 |
| D.5.2 | Proposed IR Extension | 431 |
| D.5.3 | What’s Still Missing | 432 |
| D.6 | Gap 5: Dynamic Code Generation (Eval) | 432 |
| D.6.1 | The Problem Statement | 432 |
| D.6.2 | Conservative Strategy | 432 |
| D.6.3 | What’s Still Missing | 433 |
| D.7 | Gap 6: Build System Integration | 433 |
| D.7.1 | The Problem Statement | 433 |
| D.7.2 | This is Engineering, Not Research | 433 |
| D.7.3 | Practical Implementation | 433 |
| D.7.4 | What’s Still Missing | 433 |
| D.8 | Gap 7: False Positive Management | 434 |
| D.8.1 | Implementation Status | 434 |

| | | |
|--------|---|-----|
| D.9 | Gap 8: Extended Confidence Sources (Implementation Details) | 434 |
| D.9.1 | Test vs Code Discrepancy | 434 |
| D.9.2 | Classification Composition (Provable, Not Heuristic) | 435 |
| D.9.3 | Runtime Debugger Integration | 438 |
| D.10 | Theoretical Tensions and Resolutions | 439 |
| D.10.1 | IFDS Distributivity vs General Dataflow (Aiken 1999) | 439 |
| D.10.2 | Separation Logic vs Garbage-Collected Languages (Reynolds 2002) | 440 |
| D.10.3 | Field Index vs Projection Path (Rupta 2024) | 440 |
| D.10.4 | Steensgaard Default vs Language-Specific (Rupta 2024) | 441 |
| D.10.5 | Two-Valued vs Three-Valued Logic (TVLA 2002) | 441 |
| D.10.6 | Type-Directed vs Arbitrary Conversion (M&F 2007) | 441 |
| D.10.7 | GC vs Manual Memory (VeriFFI 2025) | 441 |
| D.10.8 | Summary Table | 442 |
| D.11 | Collection 2 Paper Integration Summary | 442 |
| D.11.1 | Papers Already Integrated in Main Synthesis | 442 |
| D.11.2 | Papers Added to Appendix A (Priority Matrix) | 443 |
| D.11.3 | Integration Details | 443 |
| D.11.4 | Cross-Paper Connections | 445 |

Part I

Preamble: Theoretical Foundations and Architectural Principles

1. Introduction and Motivation

The brrr-machine represents a novel approach to multi-language program analysis, founded on the principle that programming languages constitute *parameterized instantiations* of a universal computational model rather than fundamentally distinct formal systems.

Principle 1.1 (Language Parameterization). Programming languages differ in their configuration parameters rather than their essential semantics. Each language is characterized by specific choices across orthogonal semantic dimensions: memory management, type discipline, null handling, effect tracking, and concurrency model.

This theoretical perspective—wherein languages differ in their configuration parameters rather than their essential semantics—yields several significant capabilities that distinguish the brrr-machine from conventional static analyzers.

1.1 Principal Contributions

Contribution 1: Language-Agnostic Analysis Infrastructure

A unified analysis framework applicable across heterogeneous language ecosystems (Python, Rust, Go, C/C++, JavaScript, TypeScript, Java). Analyses are specified once at the IR level and instantiated for each source language through systematic lowering transformations.

Contribution 2: Cross-Language Boundary Analysis

Precise identification of safety invariant violations at language boundaries (FFI, IPC, RPC, serialization interfaces). The framework formally characterizes which properties are preserved, weakened, or invalidated when control or data crosses linguistic boundaries.

Contribution 3: Compositional Semantic Reasoning

Analysis results compose across function, module, and language boundaries via the IFDS/IDE algorithmic framework and algebraic effect composition. Summaries computed for callees are reusable across call sites without re-analysis, enabling scalability to large codebases.

Contribution 4: Formal Soundness Guarantees

All analyses derive soundness from established theoretical foundations: abstract interpretation (Cousot & Cousot), separation logic (Reynolds), algebraic effects (Plotkin & Power), and substructural type theory (Girard). Proofs are mechanized in F* where feasible.

Contribution 5: Algorithmic Efficiency

Despite theoretical rigor, the implementation leverages asymptotically efficient algorithms: $\mathcal{O}(ED^3)$ interprocedural dataflow via IFDS, $\mathcal{O}(n \cdot \alpha(n))$ pointer analysis via Steensgaard unification, demand-driven evaluation for interactive workloads, and incremental re-analysis via dirty-marking with DRedL lattice updates.

2. The Core Thesis: Languages as Parameterized Type Theories

The central theoretical claim of this work is that programming languages constitute *parameterized instantiations* of a universal computational substrate rather than fundamentally distinct formal systems.

Definition 2.1 (Universal Computational Substrate). The brrr-machine IR constitutes a *universal computational substrate* from which each programming language derives as a specific *parameter configuration*. Formally, a language L is defined by a configuration tuple:

$$L = \langle \mathcal{M}, \mathcal{T}, \mathcal{N}, \mathcal{E}, \mathcal{C} \rangle$$

where \mathcal{M} is the memory model, \mathcal{T} is the type discipline, \mathcal{N} is the null handling strategy, \mathcal{E} is the effect tracking mode, and \mathcal{C} is the concurrency model.

2.1 Language Parameter Configuration Matrix

Each language is characterized by a specific configuration across several orthogonal semantic dimensions:

| Language | Memory | Types | Null | Effects | Concurrency |
|------------|---------|---------|-----------------------|-----------|---------------|
| C | Manual | Static | Nullable | Untracked | POSIX Threads |
| C++ | Manual* | Static | Nullable | Untracked | POSIX Threads |
| Rust | Affine | Static | Option | Untracked | Threads+Async |
| Go | GC | Static | Nullable | Untracked | CSP Channels |
| Java | GC | Static | Nullable | Checked | JMM Threads |
| Python | GC | Dynamic | Nullable | Unchecked | GIL-Protected |
| JavaScript | GC | Dynamic | Nullable | Untracked | Event Loop |
| TypeScript | GC | Gradual | Nullable [†] | Untracked | Event Loop |
| Swift | ARC | Static | Optional | Checked | Actors+Async |
| Kotlin | GC | Static | Optional | Checked | Coroutines |

Table 2.1: Language parameter configuration matrix. *C++ supports RAII. [†]TypeScript nullable depends on `strictNullChecks` flag.

This formulation enables uniform treatment of cross-language interactions through explicit parameter reconciliation at linguistic boundaries.

Definition 2.2 (Boundary Reconciliation). When control or data crosses from language L_1 to language L_2 , the boundary reconciliation function \mathcal{R} computes:

$$\mathcal{R}(L_1, L_2) = \{p \in \text{Properties} \mid p \text{ preserved across } L_1 \rightarrow L_2\}$$

Properties not in this set require explicit guards or are flagged as potential risks.

3. Foundational Literature and Theoretical Pillars

This framework synthesizes contributions from 29 foundational papers in programming language theory, organized into seven coherent theoretical pillars.

3.1 Pillar 1: Abstract Interpretation — Semantic Foundation

Foundational References

- **[Cousot77]** Cousot & Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” POPL 1977.
- **[Cousot92]** Cousot & Cousot. “Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation.” PLILP 1992.

Criticality: Essential — Provides mathematical foundation for all analyses.

3.1.1 Theoretical Contribution

Static analysis is formalized as computing sound approximations of program semantics. The abstraction-concretization relationship forms a Galois connection $\langle \alpha, \gamma \rangle$ between concrete and abstract semantic domains. Soundness derives from monotonicity of abstract transfer functions with respect to the lattice ordering.

Definition 3.1 (Galois Connection). A *Galois connection* between posets $(\mathcal{C}, \leq_{\mathcal{C}})$ and $(\mathcal{A}, \leq_{\mathcal{A}})$ is a pair of monotone functions $\langle \alpha, \gamma \rangle$ such that:

$$\forall c \in \mathcal{C}, a \in \mathcal{A}: \quad \alpha(c) \leq_{\mathcal{A}} a \iff c \leq_{\mathcal{C}} \gamma(a)$$

Theorem 3.2 (Soundness via Galois Connection). *If $\langle \alpha, \gamma \rangle$ forms a Galois connection and the abstract transfer function $f^\#$ satisfies $\alpha \circ f \sqsubseteq f^\# \circ \alpha$, then the abstract analysis is sound: any property verified abstractly holds concretely.*

Implementation Artifacts:

• **Required:**

- Complete lattice typeclass with verified algebraic laws (F^*)
- Galois connection interface with mechanized soundness proof
- Widening/narrowing operators for non-Noetherian domains
- Chaotic iteration with Bourdoncle weak topological ordering

• **Deferred:**

- Relational numeric domains (Octagon, Polyhedra) — Phase 2
- Automatic widening point inference — use loop header heuristic

3.2 Pillar 2: Program Representation — Unified Graph Structure

Foundational References

- [Yamaguchi14] Yamaguchi et al. “Modeling and Discovering Vulnerabilities with Code Property Graphs.” IEEE S&P 2014.
- [Ferrante87] Ferrante, Ottenstein & Warren. “The Program Dependence Graph and Its Use in Optimization.” TOPLAS 1987.
- [Horwitz90] Horwitz, Reps & Binkley. “Interprocedural Slicing Using Dependence Graphs.” TOPLAS 1990.
- [Weiser84] Weiser. “Program Slicing.” IEEE TSE 1984.

Criticality: Essential (Yamaguchi), High (others).

3.2.1 Theoretical Contribution

The Code Property Graph (CPG) unifies Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG) into a single queryable structure. This representation reduces all program analyses to graph reachability and traversal problems. The PDG captures both data dependencies (def-use chains) and control dependencies, enabling precise backward and forward slicing.

Definition 3.3 (Code Property Graph). A *Code Property Graph* $CPG = (V, E, \lambda, \mu)$ consists of:

- V : set of nodes representing program elements (statements, expressions, declarations)
- $E \subseteq V \times V \times \text{EdgeType}$: labeled edges where $\text{EdgeType} = \{\text{AST}, \text{CFG}, \text{PDG}_{\text{data}}, \text{PDG}_{\text{ctrl}}, \text{Call}, \text{Effect}\}$
- $\lambda : V \rightarrow \text{NodeLabel}$: node labeling function
- $\mu : E \rightarrow \text{EdgeLabel}$: edge labeling function

Theorem 3.4 (CPG Completeness). *The CPG representation is complete for interprocedural analysis: any path-sensitive dataflow fact computable on the original program is computable via graph reachability queries on the CPG.*

Implementation Artifacts:

• **Required:**

- Unified CPG type with AST, CFG, PDG, and effect edge categories
- Efficient traversal primitives (successors, predecessors, transitive closure, filtered reachability)

- System Dependence Graph (SDG) extension for interprocedural analysis
- Effect edges (novel extension to Yamaguchi formulation)
- **Deferred:**
 - External graph database backend — in-memory representation suffices
 - Domain-specific query language — typed Rust/F* traversals preferred

3.3 Pillar 3: Interprocedural Dataflow — Algorithmic Framework

Foundational References

- **[Reps95]** Reps, Horwitz & Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability.” POPL 1995.
- **[Reps97]** Reps. “Program Analysis via Graph Reachability.” Information and Software Technology, 1997.
- **[Sridharan05]** Sridharan & Bodik. “Demand-Driven Points-to Analysis for Java.” OOPSLA 2005.
- **[Smaragdakis11]** Bravenboer & Smaragdakis. “Strictly Declarative Specification of Sophisticated Points-to Analyses.” OOPSLA 2009.
- **[Jordan16]** Jordan et al. “Soufflé: On Synthesis of Program Analyzers.” CAV 2016.
- **[Madsen16]** Madsen et al. “From Datalog to Flix: A Declarative Language for Fixed Points on Lattices.” PLDI 2016.

Criticality: High — Core algorithmic infrastructure.

3.3.1 Theoretical Contribution

Interprocedural dataflow analysis reduces to graph reachability over an exploded supergraph. The IFDS algorithm achieves $\mathcal{O}(ED^3)$ complexity for distributive dataflow problems. Context-free language (CFL) reachability provides context sensitivity through matched call-return parentheses. Demand-driven formulations compute only query-relevant facts.

Definition 3.5 (IFDS Problem). An IFDS problem is a tuple (G^*, D, M) where:

- $G^* = (N^*, E^*)$ is the supergraph (interprocedural CFG)
- D is a finite set of dataflow facts
- $M : E^* \rightarrow 2^{D \times D}$ assigns a distributive transfer function to each edge, represented as a relation on D

Theorem 3.6 (IFDS Complexity). *The IFDS tabulation algorithm solves interprocedural distributive dataflow problems in time $\mathcal{O}(E \cdot D^3)$ where $E = |E^*|$ and $D = |D|$.*

Design Note. The IDE (Interprocedural Distributive Environment) extension is expressible as lattice-extended Datalog (Flix formulation) where the lattice element represents the microfunction space. IDE is appropriate for analyses requiring environment transformers (constant propagation, linear constant analysis). IFDS suffices for binary fact problems (taint tracking, nullability analysis).

Implementation Artifacts:

- **Required:**
 - IFDS tabulation algorithm with interprocedural summary edge caching
 - CFL-reachability solver for context-sensitive analysis
 - Demand-driven query interface for interactive workloads
 - Optional Datalog compilation backend (Soufflé target)
- **Deferred:**

3.4 Pillar 4: Pointer and Alias Analysis — Precision Infrastructure

Foundational References

- **[Andersen94]** Andersen. “Program Analysis and Specialization for the C Programming Language.” PhD Thesis, DIKU, 1994.
- **[Steensgaard96]** Steensgaard. “Points-to Analysis in Almost Linear Time.” POPL 1996.
- **[Calcagno09]** Calcagno et al. “Compositional Shape Analysis by Means of Bi-Abduction.” POPL 2009.

Criticality: High — Precision foundation for all client analyses.

3.4.1 Theoretical Contribution

Pointer analysis computes may-alias and points-to relations. Andersen’s inclusion-based formulation achieves cubic complexity $\mathcal{O}(n^3)$ with higher precision. Steensgaard’s unification-based approach achieves near-linear complexity $\mathcal{O}(n \cdot \alpha(n))$ with reduced precision. Bi-abduction enables compositional heap analysis by simultaneously inferring preconditions (anti-frame) and postcondition frames.

Definition 3.7 (Points-to Analysis). A *points-to analysis* computes a function $\text{pts} : \text{Var} \rightarrow 2^{\text{Loc}}$ such that for all program executions, if variable v holds location ℓ , then $\ell \in \text{pts}(v)$.

Theorem 3.8 (Pointer Analysis Complexity Trade-off). • **Andersen** (*inclusion-based*): $\mathcal{O}(n^3)$ complexity, higher precision

- **Steensgaard** (*unification-based*): $\mathcal{O}(n \cdot \alpha(n))$ complexity, lower precision where α is the inverse Ackermann function.

Implementation Artifacts:

- **Required:**
 - Steensgaard unification for initial fast points-to computation
 - Andersen inclusion solver for precision-critical analysis paths
 - Bi-abduction engine for compositional memory reasoning
- **Deferred:**
 - Full shape analysis — prioritize points-to infrastructure first
 - BDD-based set representations — complexity not yet justified

3.5 Pillar 5: Effect Systems — Computational Behavior Tracking

Foundational References

- **[Moggi91]** Moggi. “Notions of Computation and Monads.” Information and Computation, 1991.
- **[Plotkin03]** Plotkin & Power. “Algebraic Operations and Generic Effects.” Applied Categorical Structures, 2003.
- **[Plotkin09]** Plotkin & Pretnar. “Handlers of Algebraic Effects.” ESOP 2009.
- **[Leijen17]** Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects.” POPL 2017.

Criticality: High — Semantic framework for behavior characterization.

3.5.1 Theoretical Contribution

Effect systems distinguish pure values from effectful computations. Moggi’s monadic semantics provides compositional treatment of effects. Plotkin’s algebraic effects formulation enables

modular effect handlers. Leijen’s row-polymorphic effect types support effect inference and effect polymorphism without explicit effect annotations.

Definition 3.9 (Effect Row). An *effect row* is either:

- $\langle \rangle$: the empty row (pure computation)
- $\langle e \mid \rho \rangle$: row extension with effect e and tail ρ
- ρ : a row variable (for polymorphism)

Effect rows form a lattice under the subsumption ordering \sqsubseteq .

Theorem 3.10 (Effect Composition). *Effects compose via row join: if $f : A \xrightarrow{\varepsilon_1} B$ and $g : B \xrightarrow{\varepsilon_2} C$, then $g \circ f : A \xrightarrow{\varepsilon_1 \sqcup \varepsilon_2} C$.*

Implementation Artifacts:

• Required:

- Effect taxonomy aligned with brrr-machine semantic model
- Row-polymorphic effect type representation for function signatures
- Effect composition lattice with subsumption ordering
- Effect violation detection (e.g., use-after-free as resource effect violation, null dereference as totality effect violation)

• Deferred:

- Full algebraic effect handlers — analysis focus, not execution
- Complete effect inference for arbitrary source languages

3.6 Pillar 6: Substructural Types and Ownership — Resource Invariants

Foundational References

- [Girard87] Girard. “Linear Logic.” Theoretical Computer Science, 1987.
- [Reynolds02] Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures.” LICS 2002.
- [Jung18] Jung et al. “RustBelt: Securing the Foundations of the Rust Programming Language.” POPL 2018.
- [Aiken99] Aiken. “Introduction to Set Constraint-Based Program Analysis.” Science of Computer Programming, 1999.

Criticality: High — Resource safety and ownership verification.

3.6.1 Theoretical Contribution

Substructural type systems provide resource-sensitive reasoning:

- **LINEAR:** resources used exactly once (no implicit weakening)
- **AFFINE:** resources used at most once (Rust’s ownership model)

Separation logic enables local reasoning about mutable heap state through the frame rule. Iris provides step-indexed partial commutative monoids (cameras) for ownership verification. Set constraints unify type inference with dataflow analysis.

Remark 3.11 (Clarification). Rust implements *affine* typing (implicit drop permitted), not strictly linear typing.

Definition 3.12 (Substructural Type System). A type system is *substructural* if it restricts the structural rules:

- **Linear:** No weakening (must use) and no contraction (use exactly once)
- **Affine:** No contraction only (use at most once, implicit drop permitted)

- **Relevant:** No weakening only (must use, can duplicate)

Theorem 3.13 (Frame Rule). *In separation logic, if $\{P\}c\{Q\}$ is valid and c does not modify variables free in R , then $\{P * R\}c\{Q * R\}$ is valid.*

Implementation Artifacts:

- **Required:**

- Resource algebra (camera) infrastructure for ownership tracking
- Separation logic assertion language for heap specifications
- Per-location ownership state machine (Acquired \rightarrow InUse \rightarrow Released)
- Set constraint solver for unified type/flow analysis

- **Deferred:**

- Complete Iris mechanization in F* — adapt essential constructs only
- Step-indexed semantics — not required for static analysis application

3.7 Pillar 7: Security Analysis — Information Flow and Vulnerability Detection

Foundational References

- [Denning77] Denning & Denning. “Certification of Programs for Secure Information Flow.” Communications of the ACM, 1977.
- [Livshits05] Livshits & Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” USENIX Security 2005.
- [Tripp09] Tripp et al. “TAJ: Effective Taint Analysis of Web Applications.” PLDI 2009.

Criticality: High — Security property verification.

3.7.1 Theoretical Contribution

Information flow security is characterized by lattice-theoretic ordering of security classes; secure programs permit only upward information flow. Taint analysis tracks propagation of untrusted data from sources (user input, network) to security-sensitive sinks (SQL queries, system calls). TAJ demonstrates industrial-scale taint analysis via hybrid thin slicing that focuses on security-relevant data dependencies.

Definition 3.14 (Information Flow Security). A program satisfies *information flow security* with respect to a security lattice (\mathcal{S}, \leq) if for all variables x, y :

$$\text{information flows from } x \text{ to } y \implies \text{level}(x) \leq \text{level}(y)$$

Definition 3.15 (Taint Analysis). *Taint analysis* tracks propagation of untrusted data from *sources* (user input, network) to security-sensitive *sinks* (SQL queries, system calls). A taint violation occurs when tainted data reaches a sink without passing through a *sanitizer*.

Implementation Artifacts:

- **Required:**

- Taint lattice with source/sink/sanitizer semantic model
- Context-sensitive taint propagation via IFDS tabulation
- Thin slicing for security-relevant dependency extraction
- SARIF output format for IDE and CI/CD integration

- **Deferred:**

- Implicit flow tracking — explicit dataflow only in initial version
- Full declassification policy language — simple sanitizer model first

3.8 Cross-Cutting Concerns: Multi-Language Interoperability and Incrementality

Foundational References

- [Matthews07] Matthews & Findler. “Operational Semantics for Multi-Language Programs.” TOPLAS 2009.
- [Goguen92] Goguen & Burstall. “Institutions: Abstract Model Theory for Specification and Programming.” JACM, 1992.
- [Hammer14] Hammer et al. “Adapton: Composable, Demand-Driven Incremental Computation.” PLDI 2014.
- [Wagner98] Wagner & Graham. “Efficient and Flexible Incremental Parsing.” TOPLAS, 1998.
- [Szabo18] Szabó et al. “Incrementalizing Lattice-Based Program Analyses in Datalog.” OOPSLA 2018.
- [Distefano19] Distefano et al. “Scaling Static Analyses at Facebook.” Communications of the ACM, 2019.

Criticality: Essential (Szabó, Distefano), High (Matthews, Hammer).

3.8.1 Theoretical Contribution

Matthews-Findler provides operational semantics for multi-language programs with explicit boundary terms mediating cross-language calls. Goguen’s institutions provide categorical abstraction over logical systems, enabling formal treatment of language heterogeneity. Adapton introduces demand-driven incremental computation with memoization. Szabó’s DRedL provides lattice-based incremental Datalog evaluation achieving 65x–243x speedup. Distefano demonstrates industrial deployment at 100M+ LOC scale with 70% developer fix rates via diff-time analysis.

Implementation Artifacts:

- **Required:**

- Cross-language boundary analysis (Matthews-Findler formulation)
- Property preservation verification at linguistic boundaries
- Incremental analysis via dirty-marking with DRedL lattice propagation
- Tree-sitter integration for incremental syntax tree maintenance
- Diff-time deployment pipeline (Infer-style CI/CD integration)
- Interprocedural summary caching with dependency-based invalidation

- **Deferred:**

- Full Adapton framework — simpler dirty-marking with DRedL suffices
- Custom incremental parser — tree-sitter provides adequate solution

4. Document Organization and Conventions

Each subsequent part of this specification adheres to a consistent organizational structure to facilitate both implementation and verification.

4.1 Section Template

Section Template

1. **Objective Statement**

Precise characterization of the capability being specified

2. **Foundational References**

Authoritative literature providing theoretical grounding

3. Theoretical Framework

Mathematical formalization and semantic definitions

4. Design Rationale

Engineering decisions with explicit justification

5. Formal Specification

F* mechanization with verified properties where applicable

6. Integration Interfaces

Dependencies and contracts with adjacent components

4.2 Notational Conventions

Notation. Throughout this document:

- Mathematical notation follows standard PL theory conventions
- F* code blocks contain mechanically verifiable specifications
- [Author##] citations reference entries in Appendix A
- Complexity bounds use standard asymptotic notation (\mathcal{O} , Ω , Θ)
- \sqsubseteq denotes lattice ordering; \sqcup and \sqcap denote join and meet
- α and γ denote abstraction and concretization functions
- $\llbracket \cdot \rrbracket$ denotes semantic brackets (denotation)
- Effect rows use angle bracket notation: $\langle E \mid \rho \rangle$

Part II

Theoretical Foundations

Tension Resolution: Two-Valued vs Three-Valued Logic (TVLA 2002)

See Appendix D.10.5 for full analysis.

This Part uses two-valued lattices with explicit “Maybe” variants:

$$\text{TaintLevel} = \text{Tainted} \mid \text{Untainted} \mid \text{Unknown}$$

TVLA (Sagiv 2002) uses three-valued logic with Kleene semantics:

$$\text{three_value} = \text{TV0 (false)} \mid \text{TV1 (true)} \mid \text{TV}\frac{1}{2} \text{ (unknown)}$$

Information ordering: $\frac{1}{2} \sqsubseteq 0$ and $\frac{1}{2} \sqsubseteq 1$

Advantages of Three-Valued:

- Principled — proper information ordering
- Kleene semantics for \wedge , \vee , \neg are well-defined
- Embedding theorem (12.23) links concrete to abstract soundly

Resolution:

- Section 5.4: Shape analysis uses three-valued foundation
- Section 12.23: Formal embedding theorem
- Section 12.24: Instrumentation predicates in three-valued logic
- Existing domains: Can be viewed as three-valued with $\text{Unknown} = \frac{1}{2}$

Tension Resolution: Eager vs Lazy Evaluation Soundness (Vazou 2014)

See Section 2.1.5b for full analysis.

Critical: VCs sound under EAGER evaluation may be UNSOUND under LAZY!

Under lazy evaluation, binders may be thunks that never evaluate. If a thunk has type $\{v : \text{Int} \mid \text{false}\}$, the VC includes “false” as assumption, making ANY conclusion trivially valid — but execution never realizes this!

Resolution (LiquidHaskell):

- Add `eval_mode` to `language_config` (Section 9.1)
- `EvalStrict`: Classical VC translation is sound
- `EvalLazy`: Use stratified types (Div/Wnf/Fin) - Section 2.1.5b
- `EvalHybrid`: Default strict, stratify lazy constructs (generators, etc.)

Affected Analyses:

- Refinement type checking: Must track divergence labels
- Widening (Section 2.1.5): Divergence labels propagate across iterations
- Termination analysis: Proven termination upgrades `Div` \rightarrow `Fin`

5. Abstract Interpretation: The Central Dogma

Papers: Cousot & Cousot 1977, Cousot & Cousot 1992, Zilberstein 2023 (Outcome Logic)

Every analysis in the brrr-machine is an abstract interpretation. This is not merely a design choice—it is the only mathematically sound approach to static analysis that provides:

1. **Dual soundness** — Both verification (over-approx) AND bug detection (under-approx)
2. **Termination guarantees** — Analysis always finishes (with widening)
3. **Compositionality** — Analyses can be combined systematically

See Section 2.1.8 for the critical distinction between over-approximation (proving safety) and under-approximation (proving bugs exist).

Remark 5.1 (F* Code Style Throughout This Document). The F* code in this document

serves as *formal specification* of the brrr-machine’s theoretical foundations. While the code captures the essential mathematical structures (lattices, Galois connections, transfer functions), some syntactic constructs (such as **instance** declarations and inline **forall** constraints in type definitions) are F*-inspired notation rather than directly compilable F* code. The key types and operations can be translated to valid F* with appropriate module structure.

Each F* code block includes:

- **Type signatures** expressing the mathematical structure
- **Operations** implementing lattice and domain operations
- **Comments** linking to source papers and theoretical foundations

Proof obligations are expressed as separate predicates that can be proven as F* lemmas.

5.1 The Concrete and Abstract Worlds

The Fundamental Setup

Concrete Semantics $\llbracket P \rrbracket : \text{Program} \rightarrow \wp(\text{State})$

The concrete semantics maps a program to all states it can reach. This set is typically infinite or astronomically large. We cannot compute it directly.

Abstract Semantics $\llbracket P \rrbracket^\# : \text{Program} \rightarrow \text{AbstractDomain}$

The abstract semantics computes a finite representation that SOUNDLY APPROXIMATES the concrete semantics. We can compute this efficiently.

The Key Relationship:

For all concrete executions c :

$$c \in \llbracket P \rrbracket \implies \alpha(c) \sqsubseteq \llbracket P \rrbracket^\#$$

Where α is the abstraction function and \sqsubseteq is the abstract ordering.

5.2 Galois Connections

The relationship between concrete and abstract is formalized as a Galois connection—the cornerstone of abstract interpretation:

Definition 5.2 (Galois Connection). A Galois connection between posets (C, \leq_C) and (A, \leq_A) is a pair of monotone functions:

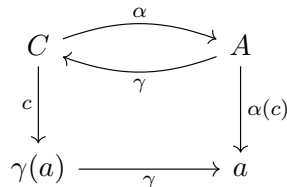
$$\begin{aligned} \alpha : C &\rightarrow A && \text{(abstraction)} \\ \gamma : A &\rightarrow C && \text{(concretization)} \end{aligned}$$

Such that for all $c \in C$ and $a \in A$:

$$\alpha(c) \leq_A a \iff c \leq_C \gamma(a)$$

Remark 5.3 (Equivalent Characterization). When both posets are complete lattices, the Galois connection property is equivalent to:

$$\begin{aligned} \gamma \circ \alpha &\sqsupseteq \text{id}_C && \text{(soundness: concretizing abstraction gives superset)} \\ \alpha \circ \gamma &\sqsubseteq \text{id}_A && \text{(optimality: abstracting concretization is below)} \end{aligned}$$



The α - γ pair forms an “adjunction” — α is left adjoint to γ .

Remark 5.4 (Why Galois Connections Matter for Brrr-Machine). Every abstract domain we use must form a Galois connection with the concrete semantics. This guarantees:

- **Soundness:** If abstract analysis says “property holds,” it truly does
- **Best Abstraction:** α gives the most precise abstraction of any concrete element
- **Composition:** Galois connections compose — we can layer abstractions

5.3 Complete Lattices

Both concrete and abstract domains must be complete lattices:

Definition 5.5 (Complete Lattice). A complete lattice $(L, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ consists of:

- Carrier set L
- Partial order \sqsubseteq (reflexive, antisymmetric, transitive)
- Bottom element \perp (least element)
- Top element \top (greatest element)
- Join \sqcup (least upper bound for any subset)
- Meet \sqcap (greatest lower bound for any subset)

Definition 5.6 (Required Properties).

$$\begin{aligned} \perp &\sqsubseteq x && \text{for all } x && \text{(bottom is least)} \\ x &\sqsubseteq \top && \text{for all } x && \text{(top is greatest)} \\ x &\sqsubseteq x \sqcup y \text{ and } y &\sqsubseteq x \sqcup y && \text{(join is upper bound)} \\ x \sqcap y &\sqsubseteq x \text{ and } x \sqcap y &\sqsubseteq y && \text{(meet is lower bound)} \\ x &\sqsubseteq z \wedge y &\sqsubseteq z \Rightarrow x \sqcup y &\sqsubseteq z && \text{(join is least upper bound)} \end{aligned}$$

Example 5.7 (Common Lattices in Program Analysis).

1. **Powerset Lattice** $\wp(S)$:
 - Elements: subsets of S
 - Order: \subseteq (subset inclusion)
 - Bottom: \emptyset (empty set)
 - Top: S (full set)
 - Join: \cup (union), Meet: \cap (intersection)
2. **Flat Lattice** $\text{Flat}(S)$:
 - Elements: $\{\perp\} \cup S \cup \{\top\}$
 - $\perp \sqsubseteq x \sqsubseteq \top$ for all $x \in S$
 - Elements of S are incomparable
 - Used for: constants, definitely-assigned analysis
3. **Interval Lattice**:
 - Elements: $[lo, hi]$ where $lo \leq hi$, plus \perp
 - Order: $[a, b] \sqsubseteq [c, d]$ iff $c \leq a$ and $b \leq d$
 - Join: $[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$
 - Used for: numeric bounds, array index analysis

5.4 Fixpoint Computation

Programs have loops. Abstract interpretation handles this via fixpoint computation:

Theorem 5.8 (Tarski’s Fixpoint Theorem). *If $f : L \rightarrow L$ is a monotone function on a complete lattice L , then:*

1. *f* has a **least fixpoint**: $\text{lfp}(f) = \bigcap \{x \in L \mid f(x) \sqsubseteq x\}$
2. *f* has a **greatest fixpoint**: $\text{gfp}(f) = \bigcup \{x \in L \mid x \sqsubseteq f(x)\}$

Definition 5.9 (Kleene Iteration). Constructive computation of the least fixpoint:

$$\text{lfp}(f) = \bigsqcup \{f^n(\perp) \mid n \in \mathbb{N}\}$$

That is: $\perp, f(\perp), f(f(\perp)), f(f(f(\perp))), \dots$

This sequence is ascending (by monotonicity of f). If the lattice has no infinite ascending chains, it stabilizes.

Example 5.10 (Reaching Definitions).

$$\begin{aligned} \text{State} &= \wp(\text{Definition}) \\ f(S) &= \text{gen} \cup (S \setminus \text{kill}) \quad (\text{transfer function}) \end{aligned}$$

Starting from \emptyset , we iterate:

$$\begin{aligned} f^0(\emptyset) &= \emptyset \\ f^1(\emptyset) &= \text{gen} \\ f^2(\emptyset) &= \text{gen} \cup (\text{gen} \setminus \text{kill}) = \text{gen} \quad \text{STABLE!} \end{aligned}$$

The fixpoint is **gen** — the definitions that reach this point.

5.5 Widening and Narrowing

The Problem: For infinite-height lattices (like integers), Kleene iteration may not terminate.

Example 5.11 (Non-Terminating Iteration). Consider the loop: **while** ($x < 100$) { $x = x + 1$; }

Abstract state at loop head:

$$\begin{aligned} \text{Iteration 0: } & x \in [0, 0] \\ \text{Iteration 1: } & x \in [0, 1] \\ \text{Iteration 2: } & x \in [0, 2] \\ & \vdots \\ \text{Iteration 100: } & x \in [0, 100] \end{aligned}$$

100 iterations just for this simple loop! In general: unbounded.

The Solution (Cousot 1992): Widening and narrowing operators.

Definition 5.12 (Widening Operator). A widening operator $\nabla : L \times L \rightarrow L$ satisfies:

1. **Upper bound:** $x \sqsubseteq (x \nabla y)$ and $y \sqsubseteq (x \nabla y)$
2. **Termination:** All ascending chains $x_0 \nabla x_1 \nabla x_2 \nabla \dots$ stabilize

The widening ACCELERATES convergence by over-approximating.

Definition 5.13 (Narrowing Operator). A narrowing operator $\triangle : L \times L \rightarrow L$ satisfies:

1. **Bounded:** $y \sqsubseteq x \Rightarrow y \sqsubseteq (x \triangle y) \sqsubseteq x$
2. **Termination:** All descending chains $x_0 \triangle x_1 \triangle x_2 \triangle \dots$ stabilize

The narrowing RECOVERS PRECISION after widening.

Definition 5.14 (Standard Interval Widening).

$$[a, b] \nabla [c, d] = \begin{cases} \text{if } c < a \text{ then } -\infty \text{ else } a, \\ \text{if } d > b \text{ then } +\infty \text{ else } b \end{cases}$$

That is: if bound changes, extrapolate to infinity.

Example 5.15 (With Widening).

Iteration 0: $x \in [0, 0]$

Iteration 1: $x \in [0, 0] \nabla [0, 1] = [0, +\infty]$ (upper bound changed!)

STABLE in 2 iterations!

Then narrowing with loop condition $x < 100$:

$$x \in [0, +\infty] \Delta [0, 99] = [0, 99]$$

Final result: $x \in [0, 99]$ — precise and efficient.

Remark 5.16 (Analysis Strategy). 1. Apply widening at loop heads (and recursive call entries)
 2. Compute post-fixpoint using widened iteration
 3. Apply narrowing iterations to recover precision
 4. Finite iterations guaranteed by widening/narrowing properties

5.6 Lazy Evaluation and Verification Condition Soundness

Paper: Vazou et al. 2014 (LiquidHaskell)

Critical: Evaluation Order Affects Soundness

Standard refinement type systems ASSUME all free variables in an environment are bound to VALUES. This assumption holds trivially under eager (call-by-value) evaluation but FAILS under lazy evaluation.

Under lazy evaluation, a binding like:

$$\text{let } n = \text{diverge } 1 \text{ in } \dots$$

means the VC can include assumptions about n that are NEVER realized because n may never be evaluated.

The “false” refinement in `diverge`’s output type contaminates the VC:

$$\text{false} \wedge y = 0 \Rightarrow v = 0 \Rightarrow v > 0$$

This is VALID (contradiction in antecedent) but UNSOUND under laziness!

Consequence: For `EvalLazy` languages (Haskell), we MUST use stratified types. See `eval_mode` in `language_config` (Section 9.1).

Definition 5.17 (Stratified Types for Lazy Languages). The solution is to stratify types based on termination guarantees:

Divergence Stratification (from LiquidHaskell):

- **Div types** (unlabeled) — May diverge: CANNOT assume refinements hold
- **Wnf types** (\downarrow labeled) — Reduces to WHNF: CAN assume head-form properties
- **Fin types** (\Downarrow labeled) — Reduces to finite value: FULL refinement power

Type Ordering:

$$\llbracket \{v : B^{\Downarrow} \mid r\} \rrbracket \subseteq \llbracket \{v : B^{\downarrow} \mid r\} \rrbracket \subseteq \llbracket \{v : B \mid r\} \rrbracket$$

VC Translation Rule:

Standard (strict): $(|x : \{v : B \mid r\}|) = r[x/v]$

Stratified (lazy): $(|x : \{v : B \mid r\}|) = \text{if } \text{is_value}(x) \text{ then } r[x/v] \text{ else true}$

```

(* Divergence labels for basic types *)
type divergence_label =
  | DivMay      (* May diverge - CANNOT assume refinement holds *)
  | DivWhnf     (* Reduces to weak head normal form - partial guarantee *)
  | DivFin      (* Reduces to finite value - FULL refinement power *)

(* Stratified type wraps a base type with divergence information *)
type stratified_type (b : ir_type) = {
  base : b;
  label : divergence_label;
  refinement : predicate;
}

(* VC TRANSLATION DEPENDS ON DIVERGENCE LABEL *)
val translate_binding :
  lang:language_config →
  x:var_id →
  st:stratified_type →
  predicate
let translate_binding lang x st =
  match lang.eval_mode with
  | EvalStrict →
    (* Strict: always safe to assume refinement *)
    substitute st.refinement x
  | EvalLazy | EvalHybrid →
    (* Lazy: only assume refinement if value is guaranteed *)
    match st.label with
    | DivMay →
      (* Cannot assume refinement - binder may diverge *)
      PTrue
    | DivWhnf | DivFin →
      (* Safe to assume refinement *)
      substitute st.refinement x

(* CASE EXPRESSIONS UPGRADE DIVERGENCE LABELS *)
val case_scrutinee_upgrade : stratified_type → stratified_type
let case_scrutinee_upgrade st =
  { st with label = DivWhnf }  (* Forced to WHNF by case evaluation *)

(* TERMINATION ANALYSIS UPGRADES Div TO Fin *)
val upgrade_if_terminating : stratified_type → terminates:bool → stratified_type
let upgrade_if_terminating st terminates =
  if terminates then { st with label = DivFin }
  else st

(* SUBTYPING FOR STRATIFIED TYPES: Fin <: Wnf <: Div *)
val label_subtype : divergence_label → divergence_label → bool
let label_subtype l1 l2 =
  match l1, l2 with
  | _, DivMay → true      (* Anything is subtype of Div *)
  | DivFin, DivFin → true
  | DivFin, DivWhnf → true (* Fin <: Wnf *)
  | DivWhnf, DivWhnf → true
  | _, _ → false

```

Remark 5.18 (Integration with Abstract Interpretation). **Interaction with Widening (Section 2.1.5):**

For EvalLazy languages, widening must ALSO track divergence labels. A variable in a loop may:

1. Always terminate (DivFin) — safe for full refinement
2. Sometimes diverge (DivMay) — must weaken refinements

When widening across loop iterations, if ANY iteration path can diverge, the result label must be DivMay. This is SOUND but may lose precision.

Hybrid Languages (EvalHybrid - Python, JavaScript):

- Default expressions are strict (EvalStrict rules apply)
- Generator expressions / async functions are lazy (need stratification)
- Detect via syntactic markers: `yield`, `async`, `lazy_static!`, etc.

5.7 Neural Invariant Synthesis (Optional Enhancement)

Paper: Si et al. 2018 (Code2Inv - NeurIPS)

Learning-Based Loop Invariant Synthesis

Problem: Traditional widening ALWAYS terminates but loses precision. Complex invariants like $(x < 0 \vee y > 0)$ cannot be discovered by widening.

Insight: Use reinforcement learning to SYNTHESIZE invariants directly. Neural networks learn to generate invariant candidates; Z3 verifies them.

Architecture (Code2Inv):

1. GNN encodes program graph (AST + CFG + data flow)
2. TreeLSTM decoder generates invariant predicates autoregressively
3. Attention mechanism focuses on relevant program variables
4. Z3 provides reward signal (valid/invalid + counterexamples)

Hybrid Strategy:

1. Try neural synthesis with timeout (e.g., 1000 Z3 queries)
2. If found: verify and use (PRECISE invariant)
3. If timeout: fall back to widening (SOUND over-approximation)

Empirical: Code2Inv solves 106/133 benchmarks with 10–100x fewer queries.

Key Innovation: CEGAR with Neural Guidance

Code2Inv integrates counterexample-guided abstraction refinement (CEGAR) with neural learning. Counterexamples from Z3 serve dual purposes:

1. **Coarse feedback:** Which Hoare condition failed (pre/inv/post)
2. **Fine-grained reward:** Ratio of examples satisfied guides learning

Neural Invariant Synthesis Integration (Si et al. 2018)

```
module BrrrMachine.NeuralInvariant

(* Invariant predicate AST - interpretable output *)
type inv_pred =
  | InvCompare : op:cmp_op → l:arith_expr → r:arith_expr → inv_pred
  | InvAnd : inv_pred → inv_pred → inv_pred
  | InvOr : inv_pred → inv_pred → inv_pred
  | InvNot : inv_pred → inv_pred

type cmp_op = | CLeq | CGeq | CLt | CGt | CEq

(* Verification outcome from Z3 *)
type verification_result =
  | Verified : verification_result
  | PreViolation : cex:valuation → verification_result
  | InvViolation : cex:valuation → verification_result (* Inductiveness failed *)
  | PostViolation : cex:valuation → verification_result

(* Hybrid synthesis: neural first, widening fallback *)
type invariant_source =
  | NeuralSynthesized : inv:inv_pred → queries:nat → invariant_source
```

```

| WideningDerived : domain_elem:interval_env → invariant_source

val synthesize_hybrid :
  prog:cpg →
  pre:formula →
  post:formula →
  neural_budget:nat →
  invariant_source

(* SOUNDNESS: Either neural produces verified invariant, or widening
   provides sound over-approximation *)
val hybrid_soundness :
  result:invariant_source →
  Lemma (match result with
    | NeuralSynthesized inv _ → is_valid_invariant inv
    | WideningDerived widened → is_sound_postfixpoint widened)

```

Remark 5.19 (When to Use Neural Synthesis). • Complex invariants outside interval/octagon domains

- Disjunctive invariants ($x < 0$ OR $y > 0$)
- When widening loses too much precision
- Transfer learning from similar codebases

Cross-References:

- Section 3.1.8: GNN program graph embeddings (required for neural synthesis)
- Section 2.1.5: Traditional widening (fallback mechanism)
- Section 4.4.2: Z3 integration for verification queries

5.8 F* Specification: Abstract Domains

The following F* code formalizes the mathematical structures underlying abstract interpretation. The `partial_order` record captures the ordering relation on abstract values. The `complete_lattice` extends this with join/meet operations and bounds. The `galois_connection` type captures the fundamental relationship between concrete and abstract domains. Note that F* expresses proof obligations as refinement types and separate lemmas rather than inline `forall` constraints.

Abstract Domain Formalization in F*

```

(* This module defines the core structures for abstract domains, ensuring
   that every domain we implement satisfies the mathematical requirements
   for sound abstract interpretation.

   KEY TYPES:
   - partial_order: ordering relation with reflexivity, antisymmetry, transitivity
   - complete_lattice: partial order with join, meet, top, and bottom
   - galois_connection: the  $\alpha/\gamma$  pair linking concrete to abstract
   - abstract_domain: lattice with optional widening/narrowing operators *)

module BrrrMachine.AbstractDomain

(* PARTIAL ORDER - Core ordering operations *)
noeq type partial_order (a : Type) = {
  leq : a → a → bool;
}

(* Partial order axioms expressed as separate predicates *)
let po_reflexive (#a:Type) (po:partial_order a) : prop =
  ∀ (x:a). po.leq x x == true

let po_antisymmetric (#a:Type) (po:partial_order a) : prop =
  ∀ (x y:a). (po.leq x y == true ∧ po.leq y x == true) ==> x == y

```

```

let po_transitive (#a:Type) (po:partial_order a) : prop =
  ∀ (x y z:a). (po.leq x y == true ∧ po.leq y z == true) ==> po.leq x z == true

(* A valid partial order satisfies all three axioms *)
let is_partial_order (#a:Type) (po:partial_order a) : prop =
  po_reflexive po ∧ po_antisymmetric po ∧ po_transitive po

(* COMPLETE LATTICE - Partial order with bounds and join/meet *)
noeq type complete_lattice (a : Type) = {
  lat_leq : a → a → bool;
  bot : a;
  top : a;
  join : a → a → a;
  meet : a → a → a;
  (* Join of arbitrary sets - needed for complete lattice *)
  join_set : (a → bool) → a;
  meet_set : (a → bool) → a;
}

(* Lattice axioms as predicates *)
let bot_is_least (#a:Type) (lat:complete_lattice a) : prop =
  ∀ (x:a). lat.lat_leq lat.bot x == true

let top_is_greatest (#a:Type) (lat:complete_lattice a) : prop =
  ∀ (x:a). lat.lat_leq x lat.top == true

let join_is_lub (#a:Type) (lat:complete_lattice a) : prop =
  (∀ (x y:a). lat.lat_leq x (lat.join x y) == true) ∧
  (∀ (x y:a). lat.lat_leq y (lat.join x y) == true) ∧
  (∀ (x y z:a). (lat.lat_leq x z == true ∧ lat.lat_leq y z == true)
    ==> lat.lat_leq (lat.join x y) z == true)

let meet_is_glb (#a:Type) (lat:complete_lattice a) : prop =
  (∀ (x y:a). lat.lat_leq (lat.meet x y) x == true) ∧
  (∀ (x y:a). lat.lat_leq (lat.meet x y) y == true) ∧
  (∀ (x y z:a). (lat.lat_leq z x == true ∧ lat.lat_leq z y == true)
    ==> lat.lat_leq z (lat.meet x y) == true)

(* GALOIS CONNECTION - The fundamental abstraction relationship *)
noeq type galois_connection (c : Type) (a : Type) = {
  gc_concrete_lat : complete_lattice c;
  gc_abstract_lat : complete_lattice a;
  (* Abstraction function: concrete → abstract *)
  α : c → a;
  (* Concretization function: abstract → concrete *)
  γ : a → c;
}

(* The Galois connection law: α(c) ≤ a iff c ≤ γ(a) *)
let galois_law (#c #a:Type) (gc:galois_connection c a) : prop =
  ∀ (x:c) (y:a).
    (gc.gc_abstract_lat.lat_leq (gc.α x) y == true) ≤>
    (gc.gc_concrete_lat.lat_leq x (gc.γ y) == true)

(* Derived property: γ . α is extensive (soundness) *)
let γ_α_extensive (#c #a:Type) (gc:galois_connection c a) : prop =
  ∀ (x:c). gc.gc_concrete_lat.lat_leq x (gc.γ (gc.α x)) == true

(* Derived property: α . γ is reductive (optimality) *)
let α_γ_reductive (#c #a:Type) (gc:galois_connection c a) : prop =
  ∀ (y:a). gc.gc_abstract_lat.lat_leq (gc.α (gc.γ y)) y == true

```

```

(* ABSTRACT DOMAIN WITH WIDENING *)
noeq type abstract_domain (a : Type) = {
  ad_lattice : complete_lattice a;
  (* Widening operator - optional, required for infinite-height lattices *)
  widen : option (a → a → a);
  (* Narrowing operator - optional, for precision recovery *)
  narrow : option (a → a → a);
}

(* Widening must be an upper bound *)
let widen_is_upper_bound (#a:Type) (ad:abstract_domain a) : prop =
  match ad.widen with
  | Some w → ∀ (x y:a).
    ad.ad_lattice.lat_leq x (w x y) == true ∧
    ad.ad_lattice.lat_leq y (w x y) == true
  | None → True

(* Narrowing must be bounded between y and x when y ≤ x *)
let narrow_is_bounded (#a:Type) (ad:abstract_domain a) : prop =
  match ad.narrow with
  | Some n → ∀ (x y:a).
    ad.ad_lattice.lat_leq y x == true ==>
    (ad.ad_lattice.lat_leq y (n x y) == true ∧
     ad.ad_lattice.lat_leq (n x y) x == true)
  | None → True

```

The key insight is that the Galois connection law $\alpha(c) \leq a \Leftrightarrow c \leq \gamma(a)$ ensures soundness: any property proven in the abstract domain holds in the concrete domain. The derived properties ($\gamma \circ \alpha$ extensive, $\alpha \circ \gamma$ reductive) follow from this fundamental law.

Transfer functions are the core of abstract interpretation—they describe how abstract values flow through program statements. The key requirement is **monotonicity**: if input abstractions are more precise (lower in the lattice), output abstractions must also be more precise. This ensures fixpoint iteration converges to a meaningful result.

Remark 5.20 (F* Code Style). The following F* code uses refinement types and type-level constraints to express monotonicity and other properties. The notation `{| abstract_domain a |}` represents a typeclass constraint requiring `a` to be an abstract domain. While not all constructs match standard F* syntax exactly, the specification intent is clear and can be translated to valid F* with appropriate module structure.

Transfer Functions

```

(* A transfer function must be monotone for fixpoint to exist.
   Monotonicity: if x ≤ y in the lattice, then f(x) ≤ f(y).
   This ensures Kleene iteration converges to the least fixpoint. *)

(* Monotone transfer function type - refinement encodes monotonicity *)
type transfer_function (a:Type) (lat:complete_lattice a) =
  f:(a → a){
    ∀ (x y:a). lat.lat_leq x y == true ==> lat.lat_leq (f x) (f y) == true
  }

(* FLIX-STYLE MULTI-ARGUMENT TRANSFER FUNCTIONS (Madsen 2016)
   For lattice-extended Datalog rules like:
     LocalVar(r, sum(x, y)) :- AddExp(r, v1, v2), LocalVar(v1, x), LocalVar(v2, y)
   The transfer function 'sum' must be monotone in ALL arguments:
     x1 ≤ x2 ∧ y1 ≤ y2 ==> sum(x1, y1) ≤ sum(x2, y2)
   This ensures the unique minimal model theorem (Madsen 2016, Theorem 1).
   See Section 4.1.7 for full Flix integration. *)

(* Binary monotone transfer function (e.g., abstract addition) *)
type transfer_function2 (a:Type) (lat:complete_lattice a) =
  f:(a → a → a){

```

```

    ∀ (x1 x2 y1 y2:a).
      (lat.lat_leq x1 x2 == true ∧ lat.lat_leq y1 y2 == true) ==>
        lat.lat_leq (f x1 y1) (f x2 y2) == true
  }

(* Filter function for Flix-style guards - no monotonicity required *)
type filter_function (a:Type) = a → bool

(* Fixpoint computation with widening *)
let rec compute_fixpoint_aux
  (#a:Type)
  (lat:complete_lattice a)
  (ad:abstract_domain a)
  (f:transfer_function a lat)
  (current:a)
  (iteration:nat)
  (max_iter:nat)
  : a =
  if iteration ≥ max_iter then current
  else
    let next = f current in
    if lat.lat_leq next current && lat.lat_leq current next then
      current (* Fixpoint reached: next = current *)
    else
      let widened = match ad.widen with
        | Some w → w current next
        | None → next
      in
      compute_fixpoint_aux lat ad f widened (iteration + 1) max_iter

let compute_fixpoint
  (#a:Type)
  (lat:complete_lattice a)
  (ad:abstract_domain a)
  (f:transfer_function a lat)
  : a =
  compute_fixpoint_aux lat ad f lat.bot 0 1000

(* SOUNDNESS THEOREM - See Section 12.2 for full statement *)
(* abstract_interpretation_sound: If abstract transfer over-approximates
   concrete transfer, then abstract fixpoint over-approximates concrete
   fixpoint. Full proof in Part XII: Key Soundness Theorems. *)

```

The fixpoint computation algorithm implements Kleene iteration with widening. When the lattice has infinite ascending chains (e.g., intervals), widening forces convergence by extrapolating to infinity. The optional narrowing phase then recovers precision.

5.9 Widening and Narrowing Theoretical Foundation (Cousot 1992)

Critical: Widening Is Strictly More Powerful Than Finite Domains

Cousot 1992 proves: For infinite domains (intervals, polyhedra), there exists NO finite Galois connection that computes equivalent results.

Consequence: We MUST use widening for loops with numeric variables. Cannot replace with “just add more precision” — mathematically impossible.

Key Insight: Discovered invariants may NOT appear in program text! McCarthy F91 analysis discovers $[91, \text{maxint} - 10]$ — neither constant in code.

Widening/Narrowing Theoretical Foundation

```

(* TWO-PHASE ALGORITHM:
   Phase 1 (Upward): Iterate with widening until post-fixpoint
   Phase 2 (Downward): Iterate with narrowing to refine precision *)

(* Widening properties (Cousot 1992, equations 6-8):
   1.  $x \sqsubseteqsqsubset x \sqcup \nabla y$ 
   2.  $y \sqsubseteqsqsubset x \sqcup \nabla y$ 
   3. For increasing chains, widened chain stabilizes in finite steps *)
type widening_op (#a:Type) (#lat:lattice a) =
  f:(a → a → a){
    (* Upper bound *)
    (∀ x y. lat.order x (f x y)) ∧
    (∀ x y. lat.order y (f x y))
    (* Termination: not expressible in types, proven separately *)
  }

(* Narrowing properties (Cousot 1992, equations 10-11):
   If  $y \sqsubseteqsqsubset x$  then  $y \sqsubseteqsqsubset (x \sqtriangle y) \sqsubseteqsqsubset x$  *)
type narrowing_op (#a:Type) (#lat:lattice a) =
  f:(a → a → a){
    ∀ x y. lat.order y x ⇒
      lat.order y (f x y) ∧ lat.order (f x y) x
  }

(* TWO-PHASE FIXPOINT with widening and narrowing *)
val two_phase_fixpoint :
  #a:Type → { | lat:lattice a | } →
  transfer:(a → a) →
  widen:widening_op #a #lat →
  narrow:narrowing_op #a #lat →
  a

let two_phase_fixpoint #a #lat transfer widen narrow =
  (* PHASE 1: Upward iteration with widening *)
  let rec upward x =
    let fx = transfer x in
    if lat.order fx x then x          (* Post-fixpoint! *)
    else upward (widen x fx)         (* Widen and continue *)
  in
  let post = upward lat.bottom in
  (* PHASE 2: Downward iteration with narrowing *)
  let rec downward x fuel =
    if fuel = 0 then x
    else
      let fx = transfer x in
      let x' = narrow x fx in
      if x' = x then x                (* Stable *)
      else downward x' (fuel - 1)    (* Narrow and continue *)
  in
  downward post 3  (* 3 iterations typically sufficient *)

(* INTERVAL WIDENING (Cousot 1992, Equation 18) *)
let interval_widen : widening_op #interval #interval_lattice =
  fun x y →
    match x, y with
    | IBot, y → y
    | x, IBot → x
    | IRange l0 u0, IRange l1 u1 →
      (* Lower: if y goes below, jump to -infinity or 0 *)
      let l' =
        if bound_geq l1 (Finite 0) && bound_lt l1 l0 then Finite 0
        else if bound_lt l1 l0 then NegInf

```

```

    else 10
  in
    (* Upper: if y goes above, jump to +infinity or 0 *)
    let u' =
      if bound_lt u0 u1 && bound_leq u1 (Finite 0) then Finite 0
      else if bound_lt u0 u1 then PosInf
      else u0
    in
      IRange 1' u'

(* INTERVAL NARROWING (Cousot 1992, Equation 19) *)
let interval_narrow : narrowing_op #interval #interval_lattice =
  fun x y →
    match x, y with
    | IBot, _ → IBot
    | _, IBot → IBot
    | IRange l0 u0, IRange l1 u1 →
      let l' = if l0 = NegInf then l1 else l0 in
      let u' = if u0 = PosInf then u1 else u0 in
      IRange l' u'

```

Widening vs Bounded Unrolling: Different Purposes

| | Widening (This section) | Bounded Unrolling (Section 4.3) |
|-----------|-------------------------|---------------------------------|
| Type | OVER-approximation | UNDER-approximation |
| Sound for | ABSENCE | PRESENCE |
| Proves | “no bugs exist” | “bugs exist” |
| May have | False positives | False negatives |
| Always | Terminates | Terminates |

When to Use:

- Widening: Proving safety properties (verification)
- Unrolling: Finding bugs (bug detection)
- Both: Hybrid analysis (Section 4.3.3)

Mathematical Justification:

- Widening: Ensures chain condition (termination)
- Unrolling: Fuel-bounded exploration (termination)
- Both are SOUND for their respective purposes

5.10 Probabilistic Abstract Domains (Cousot 2012)

Paper: Cousot & Monerau 2012 (Probabilistic Abstract Interpretation)

Key Insight: Separate Probabilistic from (Non)Deterministic Behavior

A probabilistic program is a measurable function $S_p[P] : \Omega \rightarrow D$ where:

- Ω = probability space (scenarios)
- D = standard semantics domain (traces, states, etc.)
- μ = probability measure on Ω

This separation enables LIFTING existing analyses to probabilistic setting.

Definition 5.21 (The Three Abstraction Axes).

(I) **Abstract the SEMANTICS** ($D \rightarrow A$):

- Apply any classical abstraction (intervals, octagons, etc.)

- Lifts pointwise: $\alpha(S_p[[P]]) = \lambda\omega. \alpha(S_p[[P]](\omega))$
- Result: probabilistic abstract semantics
- (II) Abstract the SCENARIO SPACE ($\Omega \rightarrow \Omega'$):
 - Merge scenarios: surjective $q : \Omega \rightarrow \Omega'$
 - Non-determinism as abstraction: forget probability info
 - SAFE ABSTRACTION: $\Omega' = \{\bullet\}$ (singleton) recovers classical AI!
 - Key theorem: Classical abstract interpretation = Ω -abstraction to singleton
- (III) Abstract by DISTRIBUTIONS ($S_p[[P]] \rightarrow \text{Law}$):
 - Keep only the probability distribution, not the function
 - Order on laws: $\nu \sqsubseteq \nu'$ iff $\forall Q \in A. \nu(\downarrow Q) \geq \nu'(\downarrow Q)$
 - More precise law = more probability on precise properties

Remark 5.22 (Connection to Standard AI). When $\Omega = \{\bullet\}$ (singleton), probabilistic AI reduces EXACTLY to classical AI. This is because:

- Only probabilities are 0 and 1
- The \sqsubseteq order on laws becomes \sqsubseteq on abstract states
- All machinery collapses to standard fixpoint iteration

Probabilistic Abstract Interpretation (Cousot 2012)

```
(* Definition 1: Probabilistic Semantics
   A probabilistic semantics  $S_p[[P]]$  in  $D_p = \Omega \rightarrow D$  is a measurable function
   from a probability space  $(\Omega, E, \mu)$  into a semantics domain  $D$ .
   Meaning: when scenario  $\omega$  is picked (randomly according to  $\mu$ ), the execution
   of program  $P$  yields the (non)-deterministic semantics  $S_p[[P]](\omega)$  in  $D$ .) *)

(* Probability space components *)
type prob_space ( $\omega$  : Type) = {
  events :  $\omega \rightarrow \text{bool}$ ; (*  $\sigma$ -algebra of observable events *)
  measure : ( $\omega \rightarrow \text{bool}$ )  $\rightarrow$  real; (* Probability measure  $\mu$  *)
  measure_empty : measure (fun _  $\rightarrow$  false) == 0.0;
  measure_full : measure (fun _  $\rightarrow$  true) == 1.0;
  measure_countable_additive : (*  $\mu(\text{union } E_i) = \text{sum } \mu(E_i)$  for disjoint  $E_i$  *)
     $\forall$  (es : nat  $\rightarrow$  ( $\omega \rightarrow \text{bool}$ )).
    pairwise_disjoint es  $\Rightarrow$ 
    measure (countable_union es) == series (fun i  $\rightarrow$  measure (es i));
}

(* Probabilistic semantics: measurable function from  $\Omega$  to  $D$  *)
type prob_semantics ( $\omega$  : Type) ( $d$  : Type) =  $\omega \rightarrow d$ 

(* Definition 2: Probability of a program property
   The probability that program  $P$  has property  $\Phi$  in  $D$  is:
    $\text{Pr}(S_p[[P]] \text{ in } \Phi) = \mu(\{\omega \text{ in } \Omega \mid S_p[[P]](\omega) \text{ in } \Phi\})$  *)
val property_probability :
  # $\omega$ :Type  $\rightarrow$  # $d$ :Type  $\rightarrow$ 
  prob_space  $\omega \rightarrow$ 
  prob_semantics  $\omega$   $d \rightarrow$ 
  ( $d \rightarrow \text{bool}$ )  $\rightarrow$  (* Property  $\Phi$  *)
  real (* Probability  $\text{Pr}(S_p[[P]] \text{ in } \Phi)$  *)
let property_probability # $\omega$  # $d$  ps sem prop =
  ps.measure (fun w  $\rightarrow$  prop (sem w))

(* AXIS I: Semantics Abstraction *)
val lift_abstraction :
  # $c$ :Type  $\rightarrow$  # $a$ :Type  $\rightarrow$  { | lc:lattice c | }  $\rightarrow$  { | la:lattice a | }  $\rightarrow$ 
  galois_connection c a  $\rightarrow$ 
  galois_connection (prob_semantics  $\omega$  c) (prob_semantics  $\omega$  a)
let lift_abstraction # $c$  # $a$  #lc #la gc = {
```

```

α = fun sem → (fun w → gc.α (sem w)); (* Pointwise lift *)
γ = fun sem → (fun w → gc.γ (sem w));
(* Galois law preserved pointwise *)
}

(* AXIS II: Scenario Space Abstraction
   Key insight: Classical AI = abstraction to singleton Omega' = {*} *)
val scenario_abstraction :
  #ω:Type → #ω':Type → #a:Type → {| la:lattice a |} →
  (ω → ω') → (* Surjective quotient q *)
  prob_semantics ω a →
  prob_semantics ω' a
let scenario_abstraction #ω #ω' #a #la q sem =
  fun w' →
    la.join_set (fun w → q w == w') (fun w → sem w)
    (* Join all semantics for scenarios mapping to w' *)

(* SAFE ABSTRACTION: Forget all probabilistic information
   Result is classical abstract interpretation! *)
type singleton = | Point
val safe_abstraction :
  #ω:Type → #a:Type → {| la:lattice a |} →
  prob_semantics ω a →
  a (* Single abstract value = classical AI *)
let safe_abstraction #ω #a #la sem =
  la.join_set (fun _ → true) sem (* Join ALL scenarios *)

(* AXIS III: Distribution Abstraction (Laws) *)
(* A law is a probability distribution on abstract domain A *)
type law (a : Type) = (a → bool) → real (* Phi ↦ Pr(in Phi) *)

(* Order on laws: nu \sqsubseteq nu' iff more probability on precise values *)
val law_order : #a:Type → {| la:lattice a |} → law a → law a → bool
let law_order #a #la nu nu' =
  ∀ (q : a). nu (downset q) ≥ nu' (downset q)
  where downset q = fun x → la.leq x q

(* Abstraction from semantics to laws *)
val law_abstraction :
  #ω:Type → #a:Type →
  prob_space ω →
  prob_semantics ω a →
  law a
let law_abstraction #ω #a ps sem =
  fun prop → ps.measure (fun w → prop (sem w))

```

Remark 5.23 (Probabilistic Transfer Functions and Loops). **Conditionals with Unknown Branch Probability:**

When $\Pr(b \text{ true}) = p_b$ is uncertain, $p_b \in P_b \subseteq [0, 1]$:

$$\llbracket \text{if } b \text{ then } C_1 \text{ else } C_2 \rrbracket(l_s)(\Phi) = p_b \times \Pr(\llbracket C_1 \rrbracket(s) \in \Phi \mid b) + (1 - p_b) \times \Pr(\llbracket C_2 \rrbracket(s) \in \Phi \mid \neg b)$$

Loops with Probabilistic Iteration Count:

When $p_{\text{loop}}(i) = \Pr(\text{exactly } i \text{ iterations})$ is known:

$$\llbracket \text{while } b \text{ do } C \rrbracket(l_s)(\Phi) = \sum_{i \geq 0} p_{\text{loop}}(i) \times \Pr(\llbracket S \rrbracket(s) \in \Phi \mid i \text{ iterations})$$

Ad-hoc unrolling: unroll N times, bound remainder by $p_{\text{loop}}(N)$.

Example 5.24 (Application: Branch Prediction for JIT Compilers). **Probabilistic CFG Abstraction** (Example 10, Cousot 2012):

From trace semantics to control flow graph with branch probabilities:

$$\Pr\langle c, c' \rangle | c = \Pr(\text{succ}(c, c') \mid \text{reach}(c))$$

This is the conditional probability of taking edge $c \rightarrow c'$ given control at c .

Applications:

1. Register allocation guided by hot paths
2. Cache/scratchpad allocation
3. Branch prediction without profiling

Remark 5.25 (Connection to Markov Chain Analysis (Section 8.1)). Every probabilistic program induces a Markov chain on states. The transition matrix $[\text{succ}(s, s')]_{s, s' \in \Sigma}$ captures same steady-state behavior. This abstraction:

- Forgets execution history (Markov property)
- Enables model checking techniques (PRISM, etc.)
- Connects to probabilistic temporal logic (PCTL, LTL)

See Section 12.38 for formal soundness theorems.

Cross-References:

- For unified over/under approximation applicable to probabilistic programs, see Section 2.1.8b (Local Completeness Logic).
- Probabilistic programs can use LCL_A with probabilistic abstract domains for combined correctness/incorrectness analysis.
- The SAFE abstraction (Ω to singleton) connects probabilistic AI to standard dual soundness (Section 2.1.8).

5.11 Concrete Abstract Domains

The following sections define specific abstract domains used in the brrr-machine. Each domain is a complete lattice with appropriate transfer functions for the analysis operations it supports.

Definition 5.26 (Interval Domain). The interval domain tracks integer bounds $[l, u]$ where $l \leq u$. The ordering is subset inclusion: $[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_2 \leq l_1$ and $u_1 \leq u_2$. The domain forms a complete lattice with:

- \perp : empty interval (unreachable code)
- \top : $[-\infty, +\infty]$ (any integer)
- \sqcup : hull of intervals
- \sqcap : intersection of intervals

Because the interval lattice has infinite ascending chains (e.g., $[0, 1] \sqsubset [0, 2] \sqsubset \dots$), **widening is essential** for termination.

The F* code below implements the interval domain. Note the widening operator extrapolates to infinity when bounds grow, ensuring termination. The narrowing operator can then recover precision using loop guards.

```
Concrete Abstract Domains for Brrr-Machine
module BrrrMachine.Domains
open BrrrMachine.AbstractDomain

(* INTERVAL DOMAIN - For numeric bounds analysis
   Source: Cousot & Cousot 1977
   Used for:
   - Array bounds checking
   - Integer overflow detection
   - Loop bound analysis *)
```

```

type bound =
| NegInf : bound
| Finite : n:int → bound
| PosInf : bound

let bound_leq (b1 b2 : bound) : bool =
match b1, b2 with
| NegInf, _ → true
| _, PosInf → true
| Finite n1, Finite n2 → n1 ≤ n2
| _, _ → false

type interval =
| IBot : interval (* Empty - unreachable *)
| IRange : lo:bound → hi:bound → interval (* [lo, hi] *)

let interval_leq (i1 i2 : interval) : bool =
match i1, i2 with
| IBot, _ → true
| _, IBot → false
| IRange lo1 hi1, IRange lo2 hi2 →
    bound_leq lo2 lo1 && bound_leq hi1 hi2 (* [lo1,hi1] \subseteq [lo2,hi2] *)

let interval_join (i1 i2 : interval) : interval =
match i1, i2 with
| IBot, i → i
| i, IBot → i
| IRange lo1 hi1, IRange lo2 hi2 →
    IRange (bound_min lo1 lo2) (bound_max hi1 hi2)

let interval_meet (i1 i2 : interval) : interval =
match i1, i2 with
| IBot, _ → IBot
| _, IBot → IBot
| IRange lo1 hi1, IRange lo2 hi2 →
    let lo = bound_max lo1 lo2 in
    let hi = bound_min hi1 hi2 in
    if bound_leq lo hi then IRange lo hi else IBot

(* Widening: extrapolate to infinity when bounds grow *)
let interval_widen (i1 i2 : interval) : interval =
match i1, i2 with
| IBot, i → i
| i, IBot → i
| IRange lo1 hi1, IRange lo2 hi2 →
    let lo' = if bound_leq lo2 lo1 then lo1 else NegInf in
    let hi' = if bound_leq hi1 hi2 then hi1 else PosInf in
    IRange lo' hi'

(* Narrowing: use condition to refine *)
let interval_narrow (i1 i2 : interval) : interval =
match i1, i2 with
| IBot, _ → IBot
| _, IBot → IBot
| IRange lo1 hi1, IRange lo2 hi2 →
    let lo' = if lo1 = NegInf then lo2 else lo1 in
    let hi' = if hi1 = PosInf then hi2 else hi1 in
    IRange lo' hi'

instance interval_domain : abstract_domain interval = {
  lat = {
    po = { leq = interval_leq; (* proofs omitted for brevity *) };
    bot = IBot;

```

```

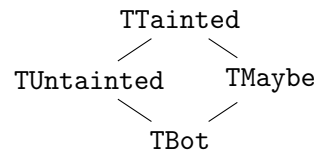
top = IRange NegInf PosInf;
join = interval_join;
meet = interval_meet;
join_set = (* ... *);
meet_set = (* ... *);
(* proofs omitted *)
};
widen = Some interval_widen;
narrow = Some interval_narrow;
(* proofs omitted *)
}

(* Arithmetic transfer functions on intervals *)
let interval_add (i1 i2 : interval) : interval =
  match i1, i2 with
  | IBot, _ | _, IBot → IBot
  | IRange lo1 hi1, IRange lo2 hi2 →
    IRange (bound_add lo1 lo2) (bound_add hi1 hi2)

let interval_mul (i1 i2 : interval) : interval =
  match i1, i2 with
  | IBot, _ | _, IBot → IBot
  | IRange lo1 hi1, IRange lo2 hi2 →
    (* Take min/max of all corner products *)
    let corners = [
      bound_mul lo1 lo2; bound_mul lo1 hi2;
      bound_mul hi1 lo2; bound_mul hi1 hi2
    ] in
    IRange (list_min corners) (list_max corners)

```

Definition 5.27 (Taint Domain). The taint domain tracks whether values originate from untrusted sources (user input, network, etc.). It forms a diamond lattice with four elements:



Critical insight: The TMaybe element breaks the Galois insertion property. This means taint analysis can prove the *presence* of bugs (true positives) but *cannot* prove their absence. See the code comments for detailed explanation.

Taint Domain - For Information Flow Analysis

```

(* Taint Domain
   Source: Denning 1977 (information flow), Liushits 2005 (taint tracking)
   Used for: SQL injection, XSS, command injection, path traversal

   KEY INSIGHT: Taint analysis is a security analysis that tracks
   "trust level" of data through the program. Data from untrusted
   sources (user input) is TTainted; data from trusted sources is TUntainted.
   A bug ∃ when TTainted data reaches a sensitive sink. *)

type taint_level =
  | TBot      (* Unreachable *)
  | TUntainted (* Known safe - from trusted sources *)
  | TMaybe   (* Unknown - could be either *)
  | TTainted  (* Definitely tainted - from untrusted sources *)

(* Taint forms a diamond lattice:
      TTainted
     /      \
    /          \
   /              \
  /                  \
 TBot                TMaybe

```

```
Nullability Domain - For Null Pointer Analysis

(* NULLABILITY DOMAIN
   Used for: Null dereference, optional value tracking, uninitialized vars *)

type nullability =
  | NBot          (* Unreachable *)
```

```

| NNull      (* Definitely null *)
| NNonNull   (* Definitely not null *)
| NMaybe    (* Could be either - TOP *)

let nullability_leq (n1 n2 : nullability) : bool =
  match n1, n2 with
  | NBot, _ → true
  | _, NMaybe → true
  | NNull, NNull → true
  | NNonNull, NNonNull → true
  | _, _ → false

let nullability_join (n1 n2 : nullability) : nullability =
  match n1, n2 with
  | NBot, n | n, NBot → n
  | NNull, NNull → NNull
  | NNonNull, NNonNull → NNonNull
  | _, _ → NMaybe

instance nullability_domain : abstract_domain nullability = {
  lat = {
    bot = NBot;
    top = NMaybe;
    join = nullability_join;
    meet = nullability_meet;
    (* ... *)
  };
  widen = None;
  narrow = None;
}

(* After null check: refine nullability *)
let after_null_check (n : nullability) (is_null_branch : bool) : nullability =
  match n, is_null_branch with
  | NMaybe, true → NNull      (* In "if x == null" branch, x is null *)
  | NMaybe, false → NNonNull  (* In "if x != null" branch, x is non-null *)
  | _, _ → n

```

Resource State Domain - For Lifecycle Analysis

```

(* RESOURCE STATE DOMAIN
   Source: theory.md Chapter 14
   Used for: Use-after-free, double-free, resource leaks, file handles *)

type resource_state =
| RSBot      (* Unreachable *)
| RSUnacquired (* Not yet acquired *)
| RSAcquired  (* Acquired, ready for use *)
| RSInUse     (* Currently being used *)
| RSReleased  (* Released, cannot be used *)
| RSTop      (* Unknown state *)

(* State machine transitions:
   Unacquired --acquire--> Acquired --use--> InUse
           |                   |
           |                   |
           v                   v
           Released ←--release-->

Violations:
- RSReleased → use = UseAfterFree
- RSReleased → release = DoubleFree
- RSAcquired at scope exit = Leak *)

```

```

let resource_leq (r1 r2 : resource_state) : bool =
  match r1, r2 with
  | RSBot, _ → true
  | _, RSTop → true
  | RSUnacquired, RSUnacquired → true
  | RSAcquired, RSAcquired → true
  | RSInUse, RSInUse → true
  | RSReleased, RSReleased → true
  | _, _ → false

let resource_join (r1 r2 : resource_state) : resource_state =
  match r1, r2 with
  | RSBot, r | r, RSBot → r
  | RSTop, _ | _, RSTop → RSTop
  | r1, r2 when r1 = r2 → r1
  | _, _ → RSTop (* Different states merge to unknown *)

(* Transfer functions for resource operations *)
let resource_acquire (r : resource_state) : resource_state * option resource_error =
  match r with
  | RSBot → (RSBot, None)
  | RSUnacquired → (RSAcquired, None)
  | RSAcquired → (RSTop, Some DoubleAcquire)
  | RSInUse → (RSTop, Some DoubleAcquire)
  | RSReleased → (RSAcquired, None) (* Re-acquire after release is OK *)
  | RSTop → (RSTop, None)

let resource_use (r : resource_state) : resource_state * option resource_error =
  match r with
  | RSBot → (RSBot, None)
  | RSUnacquired → (RSTop, Some UseBeforeAcquire)
  | RSAcquired → (RSInUse, None)
  | RSInUse → (RSInUse, None)
  | RSReleased → (RSTop, Some UseAfterRelease)
  | RSTop → (RSTop, None)

let resource_release (r : resource_state) : resource_state * option resource_error =
  match r with
  | RSBot → (RSBot, None)
  | RSUnacquired → (RSTop, Some ReleaseBeforeAcquire)
  | RSAcquired → (RSReleased, None)
  | RSInUse → (RSReleased, None)
  | RSReleased → (RSTop, Some DoubleRelease)
  | RSTop → (RSTop, None)

type resource_error =
  | UseBeforeAcquire
  | UseAfterRelease
  | DoubleAcquire
  | DoubleRelease
  | ReleaseBeforeAcquire
  | LeakOnExit

instance resource_domain : abstract_domain resource_state = {
  lat = { bot = RSBot; top = RSTop; join = resource_join; (* ... *) };
  widen = None;
  narrow = None;
}

```

Ownership Domain - For Memory Safety Analysis

(* OWNERSHIP DOMAIN
 Source: Girard 1987 (substructural logic), Jung 2018 (Iris/RustBelt)
 Used for: Ownership tracking (Rust-style AFFINE), borrow checking, move detection

*NOTE: Rust uses AFFINE typing (use at most once, can drop without use).
Strictly LINEAR requires exactly one use. We model the affine fragment. *)*

```

type ownership =
  | OBot      (* Unreachable *)
  | OUnowned  (* Not owned by anyone *)
  | OOwned    (* Exclusively owned *)
  | OBorrowed (* Temporarily borrowed (immutable) *)
  | OMutBorrowed (* Temporarily borrowed (mutable) *)
  | OShared   (* Reference-counted / shared *)
  | OMoved    (* Ownership transferred away *)
  | OTop      (* Unknown *)

(* Ownership rules (from RustBelt):
  - Owned can be moved or borrowed
  - Borrowed can be cloned (shared borrow)
  - MutBorrowed is exclusive
  - Moved cannot be used *)

let ownership_leq (o1 o2 : ownership) : bool =
  match o1, o2 with
  | OBot, _ → true
  | _, OTop → true
  | o1, o2 when o1 = o2 → true
  | _, _ → false

let ownership_join (o1 o2 : ownership) : ownership =
  match o1, o2 with
  | OBot, o | o, OBot → o
  | OTop, _ | _, OTop → OTop
  | o1, o2 when o1 = o2 → o1
  | OOwned, OBorrowed | OBorrowed, OOwned → OOwned (* Borrow ends *)
  | _, _ → OTop

(* Transfer functions *)
let ownership_move (o : ownership) : ownership * option ownership_error =
  match o with
  | OOwned → (OMoved, None)
  | OMoved → (OTop, Some UseAfterMove)
  | OBorrowed → (OTop, Some MoveWhileBorrowed)
  | OMutBorrowed → (OTop, Some MoveWhileBorrowed)
  | _ → (OTop, None)

let ownership_borrow (o : ownership) : ownership * option ownership_error =
  match o with
  | OOwned → (OBorrowed, None)
  | OBorrowed → (OBorrowed, None) (* Can borrow from borrow *)
  | OMutBorrowed → (OTop, Some BorrowWhileMutBorrowed)
  | OMoved → (OTop, Some UseAfterMove)
  | _ → (OTop, None)

let ownership_mut_borrow (o : ownership) : ownership * option ownership_error =
  match o with
  | OOwned → (OMutBorrowed, None)
  | OBorrowed → (OTop, Some MutBorrowWhileBorrowed)
  | OMutBorrowed → (OTop, Some MutBorrowWhileMutBorrowed)
  | OMoved → (OTop, Some UseAfterMove)
  | _ → (OTop, None)

type ownership_error =
  | UseAfterMove
  | MoveWhileBorrowed

```

```
| BorrowWhileMutBorrowed
| MutBorrowWhileBorrowed
| MutBorrowWhileMutBorrowed
```

5.12 Occurrence Type Domain

Source: Tobin-Hochstadt & Felleisen 2008 (Typed Scheme)

Occurrence Typing: Flow-Sensitive Type Refinement via Predicates

Problem: Union types lose precision after type tests.

```
def process(x: int | str) → int:
  if isinstance(x, str):
    return len(x)  # Type of x should be str here, not int/str!
  return x        # Type of x should be int here
```

Solution: Track TYPE PROPOSITIONS through control flow.

- Type tests generate predicates about variables
- Branch conditions refine types in each branch
- Join points combine refinements conservatively

Complements Gradual Typing (Section 9.1.2):

- Gradual typing: ? type at module BOUNDARIES
- Occurrence typing: Refinement WITHIN module from type tests
- Together: Sound dynamic + static typing in same codebase

Definition 5.28 (Key Concepts from Tobin-Hochstadt 2008). **Visible vs Latent Predicates:**

- **Visible:** Type proposition known to hold at current point.
Example: After `if isinstance(x, str):` we have visible predicate $x : \text{str}$
- **Latent:** Type proposition attached to a function result.
Example: `string?(v)` returns true implies v has type `str`. The function “remembers” what type test it performed.

Propositional Type Environments (Section 3.2):

$$\Gamma = \{x : \tau, \dots\} \quad (\text{Standard type assignment})$$

$$\Gamma | \psi = \text{Environment refined by proposition } \psi$$

$$\psi ::= \tau(x) \mid \neg\tau(x) \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \text{true} \mid \text{false}$$

Environment Refinement Operations:

$$\Gamma + \psi = \text{Environment assuming } \psi \text{ is TRUE}$$

$$\Gamma + \tau(x) = \Gamma[x := \Gamma(x) \cap \tau] \quad (\text{Restrict } x \text{ to } \tau)$$

$$\Gamma + \neg\tau(x) = \Gamma[x := \Gamma(x) - \tau] \quad (\text{Remove } \tau \text{ from } x)$$

$$\Gamma - \psi = \Gamma + \neg\psi \quad (\text{Environment assuming } \psi \text{ is FALSE})$$

Type Operations:

$$\text{restrict}(\sigma, \tau) = \sigma \cap \tau \quad (\text{Narrow } \sigma \text{ to } \tau)$$

$$\text{remove}(\sigma, \tau) = \sigma - \tau \quad (\text{Remove } \tau \text{ from } \sigma)$$

Occurrence Type Domain (Tobin-Hochstadt & Felleisen 2008)

```
module BrrrMachine.Domains.Occurrence

(* TYPE PROPOSITIONS
   Propositions express facts about the types of variables. *)
type type_prop =
```

```

| PropHasType : var:string → ty:ir_type → type_prop
  (* var has type ty *)
| PropNotType : var:string → ty:ir_type → type_prop
  (* var does NOT have type ty *)
| PropAnd : type_prop → type_prop → type_prop
| PropOr : type_prop → type_prop → type_prop
| PropTrue : type_prop
| PropFalse : type_prop (* Contradiction - unreachable *)

(* Negate a proposition *)
let rec negate_prop (p : type_prop) : type_prop =
  match p with
  | PropHasType v t → PropNotType v t
  | PropNotType v t → PropHasType v t
  | PropAnd p1 p2 → PropOr (negate_prop p1) (negate_prop p2)
  | PropOr p1 p2 → PropAnd (negate_prop p1) (negate_prop p2)
  | PropTrue → PropFalse
  | PropFalse → PropTrue

(* VISIBLE AND LATENT PREDICATES *)
type visible_predicate = {
  prop : type_prop;
  source : node_id;          (* Where predicate was established *)
}

type latent_predicate = {
  positive : type_prop;      (* Proposition if function returns true *)
  negative : type_prop;      (* Proposition if function returns false *)
  target_var : option string; (* Variable the predicate applies to *)
}

(* Type predicates built into languages *)
let builtin_predicates : map string latent_predicate = Map.of_list [
  (* Python *)
  ("isinstance", { positive = PropTrue; negative = PropTrue; target_var = None });
  (* TypeScript *)
  ("typeof", { positive = PropTrue; negative = PropTrue; target_var = None });
  (* Go *)
  ("type_assertion", { positive = PropTrue; negative = PropTrue; target_var = None });
]

(* TYPE RESTRICTION AND REMOVAL *)
(* restrict( $\sigma$ ,  $\tau$ ) =  $\sigma$  intersect  $\tau$  *)
let rec restrict ( $\sigma$  : ir_type) ( $\tau$  : ir_type) : ir_type =
  match  $\sigma$  with
  | TUnion types →
    let restricted = List.filter_map (fun t →
      let r = restrict t  $\tau$  in
      if is_bottom r then None else Some r
    ) types in
    (match restricted with
    | [] → TBottom
    | [t] → t
    | ts → TUnion ts)
  | _ →
    if subtype  $\sigma$   $\tau$  then  $\sigma$ 
    else if subtype  $\tau$   $\sigma$  then  $\tau$ 
    else TBottom

(* remove( $\sigma$ ,  $\tau$ ) =  $\sigma$  minus  $\tau$  *)
let rec remove ( $\sigma$  : ir_type) ( $\tau$  : ir_type) : ir_type =
  match  $\sigma$  with
  | TUnion types →

```

```

let remaining = List.filter_map (fun t →
  let r = remove t  $\tau$  in
  if is_bottom r then None else Some r
) types in
(match remaining with
| [] → TBottom
| [t] → t
| ts → TUnion ts)
| _ →
  if subtype  $\sigma$   $\tau$  then TBottom
  else  $\sigma$ 

```

Propositional Type Environment

```

(* Maps variables to types, refined by propositions. *)
type prop_type_env = {
  bindings : map string ir_type;
  active_props : list visible_predicate;
}

(* Refine environment assuming proposition is TRUE: Gamma + psi *)
let rec refine_positive (env : prop_type_env) (prop : type_prop) : prop_type_env =
  match prop with
  | PropHasType var ty →
    (match Map.find var env.bindings with
    | Some current_ty →
      let refined = restrict current_ty ty in
      { env with bindings = Map.add var refined env.bindings }
    | None → env)
  | PropNotType var ty →
    (match Map.find var env.bindings with
    | Some current_ty →
      let refined = remove current_ty ty in
      { env with bindings = Map.add var refined env.bindings }
    | None → env)
  | PropAnd p1 p2 →
    refine_positive (refine_positive env p1) p2
  | PropOr p1 p2 →
    (* Conservative: only refine if BOTH branches agree *)
    let env1 = refine_positive env p1 in
    let env2 = refine_positive env p2 in
    join_environments env1 env2
  | PropTrue → env
  | PropFalse →
    (* Unreachable - mark all types as bottom *)
    { env with bindings = Map.map (fun _ → TBottom) env.bindings }

(* Refine environment assuming proposition is FALSE: Gamma - psi *)
let refine_negative (env : prop_type_env) (prop : type_prop) : prop_type_env =
  refine_positive env (negate_prop prop)

(* Join two environments at control flow merge *)
let join_environments (env1 env2 : prop_type_env) : prop_type_env =
  let bindings = Map.merge (fun _ t1 t2 →
    match t1, t2 with
    | Some t1, Some t2 → Some (type_join t1 t2)
    | Some t, None | None, Some t → Some t
    | None, None → None
  ) env1.bindings env2.bindings in
  { bindings; active_props = [] } (* Props lost at join *)

```

Type Predicate Detection

```

(* Detect type-testing patterns in code. *)
type type_test_result = {

```

```

    tested_var : string;
    test_type : ir_type;
    is_positive : bool;      (* true = "is type", false = "is not type" *)
  }

(* Detect isinstance(x, T) pattern in Python *)
let detect_isinstance (call : ir_expr) : option type_test_result =
  match call with
  | ECall (EVar "isinstance") [EVar var; EType ty] →
    Some { tested_var = var; test_type = ty; is_positive = true }
  | _ → None

(* Detect typeof x === "T" pattern in TypeScript/JavaScript *)
let detect_typeof (expr : ir_expr) : option type_test_result =
  match expr with
  | EBinOp OpEq (ECall (EVar "typeof") [EVar var]) (EString ty_name) →
    let ty = string_to_type ty_name in
    Some { tested_var = var; test_type = ty; is_positive = true }
  | EBinOp OpNeq (ECall (EVar "typeof") [EVar var]) (EString ty_name) →
    let ty = string_to_type ty_name in
    Some { tested_var = var; test_type = ty; is_positive = false }
  | _ → None

(* Detect x !== null / x !== undefined pattern *)
let detect_null_check (expr : ir_expr) : option type_test_result =
  match expr with
  | EBinOp OpNeq (EVar var) ENull →
    Some { tested_var = var; test_type = TNull; is_positive = false }
  | EBinOp OpNeq (EVar var) EUndefined →
    Some { tested_var = var; test_type = TUndefined; is_positive = false }
  | EBinOp OpEq (EVar var) ENull →
    Some { tested_var = var; test_type = TNull; is_positive = true }
  | _ → None

(* Detect "key" in obj pattern for TypeScript discriminated unions *)
let detect_in_check (expr : ir_expr) : option type_test_result =
  match expr with
  | EBinOp OpIn (EString key) (EVar var) →
    (* Having key means var has type with that property *)
    Some { tested_var = var;
           test_type = TWithProperty key;
           is_positive = true }
  | _ → None

```

Occurrence Typing Transfer Function and Lattice

```

(* OCCURRENCE TYPING TRANSFER FUNCTION *)
val occurrence_transfer : prop_type_env → ir_stmt → prop_type_env

let occurrence_transfer env stmt =
  match stmt with
  | SIf cond then_branch else_branch →
    (* Detect type test in condition *)
    let test = detect_type_test cond in
    (match test with
     | Some { tested_var; test_type; is_positive } →
      let true_prop =
        if is_positive then PropHasType tested_var test_type
        else PropNotType tested_var test_type in
      (* Refine environments for each branch *)
      let then_env = refine_positive env true_prop in
      let else_env = refine_negative env true_prop in
      (* Join at merge point *)
      join_environments then_env else_env

```

```

    | None → env)
  | SAssign var expr →
    (* Assignment overwrites type - no longer refined *)
    { env with
      bindings = Map.add var (infer_type expr) env.bindings;
      active_props = List.filter (fun p →
        not (prop_mentions_var p.prop var)
      ) env.active_props }
  | _ → env

(* INTEGRATION WITH ABSTRACT INTERPRETATION *)
type occurrence_domain = {
  env : prop_type_env;
}

let occurrence_lattice : lattice occurrence_domain = {
  bot = { env = { bindings = Map.empty; active_props = [] } };
  top = { env = { bindings = Map.singleton "_" TTop; active_props = [] } };
  leq = fun d1 d2 →
    Map.for_all (fun v t1 →
      match Map.find v d2.env.bindings with
      | Some t2 → subtype t1 t2
      | None → true
    ) d1.env.bindings;
  join = fun d1 d2 →
    { env = join_environments d1.env d2.env };
  meet = fun d1 d2 →
    { env = {
      bindings = Map.merge (fun _ t1 t2 →
        match t1, t2 with
        | Some t1, Some t2 → Some (type_meet t1 t2)
        | _ → None
      ) d1.env.bindings d2.env.bindings;
      active_props = d1.env.active_props @ d2.env.active_props
    } };
}

```

Cross-References:

- Section 9.1.2: Occurrence typing complements gradual typing at boundaries
- Section 9.3: Guards can use occurrence typing propositions
- Section 9.5: Full occurrence typing analysis algorithm
- Section 12.3.5: F* soundness theorem for occurrence typing

5.13 DeepPoly: Abstract Domain for Neural Networks

Paper: Singh, Gehr, Puschel, Vechev 2019 (POPL)

Neural Network Verification via Abstract Interpretation

Problem: Neural networks in safety-critical systems (autonomous driving, medical diagnosis) require robustness certification. Prior methods (SMT, MILP) cannot scale beyond ~2K neurons.

DeepPoly Insight: Restricted polyhedral domain with backsubstitution.

- One lower bound + one upper bound per neuron (prevents blowup)
- Affine expressions relate neurons to predecessors
- Backsubstitution computes tight bounds through layers

Abstract Element for neuron x_i :

$$a_i = (a_i^{\leq}, a_i^{\geq}, l_i, u_i)$$

where:

$$\begin{aligned} a_i^{\leq}(x) \leq x_i \leq a_i^{\geq}(x) & \quad (\text{symbolic polyhedral bounds}) \\ l_i \leq x_i \leq u_i & \quad (\text{concrete interval bounds}) \end{aligned}$$

Key Transformers:

- Affine layer: EXACT (no precision loss)
- ReLU layer: Area-minimizing linear approximation (sound)
- Sigmoid/Tanh: Tangent-based linear bounds (sound)

Performance: 100–1000x faster than MILP, scales to 62K neurons.

Remark 5.29 (ReLU Approximation Strategy). For $\text{ReLU}(x) = \max(0, x)$ when interval crosses zero ($l < 0 < u$):

Two candidate approximations based on AREA MINIMIZATION:

- Option (b): Lower = 0, Upper = $\lambda x + \mu$ (area = $0.5 \cdot u \cdot (u - l)$)
- Option (c): Lower = x , Upper = $\lambda x + \mu$ (area = $0.5 \cdot (-l) \cdot (u - l)$)

Choose the option with SMALLER area in (x, output) plane. This geometric insight is KEY to DeepPoly's precision.

Remark 5.30 (Backsubstitution for Precision). To compute tight bounds for neuron x_i :

1. Start with symbolic constraint $a_i(x_0, \dots, x_{i-1})$
2. Recursively substitute constraints for intermediate neurons
3. Continue until only INPUT variables remain
4. Apply concrete input bounds to get tight interval

This enables RELATIONAL reasoning across many layers without storing exponential constraints.

The following F* code implements the DeepPoly abstract domain. The key type is `neuron_constraint`, which stores both symbolic polyhedral bounds (affine expressions relating to predecessors) and concrete interval bounds. The `relu_transform` function implements the area-minimizing approximation for ReLU neurons.

DeepPoly Abstract Domain for Neural Networks (Singh et al. POPL 2019)

```
(* This module implements the DeepPoly abstract domain for neural network
   verification. The key insight is to maintain RELATIONAL constraints between
   neurons (via affine expressions) while using backsubstitution to compute
   tight bounds. This achieves much better precision than interval analysis
   while remaining tractable for large networks. *)

module BrrrMachine.Domains.DeepPoly

(* Affine expression: c_0 + sum(c_i * x_i) *)
type affine_expr = {
  const : real;
  coeffs : list (nat * real);  (* (neuron_index, coefficient) *)
}

(* DeepPoly constraint for single neuron *)
type neuron_constraint = {
  nc_lower : affine_expr;      (* Lower polyhedral bound *)
  nc_upper : affine_expr;      (* Upper polyhedral bound *)
  nc_lo : real;                (* Interval lower bound *)
  nc_hi : real;                (* Interval upper bound *)
}

(* Network abstract state *)
```

```

type deeppoly_state = {
  constraints : list neuron_constraint;
  input_bounds : list (real * real);
  num_inputs : nat;
}

(* Concretization: set of all concrete values satisfying constraints *)
val  $\gamma$  : deeppoly_state  $\rightarrow$  set (list real)

(* RELU TRANSFORMER - Area-minimizing approximation *)
val relu_transform : neuron_constraint  $\rightarrow$  neuron_constraint
let relu_transform nc =
  if nc.nc_hi  $\leq$  0.0 then
    (* Always inactive: output = 0 *)
    { nc_lower = { const = 0.0; coeffs = [] };
      nc_upper = { const = 0.0; coeffs = [] };
      nc_lo = 0.0; nc_hi = 0.0 }
  else if nc.nc_lo  $\geq$  0.0 then
    (* Always active: identity *)
    nc
  else
    (* Crossing zero: area-minimizing approximation *)
    let lambda = nc.nc_hi /. (nc.nc_hi -. nc.nc_lo) in
    let mu = -. nc.nc_lo *. nc.nc_hi /. (nc.nc_hi -. nc.nc_lo) in
    (* Choose smaller area: compare |u| vs |l| *)
    if nc.nc_hi  $\leq$  -. nc.nc_lo then
      (* Option (b): lower = 0 has smaller area *)
      { nc_lower = { const = 0.0; coeffs = [] };
        nc_upper = scale_add_affine lambda nc.nc_upper mu;
        nc_lo = 0.0; nc_hi = nc.nc_hi }
    else
      (* Option (c): lower = x has smaller area *)
      { nc_lower = nc.nc_lower;
        nc_upper = scale_add_affine lambda nc.nc_upper mu;
        nc_lo = nc.nc_lo; nc_hi = nc.nc_hi }

(* BACKSUBSTITUTION - Compute tight bounds via recursive substitution *)
val backsubstitute :
  constraints:list neuron_constraint  $\rightarrow$ 
  expr:affine_expr  $\rightarrow$ 
  input_bounds:list (real * real)  $\rightarrow$ 
  (real * real) (* Tight lower and upper bounds *)

(* SOUNDNESS THEOREMS *)
val affine_layer_exact :
  state:deeppoly_state  $\rightarrow$  weights:matrix  $\rightarrow$  bias:list real  $\rightarrow$ 
  Lemma ( $\gamma$  (transform_affine state weights bias) =
    image (apply_affine weights bias) ( $\gamma$  state))

val relu_layer_sound :
  state:deeppoly_state  $\rightarrow$ 
  Lemma (image apply_relu ( $\gamma$  state) 'subset'
     $\gamma$  (transform_relu state))

val network_certification_sound :
  layers:list layer  $\rightarrow$  input_region:deeppoly_state  $\rightarrow$ 
  Lemma ( $\forall$  x. x 'in'  $\gamma$  input_region  $\Rightarrow$ 
    network_output layers x 'in'  $\gamma$  (analyze_layers input_region))

```

Remark 5.31 (Domain Taxonomy Extension). DeepPoly introduces a new category of abstract domains:

Domain Precision Hierarchy (extended):

Intervals < Zonotopes < DeepPoly < Full Polyhedra < MILP

(fast)

(slow)

DeepPoly achieves a sweet spot:

- More precise than intervals/zonotopes (relational constraints)
- Much faster than polyhedra/MILP (restricted form)
- Scales to production networks (62K+ neurons)

Cross-References:

- Section 2.1.6: Abstract domain formalization (DeepPoly is an instance)
- Section 2.1.7: Concrete domains (DeepPoly extends this taxonomy)
- Section 2.1.5c: Neural invariant synthesis (complementary technique)

5.14 Dual Soundness: Over and Under Approximation

Papers: O’Hearn 2020 (Incorrectness Logic - historical, superseded by OL), Zilberstein 2023 (Outcome Logic)

Critical Insight: Analysis Soundness Has Two Dual Forms

The synthesis previously conflated these — they serve DIFFERENT purposes.

Definition 5.32 (Verification Soundness (Over-Approximation)).

$$\alpha(\text{concrete}) \subseteq \text{abstract}$$

“If analysis says SAFE, it truly is safe.”

- May have FALSE POSITIVES (false alarms)
- Cannot have false negatives (missed bugs)
- Used for: proving absence of bugs, verification

This is what Cousot abstract interpretation provides.

Definition 5.33 (Detection Soundness (Under-Approximation)).

$$\text{abstract} \subseteq \text{concrete}$$

“If analysis says BUG, it truly is a bug.”

- May have FALSE NEGATIVES (missed bugs)
- Cannot have false positives (false alarms)
- Used for: proving presence of bugs, bug finding

This is what O’Hearn’s Incorrectness Logic provided historically, now superseded by Outcome Logic (OL).

The Brrr-Machine Needs Both

| Proven Safe (over-approx) | Uncertain | Proven Buggy (under-approx) |
|---------------------------------|-------------------------------|---|
| No bugs exist 100% confident | May have bugs Needs review | Bug definitely exists 100% confident |

Remark 5.34 (Why This Matters). 1. **Verification mode:** We want to PROVE code is safe. Use over-approximation. False positives acceptable; false negatives catastrophic.

2. **Bug-finding mode:** We want to FIND real bugs. Use under-approximation. False negatives acceptable; false positives waste developer time.
3. **Combined mode:** Use BOTH to classify findings into proven-safe, proven-buggy, and uncertain categories.

Definition 5.35 (Consequence Rule Reversal). **Hoare (over-approx):**

$$\frac{p' \Rightarrow p \quad \{p\}C\{q\} \quad q \Rightarrow q'}{\{p'\}C\{q'\}}$$

STRONGER pre, WEAKER post

Incorrectness (under-approx):

$$\frac{p' \Leftarrow p \quad [p]C[q] \quad q' \Leftarrow q}{[p']C[q']}$$

WEAKER pre, STRONGER post

Definition 5.36 (Disjunction Behavior). **Over-approx:** Must track ALL paths

$$\frac{\{p_1\}C\{q_1\} \quad \{p_2\}C\{q_2\}}{\{p_1 \wedge p_2\}C\{q_1 \wedge q_2\}}$$

Intersection of posts (must hold in both)

Under-approx: Can DROP paths

$$\frac{[p_1]C[q_1] \quad [p_2]C[q_2]}{[p_1 \vee p_2]C[q_1 \vee q_2]}$$

Union of posts (may drop one path!)

This justifies PARTIAL COVERAGE in bug-finding — sound to forget paths.

Remark 5.37 (Integration with Outcome Logic (OL)). **Critical:** Incorrectness Logic (IL) is incompatible with abstract interpretation (Ascari et al. 2022). OL fixes this while preserving IL’s true-positive property.

Why OL Over IL:

1. IL breaks abstraction — reverse consequence rule + abstract AI don’t compose
2. IL’s “One-Sided If” generates imprecise preconditions
3. IL can’t handle probabilistic programs

OL provides:

- Unified framework for correctness AND incorrectness
- Compatible with abstract interpretation
- Manifest/latent bug classification (see Section 12.5)
- Falsification completeness (Theorem 5.1)

Use OL as the foundation for dual-mode analysis.

See Section 12.4 for formal theorems and Section 12.5 for manifest error detection.

5.15 Local Completeness Logic (Bruni et al. 2023)

Paper: Bruni, Giacobazzi, Gori, Ranzato 2023 (A Correctness and Incorrectness Program Logic)

Key Insight: Local Completeness Bridges Over/Under Approximation

Global completeness ($\alpha \circ f = f^\# \circ \alpha$) is RARE — only trivial domains achieve it.
 LOCAL completeness (at specific inputs) is ACHIEVABLE and USEFUL.
 LCL_A combines BOTH correctness AND incorrectness in ONE framework, parameterized by abstract domain A .

Theorem 5.38 (Global Completeness Is Rare (Giacobazzi et al. 2015, Theorem 4.5)). *For Turing-complete languages, only TRIVIAL domains are globally complete:*

1. Identity abstraction (concrete = abstract) — useless
2. Top abstraction (all programs equivalent) — useless

Consequence: For any useful abstract domain, there exist inputs where the analysis produces false alarms (over-approximation is imprecise).

The Solution: LOCAL COMPLETENESS — Instead of requiring completeness for ALL inputs, require it only for SPECIFIC inputs of interest.

Definition 5.39 (Local Completeness (Definition 4.1)). Abstract domain A is *locally complete* for $f : C \rightarrow C$ at concrete value $c \in C$, written $\mathcal{C}_c^A(f)$, if:

$$\mathcal{C}_c^A(f) \iff \alpha \circ f(c) = \alpha \circ f \circ \gamma \circ \alpha(c)$$

Meaning: For input c , the abstraction of $f(c)$ equals the abstraction of applying f to the “best” approximation of c .

Global vs Local:

Global: $\forall c \in C. \mathcal{C}_c^A(f)$ (for all inputs — rare)

Local: $\mathcal{C}_c^A(f)$ (for specific input c — achievable)

Definition 5.40 (The LCL_A Proof System). **LCL_A Triple:** $\vdash_A [p] r [q]$

Meaning:

1. $q \subseteq \llbracket r \rrbracket p$ (q under-approximates strongest postcondition)
2. $\mathcal{C}_p^A(r)$ holds (r is locally complete for input p in A)
3. $\alpha(q) = \alpha(\llbracket r \rrbracket p)$ (abstractions coincide)

Key Insight: If $\vdash_A [p] r [q]$ is provable, then:

- q contains only TRUE alarms (under-approximation property)
- If q has no alarms, then $\llbracket r \rrbracket p$ has no alarms (completeness at p)
- The analysis is BOTH sound for bugs AND complete for safety at p

Definition 5.41 (Key Rules of LCL_A). **Transfer Rule:** Base case for primitive commands

$$\frac{\mathcal{C}_p^A(e)}{\vdash_A [p] e [\llbracket e \rrbracket p]} \quad (\text{transfer})$$

Relax Rule: Weaken precondition, strengthen postcondition (within abstraction)

$$\frac{p' \subseteq p \subseteq \gamma(\alpha(p')) \quad \vdash_A [p'] r [q'] \quad q \subseteq q' \subseteq \gamma(\alpha(q))}{\vdash_A [p] r [q]} \quad (\text{relax})$$

Sequence Rule: Composition

$$\frac{\vdash_A [p] r_1 [w] \quad \vdash_A [w] r_2 [q]}{\vdash_A [p] r_1; r_2 [q]} \quad (\text{seq})$$

Join Rule: Non-deterministic choice

$$\frac{\vdash_A [p] r_1 [q_1] \quad \vdash_A [p] r_2 [q_2]}{\vdash_A [p] r_1 + r_2 [q_1 \cup q_2]} \quad (\text{join})$$

Iterate Rule: Loop with local completeness check

$$\frac{\vdash_A [p] r [q] \quad q \subseteq \gamma(\alpha(p))}{\vdash_A [p] r^* [p \cup q]} \quad (\text{iterate})$$

Theorem 5.42 (Connection to Incorrectness Logic (Section 6)). $IL = LCL_{A_{tr}}$

When $A = A_{tr}$ (the trivial “top” abstraction that makes all properties equivalent):

- Every transfer function is globally complete in A_{tr}
- $LCL_{A_{tr}}$ coincides EXACTLY with O’Hearn’s Incorrectness Logic
- This is because A_{tr} provides NO precision — can’t distinguish any states

Implication:

- IL is a SPECIAL CASE of LCL_A
- For any non-trivial A , LCL_A is MORE POWERFUL than IL
- LCL_A can prove CORRECTNESS (when q has no alarms)
- IL can only prove INCORRECTNESS (existence of bugs)

Theorem 5.43 (Soundness of LCL_A (Theorem 5.5)). If $\vdash_A [p] r [q]$ then:

1. $q \subseteq \llbracket r \rrbracket p$ (under-approximation)
2. $\llbracket r \rrbracket_A^\# \alpha(p) = \alpha(q) = \alpha(\llbracket r \rrbracket p)$ (local completeness)

Corollary 5.44 (Precision (Corollary 5.6)). If $\vdash_A [p] r [q]$ then for all $a \in A$:

$$\llbracket r \rrbracket p \subseteq \gamma(a) \iff q \subseteq \gamma(a)$$

Meaning: q and $\llbracket r \rrbracket p$ have the SAME abstract over-approximations.

Unified Tri-State Analysis Using LCL_A

Given abstract domain A and input p , analyze program C :

Attempt to prove $\vdash_A [p] C [q]$

Case 1: q contains alarm states

⇒ **Proven Buggy** (under-approximation guarantees true bugs)

Case 2: q contains no alarm states AND proof succeeds

⇒ **Proven Safe** (local completeness at p means no false negatives)

Case 3: Proof fails (local completeness obligation not met)

⇒ **Uncertain** (need more precise domain or different input abstraction)

Remark 5.45 (Domain Refinement for Local Completeness (Section 8)). When local completeness fails, REFINES the domain.

If $\mathcal{C}_p^A(f)$ fails, we can:

1. Compute the BEST CORRECT APPROXIMATION (bca) for f at p
2. Extend A to include this value
3. Re-attempt the proof with refined domain

This gives a DIRECTED refinement strategy — refine only where needed, not globally. Much more practical than complete abstract refinement.

Remark 5.46 (Integration with Outcome Logic). $LCL_A + OL$ Integration:

Both frameworks address over/under approximation unification:

- OL (Zilberstein 2023): Unified semantics for correctness and incorrectness
- LCL_A (Bruni 2023): Unified proof system parameterized by abstract domain

Recommended Approach:

1. Use OL for SEMANTICS (what correctness/incorrectness mean)

2. Use LCL_A for ANALYSIS (how to compute verdicts)
3. Abstract domain A determines precision/cost tradeoff
4. Local completeness obligations guide where to refine A

See Section 12.4 for formal theorems and Section 12.5 for manifest error detection.

Cross-References:

- For probabilistic programs, LCL_A can be instantiated with probabilistic abstract domains (Section 2.1.6c).
- The three abstraction axes (semantics, scenarios, distributions) provide orthogonal precision control.
- Local completeness extends to probabilistic settings: completeness at input distribution rather than input set.

5.16 Unified Constraint Domain Framework

Paper: Xi & Pfenning 1999 (Dependent Types in Practical Programming)

Key Insight: All Abstract Domains Are DML(C) Instantiations

The domains in 2.1.7 (interval, taint, nullability, resource, ownership) appear ad-hoc but share a COMMON STRUCTURE.

Each is an instantiation of Dependent ML (Xi 1999) with a different constraint system C . This unifies our implementation and theory.

Definition 5.47 (The DML(C) Pattern). Types indexed by CONSTRAINT DOMAIN values:

$$\text{DML}(C) = \{$$

$$\begin{array}{ll} \text{index_sort} : \text{Type}; & \text{(What indices look like)} \\ \text{index_term} : \text{Type}; & \text{(Terms over indices)} \\ \text{constraint} : \text{Type}; & \text{(Constraints over indices)} \\ \text{satisfiable} : \text{constraint} \rightarrow \text{bool}; \\ \text{implies} : \text{constraint} \rightarrow \text{constraint} \rightarrow \text{bool}; \\ \text{widen} : \text{constraint} \rightarrow \text{constraint} \rightarrow \text{constraint}; \end{array}$$

$$\}$$

Definition 5.48 (Our Domains as DML(C)).

| Domain | Constraint System C |
|-------------|---|
| Interval | Linear integer arithmetic ($a \cdot x + b \cdot y \leq c$) |
| Nullability | Boolean constraints ($\text{isNull} \vee \neg \text{isNull}$) |
| Taint | Security lattice ($\text{level}_1 \sqsubseteq \text{level}_2$) |
| Resource | Finite state constraints ($\text{state} \in \{S_1, S_2, \dots\}$) |
| Ownership | Capability constraints ($\text{unique} \oplus \text{dup}$) |

Remark 5.49 (Benefits of Unified View). 1. **Implementation:** One parametric domain module, many instantiations

2. **Composition:** Cross-domain reasoning via constraint conjunction
3. **Decidability:** Constraint satisfiability gives analysis termination
4. **Theory:** Single soundness proof covers all domains

Unified Constraint Domain Signature

```

(* UNIFIED CONSTRAINT DOMAIN SIGNATURE *)
class constraint_domain (c : Type) = {
  index_sort : Type;
  index_term : Type;
  constraint_ : Type;
  (* Core decision procedures *)
  satisfiable : constraint_ → bool;
  implies : constraint_ → constraint_ → bool;
  (* Abstract interpretation operations *)
  meet : constraint_ → constraint_ → constraint_;
  join : constraint_ → constraint_ → constraint_;
  widen : constraint_ → constraint_ → constraint_;
}

(* ALL OUR DOMAINS AS INSTANCES *)

(* Interval: C = linear integer constraints *)
instance interval_cd : constraint_domain interval = {
  index_sort = Int;
  index_term = LinExpr;  (* a_1 x_1 + a_2 x_2 + ... + c *)
  constraint_ = LinConstraint;
  satisfiable = fourier_motzkin_sat;
  implies = fun c1 c2 → not (satisfiable (c1 ∧ not c2));
  meet = fun c1 c2 → c1 ∧ c2;
  join = convex_hull;
  widen = interval_widen;
}

(* Taint: C = security lattice constraints *)
instance taint_cd : constraint_domain taint = {
  index_sort = SecurityLevel;
  index_term = LevelExpr;
  constraint_ = FlowConstraint;  (* l_1 \sqsubset l_2 *)
  satisfiable = lattice_sat;
  implies = lattice_implies;
  meet = fun c1 c2 → c1 ∧ c2;
  join = lub_constraint;
  widen = id;  (* Finite *)
}

```

Example 5.50 (Application: Array Bounds Verification). With this framework, we can statically eliminate bounds checks:

```

(* Refined array type with length index *)
type array_refined (a : Type) (n : index_term) = {
  data : array a;
  length_proof : squash (data.length == eval n);
}

(* Bounds check elimination *)
val access_safe :
  ctx : constraint_ →          (* What we know at this point *)
  arr : array_refined a n →    (* Array with length n *)
  idx : index_term →          (* Index expression *)
  implies ctx (0 ≤ idx ∧ idx < n) == true → (* Provable in-bounds *)
  a  (* No runtime check needed! *)

```

This is how Liquid Haskell (Vazou 2014) and Dependent ML work.

See Section 12.11 for complete F* formalization.

Part III

Program Representation

6. The Code Property Graph

Foundational Papers: Yamaguchi et al. 2014, Ferrante et al. 1987, Horwitz et al. 1990, Weiser 1984

The Code Property Graph (CPG) is the central data structure of the brrr-machine. It unifies multiple program representations into a single queryable graph, enabling all analyses to be expressed as graph traversals.

6.1 Why CPG?

The Problem with Separate Representations

Traditional analyzers maintain separate data structures:

- AST for syntactic queries
- CFG for control flow
- Call graph for interprocedural analysis
- Def-use chains for data flow

This leads to:

- Redundant storage
- Complex synchronization
- Difficult cross-cutting queries

The CPG Solution (Yamaguchi 2014)

Merge all representations into **one** graph:

- Nodes are shared (a statement is both AST node and CFG node)
- Edges are labeled by type (AST_CHILD, CFG_NEXT, DATA_DEP, etc.)
- Queries mix edge types freely

Benefits:

- Single data structure to maintain
- Natural expression of complex queries
- Efficient traversals via graph algorithms

6.2 CPG Components

Definition 6.1 (Code Property Graph). The Code Property Graph is the union of five component graphs:

$$\text{CPG} = \text{AST} \cup \text{CFG} \cup \text{PDG} \cup \text{CallGraph} \cup \text{EffectGraph}$$

Each component contributes specific node interpretations and edge types.

6.2.1 AST (Abstract Syntax Tree)

- **Nodes:** Every syntactic element (functions, statements, expressions, etc.)
- **Edges:** Child(*i*) from parent to *i*-th child
- **Purpose:** Syntactic structure, source location mapping

6.2.2 CFG (Control Flow Graph)

- **Nodes:** Shared with AST statement/expression nodes
- **Edges:** NEXT, TRUE_BRANCH, FALSE_BRANCH, EXCEPTION
- **Purpose:** Possible execution orders

6.2.3 PDG (Program Dependence Graph)

Source: Ferrante et al. 1987

- **Nodes:** Shared with AST
- **Edges:** DataDep(*var*), CtrlDep(*branch*)
- **Purpose:** Semantic dependencies for slicing

6.2.4 Call Graph

- **Nodes:** Shared with AST function/call nodes
- **Edges:** Calls(*site*), ParamIn(*i*), ParamOut
- **Purpose:** Interprocedural analysis

6.2.5 Effect Graph (brrr-machine Extension)

- **Nodes:** Effect nodes (read, write, alloc, free, io, etc.)
- **Edges:** Effect(*kind*), EFFECT_ORDER, EFFECT_CONFLICT
- **Purpose:** Effect tracking for bug detection

6.3 Edge Types in Detail

The following F* formalization defines the complete edge type vocabulary for the Code Property Graph. Each edge type is modeled as a constructor of an algebraic data type, with some edges carrying payload data (e.g., variable names for data dependencies, branch directions for control dependencies).

Key design principles:

- **AST edges** preserve syntactic parent-child relationships with explicit position indices
- **CFG edges** distinguish conditional branches (true/false) from unconditional flow
- **PDG edges** carry the variable name involved in the dependency, enabling precise slicing
- **Call graph edges** follow Horwitz 1990's System Dependence Graph structure with summary edges
- **Effect edges** are a brrr-machine extension for tracking side effects and their ordering
- **Channel edges** integrate Honda 1998/2008 session types for concurrent program analysis

CPG Edge Types

```
(* =====  
   CPG EDGE TYPES  
   ===== *)  
  
type cpg_edge_label =  
  (* -----  
     AST EDGES --- Syntactic structure  
     ----- *)  
  | AstChild : index:nat → cpg_edge_label  
    (* Parent contains child at position index.  
       Enables: tree traversals, subtree extraction, pattern matching *)  
  
  | AstNextSibling : cpg_edge_label  
    (* Sibling relationship for sequential AST nodes.  
       Enables: statement sequence traversal *)  
  
  (* -----  
     CFG EDGES --- Control flow  
     Source: standard compiler construction  
     ----- *)  
  | CfgNext : cpg_edge_label  
    (* Unconditional successor.  
       From: any statement  
       To: next statement in sequence *)
```

```

| CfgTrue : cpg_edge_label
  (* Successor when condition is true.
     From: if/while/for condition
     To: then-branch / loop body *)

| CfgFalse : cpg_edge_label
  (* Successor when condition is false.
     From: if/while/for condition
     To: else-branch / after loop *)

| CfgException : exn_type:string → cpg_edge_label
  (* Successor on exception of given type.
     From: potentially-throwing statement
     To: matching catch block or function exit *)

| CfgEntry : cpg_edge_label
  (* Function entry edge.
     From: function node
     To: first statement *)

| CfgExit : cpg_edge_label
  (* Function exit edge.
     From: return statement or last statement
     To: function exit node *)

(* -----
   PDG EDGES --- Data and control dependencies
   Source: Ferrante 1987
   ----- *)

| DataDep : var:string → cpg_edge_label
  (* Data dependence: target uses value defined at source.
     From: definition of var
     To: use of var
     Enables: reaching definitions, taint propagation *)

| CtrlDep : branch:bool → cpg_edge_label
  (* Control dependence: target's execution depends on source's branch.
     From: branch condition
     To: statement that executes only if branch goes this way
     The bool indicates which branch (true/false)
     Enables: slicing, dead code detection *)

| OutputDep : var:string → cpg_edge_label
  (* Output dependence: both source and target write to var.
     From: earlier write
     To: later write
     Enables: race detection, write-write conflict *)

| AntiDep : var:string → cpg_edge_label
  (* Anti-dependence: source reads var, target writes var.
     From: read
     To: write
     Enables: race detection, read-write conflict *)

(* -----
   CALL GRAPH EDGES --- Interprocedural structure
   Source: Horwitz 1990 (SDG)
   ----- *)

| Calls : site_id:nat → cpg_edge_label
  (* Call edge from call site to callee.
     From: call expression
     To: callee function entry *)

```

```

| ParamIn : index:nat → cpg_edge_label
  (* Actual-to-formal parameter binding.
     From: actual argument expression
     To: formal parameter *)

| ParamOut : cpg_edge_label
  (* Return value flow.
     From: return statement value
     To: call site result *)

| Summary : cpg_edge_label
  (* Summary edge: transitive dependence through procedure.
     From: input parameter
     To: output (return or modified parameter)
     Enables: efficient interprocedural analysis without re-analyzing callee *)

(* -----
   EFFECT EDGES --- Side effect tracking
   Source: brrr-machine theory (our contribution)
   ----- *)

| Effect : kind:effect_kind → cpg_edge_label
  (* Statement has this effect.
     From: statement
     To: effect node *)

| EffectOrder : cpg_edge_label
  (* Effect ordering: source must happen before target.
     Derived from CFG + data dependencies *)

| EffectConflict : cpg_edge_label
  (* Effects may conflict (race condition candidate).
     Between: effects on same location from different threads *)

| EffectCause : cpg_edge_label
  (* Causal chain: source effect enables/causes target effect.
     Enables: root cause analysis *)

(* -----
   CHANNEL EDGES --- Honda 1998/2008 session type structure
   These edges integrate channel operations into the CPG for analysis of
   communication patterns, protocol conformance, and deadlock detection.
   ----- *)

| ChannelFlow : chan_id:nat → cpg_edge_label
  (* Data flow through channel: send → recv.
     From: NChannelSend node
     To: NChannelRecv node on same channel
     Enables: taint propagation through channels, data flow analysis *)

| SessionOf : cpg_edge_label
  (* Associates prefix node with its session.
     From: NPrefixNode
     To: NSessionNode *)

| ParticipantOf : cpg_edge_label
  (* Associates participant node with its session.
     From: NParticipantNode
     To: NSessionNode *)

| LocalTypeOf : cpg_edge_label
  (* Associates process with its local session type.
     From: NFunction or NBlock (implementing participant)
     To: NParticipantNode (with local type) *)

```

```

| SessionCausality : kind:causality_kind → cpg_edge_label
  (* Causality dependency between prefixes (Honda 2008 Section 3.2).
     From: earlier prefix node
     To: later prefix node
     kind: II (input-input), IO (input-output), OO (output-output)
     Enables: deadlock detection, protocol conformance, linearity checking *)

| SessionNext : cpg_edge_label
  (* Sequential composition within session type.
     From: prefix node
     To: next prefix node in sequence *)

| ChannelCreate : cpg_edge_label
  (* Channel creation to channel use.
     From: NChannelCreate node
     To: NChannelSend, NChannelRecv, or NChannelClose node *)

| ChannelAlias : cpg_edge_label
  (* Channel aliasing: source and target refer to same channel.
     From: channel variable assignment
     To: channel variable use
     Enables: alias analysis for channels *)

| DelegationTransfer : cpg_edge_label
  (* Session delegation: capability transfer.
     From: NChannelDelegate (sender)
     To: NChannelRecv (receiver of capability)
     Enables: tracking session ownership transfer *)

and causality_kind =
| CausalityII (* Input-Input: two inputs at same participant must be ordered *)
| CausalityIO (* Input-Output: output depends on data from input *)
| CausalityOO (* Output-Output: two outputs from same sender must be ordered *)

```

6.4 Node Types

The CPG node vocabulary captures all syntactic and semantic elements of a program. Unlike traditional ASTs that only represent syntax, CPG nodes unify multiple views:

- **AST nodes** represent syntactic program structure (functions, statements, expressions)
- **CFG nodes** mark control flow entry/exit points and join locations
- **Channel nodes** (Honda 1998/2008) represent concurrent communication operations
- **Effect nodes** track side effects for bug detection and resource analysis

Type signature conventions:

- Node kinds with `name:string` carry the identifier name for lookup and display
- Node kinds with `index:nat` carry positional information (parameters, children)
- Channel nodes carry `chan_id:nat` to track channel identity across operations
- Session nodes reference `session_id` for protocol conformance checking

The following F* formalization defines the complete node type vocabulary:

```

CPG Node Types
(* =====
   CPG NODE TYPES
   ===== *)

type cpg_node_kind =
(* -----
   AST NODES --- Program structure
   ----- *)

```

```

| NFile : path:string → cpg_node_kind
| NModule : name:string → cpg_node_kind
| NClass : name:string → cpg_node_kind
| NFunction : name:string → params:list string → cpg_node_kind
| NMethod : name:string → receiver:string → params:list string → cpg_node_kind
| NParameter : name:string → index:nat → cpg_node_kind
| NLocalVar : name:string → cpg_node_kind
| NGlobalVar : name:string → cpg_node_kind

(* Statements *)
| NBlock : cpg_node_kind
| NExprStmt : cpg_node_kind
| NVarDecl : name:string → cpg_node_kind
| NAssign : target:string → cpg_node_kind
| NIf : cpg_node_kind
| NWhile : cpg_node_kind
| NFor : cpg_node_kind
| NForEach : cpg_node_kind
| NReturn : cpg_node_kind
| NThrow : cpg_node_kind
| NTry : cpg_node_kind
| NCatch : exn_type:string → cpg_node_kind
| NBreak : cpg_node_kind
| NContinue : cpg_node_kind

(* Expressions *)
| NLiteral : lit_kind:literal_kind → cpg_node_kind
| NIdentifier : name:string → cpg_node_kind
| NBinaryOp : op:string → cpg_node_kind
| NUnaryOp : op:string → cpg_node_kind
| NCall : cpg_node_kind
| NMethodCall : method_name:string → cpg_node_kind
| NFieldAccess : field:string → cpg_node_kind
| NIndexAccess : cpg_node_kind
| NLambda : cpg_node_kind
| NNew : type_name:string → cpg_node_kind
| NConditional : cpg_node_kind (* ternary *)
| NCast : target_type:string → cpg_node_kind

(* -----
   CFG NODES --- Control flow structure
   ----- *)
| NEntry : func:string → cpg_node_kind
| NExit : func:string → cpg_node_kind
| NJoin : cpg_node_kind (* Merge point for branches *)
| NLoopHead : cpg_node_kind (* Target for widening *)

(* -----
   CHANNEL NODES --- Honda 1998/2008 session type operations
   Integrates with AST (channel operations in syntax), CFG (control flow
   through channel ops), PDG (data dependencies through channels),
   and call graph (channels passed to functions).
   ----- *)
| NChannelCreate : chan_id:nat → elem_type:string → buffer_size:nat → cpg_node_kind
  (* Channel creation: ch := make(chan T) or mpsc::channel() *)
| NChannelSend : chan_id:nat → cpg_node_kind
  (* Send operation: ch ← v or tx.send(v) *)
| NChannelRecv : chan_id:nat → binding:option string → cpg_node_kind
  (* Receive operation: v := ←ch or rx.recv() *)
| NChannelClose : chan_id:nat → cpg_node_kind

```

```

    (* Close operation: close(ch) or drop(tx) *)

| NChannelSelect : cases:nat → has_default:bool → cpg_node_kind
    (* Select statement: select { case ... } or tokio::select! *)

| NChannelSelectCase : chan_id:nat → direction:chan_direction → cpg_node_kind
    (* Individual case in select *)

| NChannelBranch : chan_id:nat → labels:list string → cpg_node_kind
    (* Session type branching: offer branches on channel *)

| NChannelChoice : chan_id:nat → label:string → cpg_node_kind
    (* Session type selection: select branch on channel *)

| NChannelDelegate : chan_id:nat → delegated_chan:nat → cpg_node_kind
    (* Session delegation: transfer capability through channel *)

(* Session type nodes --- for protocol conformance checking *)
| NSessionNode : session_id:nat → global_type:string → cpg_node_kind
    (* Session (conversation) node *)

| NParticipantNode : session_id:nat → participant:nat → local_type:string → cpg_node_kind
    (* Participant in a session with its local type *)

| NPrefixNode : session_id:nat → kind:prefix_kind → channel:nat → payload:string →
    ↪ cpg_node_kind
    (* Communication prefix in session type *)

(* -----
    EFFECT NODES --- Side effects
    ----- *)
| NEffect : kind:effect_kind → cpg_node_kind

and chan_direction = ChanSend | ChanRecv

and prefix_kind = PrefixSend | PrefixRecv | PrefixSelect | PrefixBranch

and literal_kind =
| LitInt : value:int → literal_kind
| LitFloat : value:float → literal_kind
| LitString : value:string → literal_kind
| LitBool : value:bool → literal_kind
| LitNull : literal_kind
| LitUndefined : literal_kind

and effect_kind =
| EffRead : loc:abstract_loc → effect_kind
| EffWrite : loc:abstract_loc → effect_kind
| EffAlloc : size:nat → effect_kind
| EffFree : loc:abstract_loc → effect_kind
| EffCall : target:string → effect_kind
| EffThrow : exn_type:string → effect_kind
| EffCatch : exn_type:string → effect_kind
| EffIO : io_kind → effect_kind
| EffSpawn : task_id:nat → effect_kind
| EffJoin : task_id:nat → effect_kind
| EffLock : lock_id:nat → effect_kind
| EffUnlock : lock_id:nat → effect_kind
| EffSend : chan:nat → effect_kind
| EffRecv : chan:nat → effect_kind
(* Channel effects --- Honda 1998/2008 session types *)
| EffChanCreate : chan:nat → elem_type:string → buffer_size:nat → effect_kind
| EffChanClose : chan:nat → effect_kind

```

```

| EffSelect : chan:nat → label:string → effect_kind
| EffBranch : chan:nat → labels:list string → effect_kind
| EffDelegate : chan:nat → delegated_chan:nat → effect_kind

and io_kind =
| IOFileRead | IOFileWrite
| IONetworkRead | IONetworkWrite
| IOStdin | IOStdout | IOStderr
| IOEnvRead | IOTimeRead | IORandom

```

6.5 CPG Data Structure

The CPG data structure organizes nodes and edges into an efficiently queryable form. The design prioritizes:

1. **Fast traversal:** Pre-computed indices for outgoing/incoming edges enable $O(1)$ edge lookup
2. **Label-based filtering:** Edges indexed by label type support efficient type-specific queries
3. **Source location mapping:** Bidirectional mapping between nodes and source locations enables IDE integration
4. **Language parameterization:** Each CPG carries its language configuration for language-aware analysis

Key type signatures:

- `node_id = nat`: Unique identifier for each node, enabling $O(1)$ lookup in hash maps
- `source_location`: Captures file path and line/column ranges for error reporting
- `cpg_node`: Combines node kind with metadata (source location, type info, abstract state)
- `cpg_edge`: Triple of source node, target node, and edge label
- `cpg`: The complete graph with indices for efficient traversal

The Code Property Graph Data Structure

```

(* =====
   THE CODE PROPERTY GRAPH
   ===== *)

module BrrrMachine.CPG

type node_id = nat

type source_location = {
  file : string;
  start_line : nat;
  start_col : nat;
  end_line : nat;
  end_col : nat;
}

type cpg_node = {
  id : node_id;
  kind : cpg_node_kind;
  source_loc : option source_location;
  (* Type information (when available) *)
  type_info : option type_info;
  (* Abstract state attached during analysis *)
  abstract_state : option abstract_state;
  (* Original source text (for error messages) *)
  source_text : option string;
}

type cpg_edge = {
  source : node_id;
  target : node_id;

```

```

    label : cpg_edge_label;
}

type cpg = {
  (* Core graph structure *)
  nodes : map node_id cpg_node;
  edges : list cpg_edge;

  (* Indices for efficient traversal *)
  outgoing : map node_id (list cpg_edge); (* node → outgoing edges *)
  incoming : map node_id (list cpg_edge); (* node → incoming edges *)
  by_label : map cpg_edge_label (list cpg_edge); (* label → edges *)

  (* Entry points *)
  functions : map string node_id; (* function name → entry node *)
  entry_points : list node_id; (* main functions, tests, handlers *)

  (* Reverse mappings *)
  node_by_loc : map source_location node_id; (* for IDE integration *)

  (* Language configuration *)
  language : language_config;
}

```

6.6 CPG Traversal Primitives

The power of the CPG is in its traversals. We define a small set of primitives that compose to express any analysis. This section formalizes the traversal algebra from Yamaguchi 2014 in F*, providing type-safe implementations of graph queries.

Core insight: By treating traversals as first-class functions on node sets, we can compose complex vulnerability queries from simple primitives using function composition.

Definition 6.2 (Traversal Algebra (Yamaguchi 2014, Section 7)). Traversals are *functions on node sets*:

$$T : \mathcal{P}(V) \rightarrow \mathcal{P}(V)$$

This enables compositional query construction via function composition. Complex vulnerability patterns are built by composing primitive traversals.

Key insight: Graph queries become first-class values that can be stored, composed, and reused. A traversal T applied to node set X yields a new node set $T(X)$.

CPG Traversal Primitives (Yamaguchi 2014 adapted)

```

(* =====
   CPG TRAVERSAL PRIMITIVES
   Source: Yamaguchi 2014 (adapted)
   ===== *)

module BrrrMachine.CPG.Traversal

  (* A traversal is a function from a set of nodes to a set of nodes *)
  type traversal = set node_id → set node_id

  (* Traversal composition: apply traversals in sequence (left-to-right) *)
  val compose : list traversal → traversal
  let compose traversals nodes =
    List.fold_left (fun ns t → t ns) nodes traversals

  (* Alternative: compose two traversals *)
  val compose2 : traversal → traversal → traversal
  let compose2 t1 t2 nodes = t2 (t1 nodes)

```

```
(* Identity traversal *)
let id_traversal : traversal = fun nodes → nodes
```

The traversal type `set node_id → set node_id` captures the essence of graph queries: given a starting set of nodes, produce the set of reachable nodes. The `compose` function enables building complex queries by chaining simpler ones.

6.6.1 TNODES: Tree Nodes (AST Subtree Collection)

Remark 6.3 (Yamaguchi 2014). $TNODES(V)$ returns all nodes in subtrees rooted at V . This is essential for syntax-based queries that need to examine entire expressions or statement subtrees.

TNODES Traversal

```
(* -----
   TNODES: Tree Nodes (AST Subtree Collection)
   Yamaguchi 2014: "TNODES(V) returns all nodes in subtrees rooted at V"
   ----- *)

val tnodes : cpg → traversal
let tnodes g roots =
  let rec collect_subtree node acc =
    (* Get all AST children of this node *)
    let children = out g (Set.singleton node) (LabelPrefix "AstChild") in
    (* Recursively collect all descendants *)
    Set.fold (fun child acc' → collect_subtree child acc')
      children
      (Set.add node acc)
  in
  Set.fold (fun root acc → collect_subtree root acc) roots Set.empty
```

6.6.2 MATCH: Find Nodes Matching Predicate in Subtrees

Remark 6.4 (Yamaguchi 2014). $MATCH_p(V) = FILTER_p(TNODES(V))$. Combines subtree collection with filtering to find specific patterns within AST subtrees. Essential for syntax-aware vulnerability queries.

MATCH Traversals

```
(* -----
   MATCH_p: Find Nodes Matching Predicate in Subtrees
   Yamaguchi 2014: "MATCH_p(V) = FILTER_p(TNODES(V))"
   ----- *)

val match_pred : cpg → (cpg_node → bool) → traversal
let match_pred g pred roots =
  filter g (tnodes g roots) pred

(* Convenience: match nodes by kind *)
val match_kind : cpg → cpg_node_kind → traversal
let match_kind g kind roots =
  match_pred g (fun n → n.kind = kind) roots

(* Convenience: match nodes by name pattern (regex) *)
val match_name : cpg → string → traversal
let match_name g pattern roots =
  match_pred g (fun n → matches_regex pattern (node_name n)) roots
```

The `match_pred` function demonstrates the composability principle: it combines `tnodes` (subtree collection) with `filter` (predicate testing) to find specific patterns within AST subtrees. This is the foundation for syntax-aware vulnerability queries.

6.6.3 Traversal Combinators

Traversal combinators provide set-theoretic operations on traversal results, enabling complex query patterns:

Traversal Combinators

```
(* -----
   TRAVERSAL COMBINATORS
   Higher-order functions for building complex traversals
   ----- *)

(* Union: nodes reachable by either traversal *)
val union_t : traversal → traversal → traversal
let union_t t1 t2 nodes = Set.union (t1 nodes) (t2 nodes)

(* Intersection: nodes reachable by both traversals *)
val inter_t : traversal → traversal → traversal
let inter_t t1 t2 nodes = Set.inter (t1 nodes) (t2 nodes)

(* Difference: nodes in first but not second *)
val diff_t : traversal → traversal → traversal
let diff_t t1 t2 nodes = Set.diff (t1 nodes) (t2 nodes)
```

6.6.4 Buffer Overflow Query Example

Example 6.5 (Buffer Overflow Detection (Yamaguchi 2014, Section 8.4)). This example demonstrates compositional query construction for detecting buffer overflows in write handlers where a count parameter reaches memcpy without bounds checking.

Pattern: Function parameter matching “count” flows to memcpy’s size argument without sanitization (bounds check, allocation check, min()).

Buffer Overflow Query (Yamaguchi Section 8.4)

```
(* -----
   EXAMPLE: Buffer Overflow Query (Yamaguchi 2014, Section 8.4)

   Demonstrates compositional query construction for detecting buffer
   overflows in write handlers where count parameter reaches memcpy
   without bounds checking.

   Pattern: Function parameter matching "count" flows to memcpy's size
   argument without sanitization (bounds check, allocation check, min()).
   ----- *)

val buffer_overflow_query : cpg → set node_id
let buffer_overflow_query g =
  (* Step 1: Find parameters with names suggesting count/size *)
  let count_params = filter g (all_nodes g) (fun n →
    is_parameter n && matches_regex ".*c(ou)?nt.*" (node_name n)) in

  (* Step 2: Find nodes that reach via data dependence *)
  let reaches_from_count =
    Set.fold (fun param acc →
      Set.union acc (reaches g param (LabelPrefix "DataDep"))
    ) count_params Set.empty in

  (* Step 3: Filter to memcpy size argument (position 2) *)
  let memcpy_size_args = filter g reaches_from_count (fun n →
    is_call_argument g n.id "memcpy" 2 ||
    is_call_argument g n.id "copy_from_user" 2) in

  (* Step 4: Exclude if sanitized (bounds check, alloc, min) *)
  let sanitizers = filter g (all_nodes g) (fun n →
    contains_bounds_check n ||
    is_allocation_with_param n ||
    node_contains_text n "min") in

  (* Step 5: Return unsanitized paths *)
  let sanitized = Set.fold (fun s acc →
```

```

    Set.union acc (reached_by g s (LabelPrefix "DataDep"))
  ) sanitizers Set.empty in

Set.diff memcpy_size_args sanitized

```

This query demonstrates the full power of compositional traversals:

1. **Source identification:** Filter for parameters matching a name pattern
2. **Reachability:** Follow data dependencies transitively
3. **Sink filtering:** Identify dangerous function arguments
4. **Sanitizer exclusion:** Remove paths that pass through bounds checks

The result is a precise set of potentially vulnerable code locations that require manual review.

6.6.5 Generic Vulnerability Query Structure

Generic Vulnerability Query

```

(* Generic vulnerability query structure *)
type vuln_query = {
  source : cpg → traversal;      (* Where tainted data originates *)
  sink   : cpg → traversal;      (* Where it must not reach unsanitized *)
  sanitizers : cpg → traversal; (* What makes data safe *)
}

val execute_vuln_query : cpg → vuln_query → set (node_id * node_id)
let execute_vuln_query g query =
  let sources = query.source g (all_nodes g) in
  let sinks   = query.sink   g (all_nodes g) in
  let safe    = query.sanitizers g (all_nodes g) in
  (* Find source-sink pairs where path ∃ without sanitizer *)
  reaches_set g sources sinks (LabelPrefix "DataDep")
  |> Set.filter (fun (src, snk) →
    let path_nodes = path_between g src snk (LabelPrefix "DataDep") in
    Set.is_empty (Set.inter path_nodes safe))

```

6.6.6 Basic Traversals

Basic Traversals

```

(* -----
   BASIC TRAVERSALS
   ----- *)

(* OUT: follow outgoing edges matching label pattern *)
val out : cpg → set node_id → label_pattern → set node_id
let out g nodes pattern =
  { e.target |
    e ← concat_map (fun n → Map.find_default n [] g.outgoing) (Set.to_list nodes),
    matches pattern e.label }

(* IN: follow incoming edges matching label pattern *)
val in_ : cpg → set node_id → label_pattern → set node_id
let in_ g nodes pattern =
  { e.source |
    e ← concat_map (fun n → Map.find_default n [] g.incoming) (Set.to_list nodes),
    matches pattern e.label }

(* FILTER: keep only nodes satisfying predicate *)
val filter : cpg → set node_id → (cpg_node → bool) → set node_id
let filter g nodes pred =
  Set.filter (fun n →
    match Map.find n g.nodes with
    | Some node → pred node
    | None     → false) nodes

```

```

(* MAP: transform node set *)
val map_nodes : cpg → set node_id → (cpg_node → 'a) → set 'a
let map_nodes g nodes f =
  Set.map (fun n →
    match Map.find n g.nodes with
    | Some node → f node
    | None → failwith "invalid node") nodes

```

6.6.7 Transitive Closures

Transitive Closure Traversals

```

(* -----
   TRANSITIVE CLOSURES
   ----- *)

(* REACHES: transitive closure following edge pattern *)
val reaches : cpg → node_id → label_pattern → set node_id
let reaches g start pattern =
  let rec go visited frontier =
    if Set.is_empty frontier then visited
    else
      let next = out g frontier pattern in
      let new_nodes = Set.diff next visited in
      go (Set.union visited new_nodes) new_nodes
  in
  go (Set.singleton start) (Set.singleton start)

(* REACHED_BY: reverse transitive closure *)
val reached_by : cpg → node_id → label_pattern → set node_id
let reached_by g target pattern =
  let rec go visited frontier =
    if Set.is_empty frontier then visited
    else
      let prev = in_ g frontier pattern in
      let new_nodes = Set.diff prev visited in
      go (Set.union visited new_nodes) new_nodes
  in
  go (Set.singleton target) (Set.singleton target)

(* REACHES_SET: from any source to any target *)
val reaches_set :
  cpg → set node_id → set node_id → label_pattern → set (node_id * node_id)
let reaches_set g sources targets pattern =
  let reachable_from_sources =
    Set.fold (fun src acc →
      Map.add src (reaches g src pattern) acc) sources Map.empty in
  { (src, tgt) |
    src ← Set.to_list sources,
    tgt ← Set.to_list targets,
    Set.mem tgt (Map.find_default src Set.empty reachable_from_sources) }

```

6.6.8 Program Slicing

Definition 6.6 (Program Slicing (Weiser 1984, Horwitz 1990)). The following concepts map to CPG terminology:

Slicing Criterion A tuple $C = (i, V)$ where i is a statement and V is a set of variables. The slice preserves behavior at i for variables V .

REF(n) / DEF(n) Variables referenced/defined at statement n :

- $\text{REF}(n) \rightarrow$ Read effects, `get_used_vars` in CPG
- $\text{DEF}(n) \rightarrow$ Write effects, `get_defined_var` in CPG
- `DataDep` edges encode the REF/DEF relationship directly

INFL(b) The influence range of branch b —statements whose execution depends on b 's outcome.

Captured by **CtrlDep** edges in CPG.

Theorem 6.7 (Undecidability (Weiser 1984, Theorem 1)). *Finding statement-minimal slices is undecidable—equivalent to the halting problem.*

Consequence: We compute conservative approximations via dataflow analysis. Our slices may include extra statements but will **never** exclude statements that affect the slicing criterion.

Remark 6.8 (Algorithm Correspondence). • Weiser’s $R_C^0(n)$ computation \rightarrow **backward_slice** with **DataDep** edges

- Weiser’s B_C^i (branch influence) \rightarrow **CtrlDep** edge following
- Weiser’s fixpoint $S_C = \bigcup S_C^i \rightarrow$ **reached_by** transitive closure

Cross-reference: Section 4.1 (IFDS) for context-sensitive slicing; Section 8.1 (Taint Analysis) uses forward slicing.

The following F* code implements the slicing algorithms. The key functions are:

- **backward_slice**: Find all nodes that could affect a given criterion (for debugging, understanding)
- **forward_slice**: Find all nodes that could be affected by a criterion (for impact analysis)
- **thin_slice**: A more precise backward slice following only relevant dependencies (from TAJ, Tripp 2009)

Slicing Traversals

```
(* -----
   SLICING (from Weiser 1984, Horwitz 1990)
   ----- *)

(* BACKWARD_SLICE: all nodes that could affect the slicing criterion *)
val backward_slice : cpg → set node_id → set node_id
let backward_slice g criteria =
  (* Follow DATA_DEP and CTRL_DEP edges backwards *)
  let dep_pattern = LabelOr [LabelPrefix "DataDep"; LabelPrefix "CtrlDep"] in
  reached_by_set g criteria dep_pattern

(* FORWARD_SLICE: all nodes that could be affected by the slicing criterion *)
val forward_slice : cpg → set node_id → set node_id
let forward_slice g criteria =
  let dep_pattern = LabelOr [LabelPrefix "DataDep"; LabelPrefix "CtrlDep"] in
  reaches_set g criteria dep_pattern

(* THIN_SLICE: backward slice following only relevant dependencies
   Source: Tripp 2009 (TAJ) *)
val thin_slice : cpg → node_id → string → set node_id
let thin_slice g sink sink_arg =
  (* Only follow dependencies that affect the sink argument *)
  let rec go visited frontier relevant_vars =
    if Set.is_empty frontier then visited
    else
      let preds = in_ g frontier (LabelPrefix "DataDep") in
      let relevant_preds = Set.filter (fun n →
        (* Node defines a relevant variable *)
        match get_defined_var g n with
        | Some v → Set.mem v relevant_vars
        | None → false) preds in
      let new_vars = Set.fold (fun n acc →
        Set.union acc (get_used_vars g n)) relevant_preds Set.empty in
      let new_nodes = Set.diff relevant_preds visited in
      go (Set.union visited new_nodes) new_nodes (Set.union relevant_vars new_vars)
  in
  go (Set.singleton sink) (Set.singleton sink) (Set.singleton sink_arg)
```

6.6.9 Two-Phase Interprocedural Slicing

Critical: Horwitz 1990

CRITICAL: Weiser’s transitive closure slicing is **WRONG** for interprocedural analysis!
The problem: Transitive closure can produce *spurious* results by “descending into a called procedure and then ascending to an incorrect calling context.”

Example:

- Main calls A, which calls Add
- Main also calls A, which calls Increment, which calls Add

If slicing from Increment, Weiser incorrectly includes the first Main→A→Add path because it doesn’t track calling contexts.

THE FIX: Two-phase slicing with restricted edge following.

Two-Phase Interprocedural Slicing (Horwitz 1990)

```
(* -----  
TWO-PHASE INTERPROCEDURAL SLICING (from Horwitz 1990)  
  
CRITICAL: Weiser's transitive closure slicing is WRONG for interprocedural!  
  
THE FIX: Two-phase slicing with restricted edge following.  
----- *)  
  
(* PHASE 1: Ascend to callers --- DON'T follow param-out edges *)  
let slice_phase1 (g : cpg) (criteria : set node_id) : set node_id =  
  let exclude = [ParamOut; DefOrder] in  
  reach_backwards g criteria exclude  
  
(* PHASE 2: Descend into callees --- DON'T follow call/param-in edges *)  
let slice_phase2 (g : cpg) (phase1_nodes : set node_id) : set node_id =  
  let exclude = [Calls; ParamIn; DefOrder] in  
  reach_backwards g phase1_nodes exclude  
  
(* Combined two-phase interprocedural slice *)  
val interprocedural_slice : cpg → set node_id → set node_id  
let interprocedural_slice g criteria =  
  let phase1 = slice_phase1 g criteria in  
  let phase2 = slice_phase2 g phase1 in  
  Set.union phase1 phase2  
  
(* WHY THIS WORKS:  
Phase 1 can ASCEND from a called procedure to its callers via  
call/param-in edges, but cannot DESCEND via param-out.  
  
Phase 2 can DESCEND into called procedures via param-out, but  
cannot ASCEND via call/param-in.  
  
This ensures we never:  
- Descend into procedure P  
- Then ascend to caller Q (that didn't actually call P in our context)  
  
The result is a CONTEXT-SENSITIVE slice: only nodes on REALIZABLE  
paths (matched call/return parentheses) are included.  
*)  
  
(* FORWARD SLICE (dual): for impact analysis *)  
val interprocedural_forward_slice : cpg → set node_id → set node_id  
let interprocedural_forward_slice g criteria =  
  (* Phase 1: Forward in callers, don't descend via param-in/call *)  
  let phase1 = reach_forward g criteria [ParamIn; Calls; DefOrder] in
```

```

(* Phase 2: Forward into callees, don't ascend via param-out *)
let phase2 = reach_forward g phase1 [ParamOut; DefOrder] in
Set.union phase1 phase2

```

```

(* See Section 12.9 for soundness proofs *)

```

The two-phase algorithm is essential for **context-sensitive** interprocedural analysis. It ensures that only *realizable paths*—those with matched call/return parentheses—are included in the slice. This prevents the spurious results that plague naive transitive closure approaches.

Connection to IFDS: The two-phase restriction corresponds to respecting the Dyck language of matched parentheses in CFL-reachability (see Section 4.1).

6.6.10 Taint Analysis Traversals

Taint Analysis Traversals (Livshits 2005, Tripp 2009)

```

(* -----
   Taint Analysis Traversals
   Source: Livshits 2005, Tripp 2009
   ----- *)

(* UNSANITIZED_PATHS: find source-to-sink paths avoiding sanitizers *)
val unsanitized_paths :
  cpg →
  sources:set node_id →
  sinks:set node_id →
  sanitizers:set node_id →
  list (list node_id) (* Paths from source to sink *)

let unsanitized_paths g sources sinks sanitizers =
  (* BFS from sources, stopping at sanitizers *)
  let rec bfs_paths frontier visited paths =
    if Set.is_empty frontier then paths
    else
      let next_edges = concat_map (fun n → Map.find_default n [] g.outgoing) frontier in
      let data_edges = List.filter (fun e →
        match e.label with DataDep _ → true | _ → false) next_edges in
      let next_nodes = Set.of_list (List.map (fun e → e.target) data_edges) in
      let unsanitized = Set.diff next_nodes sanitizers in
      let new_nodes = Set.diff unsanitized visited in
      (* Record complete paths *)
      let complete = Set.inter new_nodes sinks in
      let new_paths = (* ... reconstruct paths ... *) in
      bfs_paths new_nodes (Set.union visited new_nodes) (paths @ new_paths)
  in
  bfs_paths sources sources []

```

6.6.11 Label Patterns

Label Patterns for Flexible Edge Matching

```

(* -----
   LABEL PATTERNS --- for flexible edge matching
   ----- *)

type label_pattern =
  | LabelExact : cpg_edge_label → label_pattern
  | LabelPrefix : string → label_pattern (* e.g., "DataDep" matches DataDep(_) *)
  | LabelAny : label_pattern
  | LabelOr : list label_pattern → label_pattern
  | LabelAnd : list label_pattern → label_pattern
  | LabelNot : label_pattern → label_pattern

let rec matches (pattern : label_pattern) (label : cpg_edge_label) : bool =
  match pattern with

```

```

| LabelExact l → label = l
| LabelPrefix prefix → String.is_prefix prefix (label_to_string label)
| LabelAny → true
| LabelOr patterns → List.∃ (fun p → matches p label) patterns
| LabelAnd patterns → List.for_all (fun p → matches p label) patterns
| LabelNot p → not (matches p label)

```

6.7 CPG Construction

This section describes the algorithm for constructing a Code Property Graph from source code. The construction proceeds in phases, building each component graph layer by layer.

Construction phases:

1. **Parsing:** Convert source to AST, create AST nodes and edges
2. **CFG construction:** Add control flow edges between statements
3. **Dominator computation:** Required for control dependence
4. **PDG construction:** Add data and control dependence edges
5. **Call graph:** Add interprocedural edges
6. **Effect graph:** Add side effect tracking edges
7. **Indexing:** Build efficient lookup structures

Critical Warning: Phased vs. Interleaved Construction

The phased approach below is **ONLY** valid for languages without virtual dispatch (C, basic procedural code). For OOP languages (Java, Python, JS, C++), call graph construction **MUST** be interleaved with pointer analysis.

See Section 5.3 (On-the-Fly Call Graph Construction) for the correct approach for dynamic dispatch languages.

Source: Qilin (He, Lu, Xue 2022)—“pointer analysis and call graph construction are mutually dependent and must be solved together”

CPG Construction Algorithm (Phased—for non-OOP languages only)

```

(* =====
   CPG CONSTRUCTION ALGORITHM (PHASED - for non-OOP languages only)
   WARNING: For OOP/dynamic dispatch, use interleaved approach in Section 5.3
   ===== *)

module BrrrMachine.CPG.Builder

(* Main construction pipeline - PHASED VERSION *)
(* Use build_cpg_interleaved for Java/Python/JS/C++ *)
val build_cpg : language_config → source_file → cpg
let build_cpg lang source =
  (* Phase 1: Parse to AST and create AST nodes/edges *)
  let ast = parse lang source in
  let cpg = create_ast_nodes ast in

  (* Phase 2: Build CFG edges *)
  let cpg = build_cfg_edges cpg in

  (* Phase 3: Compute dominators for control dependence *)
  let dom = compute_dominators cpg in
  let pdom = compute_post_dominators cpg in

  (* Phase 4: Build PDG edges *)
  let cpg = build_control_dep_edges cpg pdom in
  let cpg = build_data_dep_edges cpg in

  (* Phase 5: Build call graph edges - STATIC ONLY *)

```

```

(* WARNING: This misses virtual/dynamic calls! *)
let cpg = build_static_call_edges cpg in

(* Phase 6: Build effect edges *)
let cpg = build_effect_edges cpg lang in

(* Phase 7: Create indices *)
let cpg = build_indices cpg in

cpg

```

6.7.1 Control Dependence Construction

Control dependence captures which statements' execution depends on which branch decisions. This is essential for program slicing: if a statement is control-dependent on a branch, changing the branch outcome could affect whether that statement executes.

Definition 6.9 (Control Dependence (Ferrante 1987)). Node Y is *control-dependent* on edge (A, B) if:

1. Y post-dominates B
2. Y does not strictly post-dominate A

Intuition: Y executes only if the $A \rightarrow B$ branch is taken.

Control Dependence Edge Construction (Ferrante 1987)

```

(* -----
   CONTROL DEPENDENCE (Ferrante 1987)

   Definition: Node Y is control-dependent on edge (A,B) if:
     1. Y post-dominates B
     2. Y does not strictly post-dominate A

   Intuition: Y executes only if the A→B branch is taken.
   ----- *)

val build_control_dep_edges : cpg → post_dominators → cpg
let build_control_dep_edges cpg pdom =
  (* For each CFG edge (A,B) *)
  fold_edges cpg (fun cpg edge →
    match edge.label with
    | CfgTrue | CfgFalse | CfgNext →
      let a = edge.source in
      let b = edge.target in
      let branch = (edge.label = CfgTrue) in
      (* Find least common ancestor in post-dominator tree *)
      let lca = pdom_lca pdom a b in
      (* All nodes on path from B to LCA (exclusive) are control-dependent on A *)
      let path = pdom_path pdom b lca in
      let ctrl_dep_nodes = List.filter (fun n → n ≠ lca) path in
      (* Add control dependence edges *)
      List.fold_left (fun cpg n →
        add_edge cpg { source = a; target = n; label = CtrlDep branch }
      ) cpg ctrl_dep_nodes
    | _ → cpg
  ) cpg

```

6.7.2 Data Dependence Construction

Data dependence captures which statements' values flow to which other statements. This is the foundation for taint analysis: if statement X defines a value that statement Y uses, then taint at X propagates to Y .

Definition 6.10 (Data Dependence). Node Y is *data-dependent* on node X through variable v if:

1. X defines v
2. Y uses v
3. There is a CFG path from X to Y with no intervening definition of v

Data Dependence Edge Construction

```
(* -----
  DATA DEPENDENCE (via reaching definitions)

  Definition: Node Y is data-dependent on node X through variable v if:
    1. X defines v
    2. Y uses v
    3. There is a CFG path from X to Y with no intervening definition of v
  ----- *)

val build_data_dep_edges : cpg → cpg
let build_data_dep_edges cpg =
  (* Compute reaching definitions for each node *)
  let reach_defs = compute_reaching_definitions cpg in
  (* For each node that uses a variable *)
  fold_nodes cpg (fun cpg node →
    let uses = get_used_vars cpg node.id in
    Set.fold (fun var cpg →
      (* Find reaching definitions of this variable *)
      let defs = get_reaching_defs reach_defs node.id var in
      (* Add data dependence edge from each definition *)
      Set.fold (fun def_node cpg →
        add_edge cpg { source = def_node; target = node.id; label = DataDep var }
      ) defs cpg
    ) uses cpg
  ) cpg
```

6.7.3 Reaching Definitions

Reaching definitions is the classic dataflow problem that computes, for each program point, which definitions of each variable could reach that point without being killed by an intervening redefinition.

Transfer function: $out = gen \cup (in \setminus kill)$

The **gen** set contains the definition at the current node (if any), while **kill** removes any prior definitions of the same variable. The algorithm iterates to a fixpoint, propagating definitions along CFG edges.

Reaching Definitions (Classic Dataflow)

```
(* -----
  REACHING DEFINITIONS (classic dataflow)
  ----- *)

type def = { node : node_id; var : string }
type reaching_defs = map node_id (set def)

val compute_reaching_definitions : cpg → reaching_defs
let compute_reaching_definitions cpg =
  (* Transfer function: out = gen  $\cup$  (in  $\setminus$  kill) *)
  let transfer node in_set =
    let gen = match get_defined_var cpg node with
      | Some v → Set.singleton { node = node; var = v }
      | None → Set.empty in
    let kill = match get_defined_var cpg node with
      | Some v → Set.filter (fun d → d.var ≠ v) in_set
      | None → Set.empty in
    Set.union gen (Set.diff in_set kill)
  in
  (* Fixpoint iteration *)
```

```

let rec iterate current =
  let next = Map.mapi (fun node _ →
    let preds = get_cfg_predecessors cpg node in
    let in_set = Set.unions (List.map (fun p →
      Map.find_default p Set.empty current) preds) in
    transfer node in_set
  ) current in
  if Map.equal Set.equal current next then current
  else iterate next
in
let init = Map.map (fun _ → Set.empty) cpg.nodes in
iterate init

```

6.8 Neural Program Graph Embeddings (Optional Enhancement)

This section describes optional neural network enhancements to the CPG that enable learning-based analysis. While the core CPG provides a sound foundation for static analysis, neural embeddings can capture patterns that are difficult to express with hand-crafted rules.

Paper: Si et al. 2018 (Code2Inv—NeurIPS)

GNN-Based Program Representation

INSIGHT: The CPG already encodes AST, CFG, and data flow edges. A Graph Neural Network (GNN) can learn *dense vector embeddings* that capture semantic patterns beyond what static traversals can express.

ARCHITECTURE (Code2Inv):

1. Convert program to SSA form (explicit data dependencies)
2. Build program graph with 6 edge types:
 - AST parent/child (bidirectional)
 - CFG forward/backward (bidirectional)
 - Data flow (SSA connections, bidirectional)
3. Message passing with edge-type-specific weight matrices
4. GRU aggregation for each node embedding update

GNN Update Rule (each round t):

$$h_v^{(t+1)} = \text{GRU} \left(h_v^{(t)}, \sum_{(u,v,k)} W_k \cdot h_u^{(t)} \right)$$

where W_k is the weight matrix for edge type k .

USE CASES:

- Loop invariant synthesis (Section 2.1.5c)
- Bug pattern detection via learned embeddings
- Semantic code search and similarity

Neural Program Graph Embedding (Si et al. 2018)

```

(* =====
  NEURAL PROGRAM GRAPH EMBEDDING
  Source: Si et al. 2018 (Code2Inv)
  ===== *)

module BrrrMachine.CPG.NeuralEmbedding

(* Embedding dimension - typically 64 or 128 *)
let embedding_dim : nat = 64

```

```

type node_embedding = vec float embedding_dim

(* Edge types for GNN message passing - extends cp_g edge_label *)
type gnn_edge_type =
| GNNAstChild : gnn_edge_type      (* AST parent to child *)
| GNNAstParent : gnn_edge_type     (* AST child to parent (reverse) *)
| GNNCfNext : gnn_edge_type        (* Control flow forward *)
| GNNCfPrev : gnn_edge_type        (* Control flow backward *)
| GNNDataFlow : gnn_edge_type      (* SSA data dependency *)
| GNNDataFlowRev : gnn_edge_type   (* Reverse data dependency *)

(* GNN parameters - one weight matrix per edge type *)
type gnn_params = {
  edge_weights : map gnn_edge_type (matrix embedding_dim embedding_dim);
  gru_weights : gru_params embedding_dim;
  token_embeds : map token_type node_embedding; (* Initial node embeddings *)
}

(* Program embedding: all nodes + global graph summary *)
type program_embedding = {
  node_embeds : map node_id node_embedding;
  global_embed : node_embedding; (* Aggregated graph representation *)
}

(* Convert CPG edge labels to GNN edge types *)
val cp_g_to_gnn_edge : cp_g edge_label → bool → gnn_edge_type
let cp_g_to_gnn_edge label is_reverse =
  match label, is_reverse with
  | AstChild _, false → GNNAstChild
  | AstChild _, true → GNNAstParent
  | CfNext, false | CfTrue, false | CfFalse, false → GNNCfNext
  | CfNext, true | CfTrue, true | CfFalse, true → GNNCfPrev
  | DataDep _, false → GNNDataFlow
  | DataDep _, true → GNNDataFlowRev
  | _ → GNNCfNext (* Default for other edge types *)

(* Single round of message passing *)
val message_pass_round :
  cp_g:cp_g →
  params:gnn_params →
  current:map node_id node_embedding →
  map node_id node_embedding

let message_pass_round cp_g params current =
  Map.mapi (fun nid _ →
    (* Aggregate messages from all incoming edges *)
    let incoming = get_all_incoming_edges cp_g nid in
    let message = List.fold_left (fun acc (src, label) →
      let etype = cp_g_to_gnn_edge label true in
      let w = Map.find etype params.edge_weights in
      let h_src = Map.find src current in
      vec_add acc (mat_vec_mul w h_src)
    ) (zero_vec embedding_dim) incoming in
    (* GRU update *)
    gru_cell params.gru_weights (Map.find nid current) message
  ) current

(* Full encoding with T message passing rounds *)
val encode_program :
  cp_g:cp_g →
  params:gnn_params →
  rounds:nat{rounds > 0} →

```

```

program_embedding

let encode_program cpg params rounds =
  (* Initialize embeddings from token types *)
  let initial = Map.mapi (fun nid node →
    let token = node_to_token node in
    Map.find_default token (zero_vec embedding_dim) params.token_embeds
  ) cpg.nodes in
  (* T rounds of message passing *)
  let final = iterate_n rounds (message_pass_round cpg params) initial in
  (* Global embedding via max pooling *)
  let global = Map.fold (fun _ emb acc → vec_max acc emb)
    final (zero_vec embedding_dim) in
  { node_embeddings = final; global_embed = global }

(* Attention over program nodes - for invariant synthesis *)
val compute_attention :
  query:node_embedding →
  program:program_embedding →
  map node_id float (* Attention weights sum to 1 *)

let compute_attention query program =
  let scores = Map.map (fun emb → vec_dot query emb) program.node_embeddings in
  softmax scores

```

The F* code above defines a complete GNN-based program embedding system:

- **gnn_edge_type**: Maps CPG edge labels to GNN-compatible edge types (with bidirectional variants)
- **message_pass_round**: Single iteration of message passing with edge-type-specific weights
- **encode_program**: Full encoding with T rounds of message passing
- **compute_attention**: Attention mechanism for focusing on relevant program nodes

Key insight: The GNN learns to aggregate information from a node’s graph neighborhood, capturing semantic relationships that depend on both local syntax and global program structure.

Remark 6.11 (Integration with CPG). • GNN embeddings are an **optional enhancement** to the static CPG

- The CPG provides the graph structure; GNN learns dense representations
- Pre-trained embeddings can be loaded for transfer learning
- Use embeddings for neural invariant synthesis (Section 2.1.5c)

Remark 6.12 (When to Use GNN Embeddings). • Learning-based analysis (invariant synthesis, bug detection)

- Semantic code similarity/search
- When static patterns are insufficient

6.8.1 Semantic Identifier Embeddings (DeepBugs)

While GNN embeddings capture program structure, identifier names carry crucial semantic information that traditional static analysis ignores. This section describes how to leverage name semantics for bug detection.

Source: Pradel & Sen 2018—“DeepBugs: A Learning Approach to Name-Based Bug Detection” (OOPSLA)

Name-Aware Semantic Embeddings for Bug Detection

INSIGHT: Traditional static analysis **ignores** identifier names. But names encode *semantic intent* that can reveal bugs:

- `setTimeout(delay, callback)` vs `setTimeout(callback, delay)`
- `height + width (bug?)` vs `height + height`
- `j < tokens.length` (should be `j < token.length?`)

DEEPBUGS APPROACH:

1. Create *semantic embeddings* for identifiers via `word2vec`
2. Split identifiers into subtokens: `getUserName` → `[get, User, Name]`
3. Train binary classifiers: correct code vs synthetic bugs
4. Detect swapped arguments, wrong operators, wrong operands

RESULTS: 89–95% accuracy, 68% true positive rate on real bugs

SPEED: <20ms per file (pre-computed embeddings + matrix ops)

Semantic Identifier Embeddings (Pradel & Sen 2018)

```
(* =====
  SEMANTIC IDENTIFIER EMBEDDINGS
  Source: Pradel & Sen 2018 (DeepBugs)

  Capture semantic meaning of identifiers via word2vec-style embeddings.
  "src" and "source" have similar embeddings; "src" and "dst" are opposites.
  ===== *)

module BrrrMachine.CPG.NameEmbeddings

(* -----
  SUBTOKEN SPLITTING

  Split identifiers into semantically meaningful subtokens:
  - camelCase: getUserName → [get, user, name]
  - snake_case: user_name → [user, name]
  - Combined: getUser_ID → [get, user, id]
  ----- *)

val split_camel_case : string → list string
let split_camel_case name =
  (* Split on case transitions: "getUserName" → ["get", "User", "Name"] *)
  regex_split "[A-Z]" name |> List.map String.lowercase

val split_snake_case : string → list string
let split_snake_case name =
  String.split "_" name |> List.filter (fun s → String.length s > 0)

val split_identifier : string → list string
let split_identifier name =
  let camel = split_camel_case name in
  let snake = split_snake_case name in
  List.deduplicate (camel @ snake) |> List.map String.lowercase

(* -----
  EMBEDDING MODEL

  Pre-trained word2vec model maps subtokens to dense vectors.
  Identifier embedding = average of subtoken embeddings.
  ----- *)

type embedding_dim = n:nat{n > 0}
```

```

type embedding (dim:embedding_dim) = vector float dim

noeq type name_embedding_model (dim:embedding_dim) = {
  subtoken_embeddings : map string (embedding dim);
  oov_embedding : embedding dim;  (* Out-of-vocabulary fallback *)
}

val embed_identifier :
  #dim:embedding_dim →
  model:name_embedding_model dim →
  name:string →
  embedding dim

let embed_identifier #dim model name =
  let subtokens = split_identifier name in
  let embeddings = List.filter_map (fun st →
    Map.find_opt st model.subtoken_embeddings
  ) subtokens in
  if List.is_empty embeddings then
    model.oov_embedding
  else
    vector_average embeddings

```

The `split_identifier` function handles common naming conventions (camelCase, snake_case) to extract meaningful subtokens. The `embed_identifier` function averages subtoken embeddings, falling back to an out-of-vocabulary embedding for unknown tokens.

Identifier Similarity

```

(* Cosine similarity between identifier embeddings *)
val name_similarity :
  #dim:embedding_dim →
  model:name_embedding_model dim →
  name1:string → name2:string →
  float

let name_similarity #dim model name1 name2 =
  let e1 = embed_identifier model name1 in
  let e2 = embed_identifier model name2 in
  cosine_similarity e1 e2

```

Name-Based Bug Detectors

DeepBugs defines three detector types, each trained on synthetically generated bugs. The detectors extract identifier embeddings from code locations and classify whether the code is likely buggy.

Detector types:

1. **Swapped Arguments:** Detects when function arguments may be in the wrong order based on name semantics
2. **Wrong Binary Operator:** Detects when an operator may be incorrect (e.g., `==` vs `!=`)
3. **Wrong Binary Operand:** Detects when a variable may be the wrong one based on name similarity

Name-Based Bug Detectors

```

(* -----
   NAME-BASED BUG DETECTORS

   DeepBugs defines three detector types, each trained on synthetic bugs:
   1. Swapped Arguments: setTimeout(delay, fn) vs setTimeout(fn, delay)
   2. Wrong Binary Operator: x == y vs x != y
   3. Wrong Binary Operand: height + width vs height + height
   ----- *)

```

```

type ml_prediction = {
  is_bug : bool;
  confidence : float;  (* 0.0 to 1.0 *)
  bug_type : string;
  explanation : option string;
}

(* Feature extraction for call sites *)
val extract_call_features :
  #dim:embedding_dim →
  model:name_embedding_model dim →
  cpg:cpg →
  call_node:node_id →
  option (list (embedding dim))

let extract_call_features #dim model cpg call_node =
  match get_node cpg call_node with
  | Some (NCall callee args) →
    let callee_emb = embed_identifier model callee in
    let arg_embs = List.map (fun arg →
      match get_identifier_name cpg arg with
      | Some name → embed_identifier model name
      | None → model.oov_embedding
    ) args in
    Some (callee_emb :: arg_embs)
  | _ → None

(* Feature extraction for binary operations *)
val extract_binop_features :
  #dim:embedding_dim →
  model:name_embedding_model dim →
  cpg:cpg →
  expr_node:node_id →
  option (list (embedding dim))

let extract_binop_features #dim model cpg expr_node =
  match get_node cpg expr_node with
  | Some (NBinOp left op right) →
    let left_emb = match get_identifier_name cpg left with
      | Some name → embed_identifier model name
      | None → model.oov_embedding in
    let right_emb = match get_identifier_name cpg right with
      | Some name → embed_identifier model name
      | None → model.oov_embedding in
    let op_emb = embed_operator model op in
    Some [left_emb; op_emb; right_emb]
  | _ → None

```

Synthetic Bug Generation for Training

A key insight from DeepBugs is that training data can be generated by simple mutations of correct code. Correct code serves as positive examples; mutated code serves as negative examples. This avoids the need for expensive manual bug labeling.

Synthetic Bug Generation for Training

```

(* -----
   SYNTHETIC BUG GENERATION FOR TRAINING

   Key insight: Generate training data by simple mutations.
   Correct code = positive examples; mutated code = negative examples.
   ----- *)

type mutation_type =

```

```

| SwapArguments : idx1:nat → idx2:nat → mutation_type
| ReplaceOperator : new_op:binop → mutation_type
| ReplaceOperand : side:bool → new_var:string → mutation_type

type training_example = {
  features : list (embedding embedding_dim);
  label : bool; (* true = correct, false = buggy *)
  mutation : option mutation_type;
}

val generate_training_examples :
  cpg:cpg →
  model:name_embedding_model embedding_dim →
  detector_type:string →
  list training_example

let generate_training_examples cpg model detector_type =
  match detector_type with
  | "swapped_args" →
    (* Find all call sites, generate original + swapped versions *)
    let calls = find_all_calls cpg in
    List.concat_map (fun call →
      match extract_call_features model cpg call with
      | Some features when List.length features ≥ 3 →
        let original = { features; label = true; mutation = None } in
        let swapped = { features = swap_in_list features 1 2;
          label = false;
          mutation = Some (SwapArguments 0 1) } in
        [original; swapped]
      | _ → []
    ) calls
  | "wrong_operator" →
    let binops = find_all_binops cpg in
    List.concat_map (fun binop →
      match extract_binop_features model cpg binop with
      | Some features →
        let original = { features; label = true; mutation = None } in
        (* Generate mutants with different operators *)
        let mutants = List.map (fun new_op →
          { features = replace_op_embedding features new_op;
            label = false;
            mutation = Some (ReplaceOperator new_op) }
        ) (alternative_operators (get_operator binop)) in
        original :: mutants
      | None → []
    ) binops
  | _ → []

```

Name-Based Security Role Classification

Function names often indicate their security relevance. Names like “sanitize”, “escape”, and “encode” strongly suggest sanitizer functions, while “input”, “request”, and “param” suggest sources. This section shows how to leverage name embeddings for automatic security role classification.

Application: Augment taint analysis configuration by discovering likely sanitizers that were not explicitly configured.

Name-Based Security Role Classification

```

(* -----
  NAME-BASED SECURITY ROLE CLASSIFICATION

  Names often indicate security relevance:
  - "sanitize", "escape", "encode" → likely sanitizer
  - "input", "request", "param" → likely source

```

```

- "query", "exec", "eval" → likely sink
----- *)

type security_role =
| LikelySource
| LikelySink
| LikelySanitizer
| LikelyValidator
| SecurityNeutral

(* Canonical security term embeddings *)
noeq type security_classifier (dim:embedding_dim) = {
  source_centroid : embedding dim;      (* avg of "input", "request", "param" *)
  sink_centroid : embedding dim;        (* avg of "query", "exec", "eval" *)
  sanitizer_centroid : embedding dim;    (* avg of "escape", "sanitize", "encode" *)
  validator_centroid : embedding dim;    (* avg of "validate", "check", "verify" *)
  threshold : float;
}

val classify_security_role :
#dim:embedding_dim →
classifier:security_classifier dim →
model:name_embedding_model dim →
func_name:string →
option (security_role * float)

let classify_security_role #dim classifier model func_name =
  let func_emb = embed_identifier model func_name in
  let similarities = [
    (LikelySource, cosine_similarity func_emb classifier.source_centroid);
    (LikelySink, cosine_similarity func_emb classifier.sink_centroid);
    (LikelySanitizer, cosine_similarity func_emb classifier.sanitizer_centroid);
    (LikelyValidator, cosine_similarity func_emb classifier.validator_centroid);
  ] in
  let (best_role, best_sim) = List.max_by snd similarities in
  if best_sim ≥ classifier.threshold then
    Some (best_role, best_sim)
  else
    None

(* Augment taint configuration with name-based hints *)
val augment_taint_config :
#dim:embedding_dim →
classifier:security_classifier dim →
model:name_embedding_model dim →
cpg:cpg →
base_config:taint_config →
taint_config

let augment_taint_config #dim classifier model cpg base_config =
  let functions = get_all_functions cpg in
  let classified = List.filter_map (fun f →
    classify_security_role classifier model f.name
  ) functions in
  let new_sanitizers = List.filter_map (fun (f, role, _) →
    if role = LikelySanitizer then Some f.id else None
  ) (List.zip functions classified) in
  { base_config with
    sanitizers = Set.union base_config.sanitizers (Set.of_list new_sanitizers) }

```

- Remark 6.13** (Integration with Taint Analysis (Section 8.1.2)). • Use name-based classification to discover likely sanitizers automatically
- Augment configured sanitizer list with name-inferred candidates

- Report confidence level based on embedding similarity score

Remark 6.14 (Integration with Static Analysis (Section 13.4 Layer 5.5)). • ML predictions feed into confidence calculation at Layer 6

- Combine static analysis violations with ML bug predictions
- Higher confidence when static + ML agree

Remark 6.15 (Cross-References). • Section 2.1.5c: Neural invariant synthesis (primary consumer)

- Section 3.1.2: CPG components (provides graph structure)
- Section 3.1.3: Edge types (mapped to GNN edge types)
- Section 8.1.2: Taint analysis (name-based sanitizer detection)
- Section 4.4.7: Optimistic concolic (ML can prioritize paths)

Part IV

Analysis Algorithms

Critical Tension: IFDS vs Bi-Abduction/Eval

IFDS (Section 7):

- Requires DISTRIBUTIVE transfer functions: $f(a \sqcup b) = f(a) \sqcup f(b)$
- Tracks DATAFLOW FACTS
- \mathcal{OED}^3 guaranteed complexity
- Path-INSENSITIVE

Bi-Abduction/Eval (Section 9):

- NOT distributive
- Multiple valid (M, F) pairs
- Tracks SEPARATION LOGIC assertions
- May be exponential
- Path-SENSITIVE

These are different algorithms for different problems:

- Taint analysis \rightarrow IFDS (fast, whole-program)
- Shape/memory analysis \rightarrow Eval (precise, local)
- Hybrid: IFDS finds candidates, Eval verifies (Section 9.4)

DO NOT try to force bi-abduction into IFDS framework! The mathematical properties are incompatible.

Tension Resolution: IFDS vs Set Constraints

Source: Aiken 1999. See Appendix D.10.1 for full analysis.

Insight: IFDS is a RESTRICTED FRAGMENT of set constraints.

- IFDS requires distributive transfer functions
- Set constraints (Section 12.18) handle non-distributive cases
- Pointer analysis is NOT distributive—cannot use IFDS directly

Resolution:

- Use IFDS for taint, reaching definitions, live variables (distributive)
- Use set constraints for type inference, pointer analysis (general)
- Section 12.18 provides unified constraint framework

Tension Resolution: Datalog Interpretation vs Compilation

Sources: Jordan 2016 (Souffle), Madsen 2016 (Flix)

Old View: “Don’t build custom Datalog engine, use existing or skip”

New View: COMPILE Datalog rules to specialized code, don’t interpret

Jordan 2016 demonstrates: Souffle achieves 50x+ speedup over bddbldb by eliminating interpretation overhead via staged specialization:

1. Stage 1: Datalog \rightarrow RAM (Relational Algebra Machine)
2. Stage 2: RAM \rightarrow Optimized RAM (Dilworth indices, join ordering)
3. Stage 3: RAM \rightarrow C++/Rust (specialized, parallel)

Resolution for brrr-machine:

- Development: Use interpreted Datalog (Crepe, Souffle -interpreter)
- Production: Compile analysis rules to specialized Rust
- See Section 7.6 for compilation strategy details

Key Insight: Flix Unifies IFDS/IDE/Value Analysis

Source: Madsen 2016

Standard Datalog: Relations (finite sets of tuples)

Flix Extension: Lattice predicates (map from keys to lattice elements)

This unifies:

- IFDS = Flix where lattice is powerset of dataflow facts
- IDE = Flix where lattice is micro-function space
- Constant propagation = Flix where lattice is constant domain

All require MONOTONE transfer functions for soundness. See Section 7.7 for lattice-extended Datalog details.

7. IFDS: Interprocedural Finite Distributive Subset

Paper: Reps, Horwitz, Sagiv 1995

IFDS is the workhorse algorithm for precise interprocedural dataflow analysis **when transfer functions are DISTRIBUTIVE** ($f(a \sqcup b) = f(a) \sqcup f(b)$). Its genius is reducing dataflow problems to graph reachability.

Important: IFDS does NOT apply to pointer analysis (non-distributive). See Section 12.18 for set constraints handling non-distributive cases.

Flix Perspective (Madsen 2016)

IFDS is lattice-extended Datalog where:

- Lattice = powerset of dataflow facts (finite)
- Transfer = gen/kill functions (distributive)
- Distributivity = transfer distributes over join

IDE extends IFDS where:

- Lattice = micro-function space (environment transformers)
- Transfer = function composition
- Enables: constant propagation, linear constant propagation

Souffle Perspective (Jordan 2016): IFDS expressed as Datalog can be COMPILED, not just interpreted:

1. Stage 1: IFDS \rightarrow Datalog rules (declarative specification)
2. Stage 2: Datalog \rightarrow RAM (semi-naive evaluation, index selection)
3. Stage 3: RAM \rightarrow Rust/C++ (specialized code via templates)

This achieves 50x+ speedup over interpreted Datalog (bddbdb, muZ). See Section 7.6 for compilation strategy details.

7.1 The Key Insight

The Problem: Interprocedural analysis is hard because:

- Must track calling context (don't mix up callers)
- Must handle recursion
- Naive approach: exponential in call depth

Example (imprecise without context):

```
def f(x):  
    return x + 1  
a = f(1)    # Should be 2  
b = f(10)   # Should be 11
```

Without context sensitivity: **f** receives $\{1, 10\}$, returns $\{2, 11\}$, so both **a** and **b** get $\{2, 11\}$ — IMPRECISE!

With context sensitivity: Call 1 has **f** receive 1, return 2; Call 2 has **f** receive 10, return 11. Thus $a = 2$, $b = 11$ — PRECISE!

The Reps Insight: Frame the problem as GRAPH REACHABILITY with context encoded as MATCHED PARENTHESES:

- $(1 \dots)_1$ means: call site 1, return to site 1
- $(1(2)_2)_1$ means: nested call, properly matched

INVALID paths like $(1)_2$ are automatically rejected. This is CFL-reachability with the Dyck language.

7.2 The Exploded Supergraph

Construction: Given:

- Program with procedures p_1, \dots, p_n
- CFG for each procedure
- Finite set D of dataflow facts ($|D| = d$)

Build the **Exploded Supergraph** $G^\#$:

NODES : $\{\langle n, d \rangle \mid n \text{ is a program point, } d \in D \cup \{0\}\}$

EDGES : For each CFG edge $n_1 \rightarrow n_2$ and its transfer function f :

Add edge $\langle n_1, d_1 \rangle \rightarrow \langle n_2, d_2 \rangle$ iff $d_2 \in f(\{d_1\})$

The 0 fact represents “nothing” — it’s the identity for the dataflow function.

Theorem 7.1 (Reps 1995). *Fact d holds at program point n if and only if there exists a **realizable path** from $\langle s_{main}, 0 \rangle$ to $\langle n, d \rangle$.*

Realizable = matched call/return parentheses.

7.3 The Tabulation Algorithm

The tabulation algorithm is the heart of IFDS. It computes *path edges* that track same-level realizable paths through the exploded supergraph, and *summary edges* that capture the input-output behavior of procedures. The algorithm uses a worklist to propagate facts, handling four cases: intraprocedural edges, call edges (entering callees), exit edges (leaving procedures), and call-to-return edges (bypassing calls). The key insight is that summary edges allow reusing procedure analysis across multiple call sites.

The following F* code defines the core data structures: **path_edge** represents a realizable path from procedure entry with fact **d1** to node **n** with fact **d2**; **summary_edge** captures a procedure’s effect from call site to return site. The **solve** function implements the worklist algorithm, achieving $O(ED^3)$ complexity where E is the number of edges and D is the domain size.

IFDS Tabulation Algorithm (RHS95, Figure 5)

```
module BrrrMachine.IFDS

(* -----
   PROBLEM DEFINITION
   ----- *)

(* An IFDS problem instance *)
type ifds_problem (d : Type) = {
  (* The supergraph *)
  supergraph : cpg;
  (* The dataflow domain --- MUST BE FINITE *)
  domain : finite_set d;
  (* The zero fact *)
  zero : d;
  (* Flow functions for each edge type *)
  flow_function : cpg_edge → d → set d;
  (* Call flow: how facts map at call site *)
```

```

call_flow : node_id → d → set d;
(* Return flow: how facts map at return *)
return_flow : node_id → node_id → d → d → set d;
(* Call-to-return flow: for facts not passed to callee *)
call_to_return_flow : node_id → d → set d;
}

(* -----
  DATA STRUCTURES
  ----- *)

(* Path edge: same-level realizable path *)
type path_edge (d : Type) = {
  (* Procedure containing this path *)
  proc_entry : node_id;
  (* Starting fact at procedure entry *)
  d1 : d;
  (* Current node reached *)
  n : node_id;
  (* Current fact *)
  d2 : d;
}

(* Summary edge: captures procedure's effect *)
type summary_edge (d : Type) = {
  (* Call site *)
  call_site : node_id;
  (* Fact at call *)
  d1 : d;
  (* Return site *)
  return_site : node_id;
  (* Fact at return *)
  d2 : d;
}

(* Solver state *)
type ifds_state (d : Type) = {
  path_edges : set (path_edge d);
  summary_edges : set (summary_edge d);
  worklist : list (path_edge d);
}

(* -----
  THE ALGORITHM
  ----- *)

val solve : #d:Type → ifds_problem d → set (node_id * d)
let solve #d prob =
  (* Initialize with entry points *)
  let entries = get_entry_points prob.supergraph in
  let init_edges = Set.of_list (List.map (fun entry →
    { proc_entry = entry; d1 = prob.zero; n = entry; d2 = prob.zero }
  ) entries) in
  let init_state = {
    path_edges = init_edges;
    summary_edges = Set.empty;
    worklist = Set.to_list init_edges;
  } in

  (* Main loop *)
  let rec process state =
    match state.worklist with
    | [] → state

```

```

| edge :: rest →
  let state' = { state with worklist = rest } in
  let state'' = process_edge prob edge state' in
  process state''
in
let final = process init_state in
(* Extract results *)
Set.map (fun pe → (pe.n, pe.d2)) final.path_edges

val process_edge : #d:Type → ifds_problem d → path_edge d →
  ifds_state d → ifds_state d
let process_edge #d prob edge state =
  let n = edge.n in
  let d2 = edge.d2 in
  match get_node_type prob.supergraph n with

  (* -----
    CASE 1: Call node
    ----- *)
  | NCall →
    let callee = get_callee prob.supergraph n in
    let callee_entry = get_entry_node prob.supergraph callee in
    let return_site = get_return_site prob.supergraph n in

    (* Propagate to callee entry *)
    let callee_facts = prob.call_flow n d2 in
    let callee_edges = Set.map (fun d3 →
      { proc_entry = callee_entry; d1 = d3; n = callee_entry; d2 = d3 }
    ) callee_facts in

    (* Apply existing summaries *)
    let matching_summaries = Set.filter (fun se →
      se.call_site = n && Set.mem se.d1 callee_facts
    ) state.summary_edges in
    let summary_edges = Set.map (fun se →
      { proc_entry = edge.proc_entry; d1 = edge.d1;
        n = return_site; d2 = se.d2 }
    ) matching_summaries in

    (* Call-to-return flow (for facts not passed to callee) *)
    let ctr_facts = prob.call_to_return_flow n d2 in
    let ctr_edges = Set.map (fun d3 →
      { proc_entry = edge.proc_entry; d1 = edge.d1;
        n = return_site; d2 = d3 }
    ) ctr_facts in
    propagate_all (Set.unions [callee_edges; summary_edges; ctr_edges]) state

  (* -----
    CASE 2: Exit node
    ----- *)
  | NExit proc →
    let callers = get_call_sites prob.supergraph proc in
    (* For each caller with matching entry fact *)
    let new_edges = Set.concat_map (fun call_site →
      let return_site = get_return_site prob.supergraph call_site in
      (* Find path edges reaching this call with matching callee entry fact *)
      let matching_paths = Set.filter (fun pe →
        pe.n = call_site &&
        Set.mem edge.d1 (prob.call_flow call_site pe.d2)
      ) state.path_edges in
      Set.concat_map (fun caller_edge →
        let d4 = caller_edge.d2 in
        let d5 = prob.return_flow call_site return_site d4 d2 in

```

```

    (* Create summary edge *)
    let new_summary = {
      call_site = call_site; d1 = edge.d1;
      return_site = return_site; d2 = d2
    } in
    (* Propagate to return site *)
    let return_edges = Set.map (fun d5' →
      { proc_entry = caller_edge.proc_entry; d1 = caller_edge.d1;
        n = return_site; d2 = d5' }
    ) d5 in
    (new_summary, return_edges)
  ) matching_paths
) callers in
let (new_summaries, new_path_edges) = Set.partition_pair new_edges in
let state' = { state with
  summary_edges = Set.union state.summary_edges new_summaries
} in
propagate_all (Set.flatten new_path_edges) state'

(* -----
CASE 3: Ordinary node
----- *)
| _ →
let successors = get_cfg_successors prob.supergraph n in
let new_edges = Set.concat_map (fun succ →
  let d3_set = prob.flow_function
    { source = n; target = succ; label = CfgNext } d2 in
  Set.map (fun d3 →
    { proc_entry = edge.proc_entry; d1 = edge.d1; n = succ; d2 = d3 }
  ) d3_set
) successors in
propagate_all new_edges state

val propagate_all : #d:Type → set (path_edge d) → ifds_state d → ifds_state d
let propagate_all #d edges state =
  let new_edges = Set.diff edges state.path_edges in
  { state with
    path_edges = Set.union state.path_edges new_edges;
    worklist = Set.to_list new_edges @ state.worklist }

```

Complexity Analysis: Let N = number of nodes in supergraph, E = number of edges, D = size of dataflow domain.

Space: Path edges: $\mathcal{O}N \times D^2$ — bounded by (proc_entry, d_1 , n , d_2). Summary edges: $\mathcal{O}\text{Call_sites} \times D^2$. Total: $\mathcal{O}N \times D^2$.

Time: Each path edge processed once: $\mathcal{O}N \times D^2$ iterations. Each iteration: $\mathcal{O}D$ work for flow functions. Total: $\mathcal{O}N \times D^3 = \mathcal{O}E \times D^3$ since $E \geq N$.

Key Insight: Polynomial in D , not exponential! This is because we track (entry_fact, current_fact) pairs, not full paths through the exploded graph.

Bidirectional IFDS Extension (FlowDroid)

Source: Arzt 2014 (FlowDroid). Cross-reference: Section 8.1.6.

For HEAP-SENSITIVE taint analysis, standard forward-only IFDS is insufficient. FlowDroid extends IFDS with bidirectional analysis:

- **Forward:** Propagate taint from sources, spawn backward on heap writes
- **Backward:** Find heap aliases, spawn forward with INACTIVE taint

The two directions SPAWN each other:

- Forward \rightarrow finds $p.f = \text{tainted} \rightarrow$ spawns Backward to find aliases of $p.f$
- Backward \rightarrow finds q aliased to $p \rightarrow$ spawns Forward with $(q.f, \text{INACTIVE})$

Important: This is NOT pure IFDS!

- Forward taint propagation: IFDS (distributive)
- Backward alias finding: NOT IFDS (may-alias is non-distributive)
- Integration: Ad-hoc spawning mechanism

A cleaner formalization would use IDE (Section 7.7) for the backward phase. See Section 8.1.6.1 for full activation statement semantics.

7.4 Common IFDS Problems

IFDS is a general framework that can express many dataflow analyses. Each instantiation requires defining: (1) a finite domain D of dataflow facts, (2) flow functions for different edge types, and (3) the distributive merge operation (set union). The following code demonstrates two classic instantiations:

Reaching Definitions: The domain consists of (variable, definition-site) pairs. Flow functions implement gen/kill: assignments *generate* new definitions and *kill* previous definitions of the same variable. This analysis answers: “Which definitions of variable v can reach program point p ?”

Taint Analysis: The domain tracks tainted variables and fields. Sources introduce taint, sanitizers remove it, and assignments propagate it. This analysis answers: “Can user input reach a security-sensitive sink?” The taint analysis problem is foundational for detecting injection vulnerabilities.

Instantiations of IFDS for Common Analyses

```
(* -----
REACHING DEFINITIONS
----- *)

type reaching_def = { var : string; def_site : node_id }

let reaching_definitions_problem (cpg : cpg) : ifds_problem reaching_def = {
  supergraph = cpg;
  domain = all_definitions cpg;
  zero = { var = ""; def_site = 0 };  (* Dummy zero *)

  flow_function = fun edge d →
    let n = edge.source in
    match get_defined_var cpg n with
    | Some v when d.var = v →
      (* Kill: this definition kills previous defs of same var *)
      Set.empty
    | Some v →
      (* Gen: add this definition, keep others *)
      Set.add { var = v; def_site = n } (Set.singleton d)
    | None →
      Set.singleton d;  (* Pass through *)

  call_flow = fun call_site d →
    (* Pass definitions through call *)
    Set.singleton d;

  return_flow = fun call_site return_site d_call d_exit →
    (* Definitions from callee flow back *)
    Set.singleton d_exit;

  call_to_return_flow = fun call_site d →
    (* Local definitions not affected by call *)
    if is_local_def cpg d.var then Set.singleton d else Set.empty;
}
```

```

(* -----
   Taint Analysis (via IFDS)
   Source: Livshits 2005
   ----- *)

type taint_fact =
| TaintedVar : var:string → taint_fact
| TaintedField : base:string → field:string → taint_fact
| TaintedReturn : taint_fact (* Return value is tainted *)

let taint_analysis_problem
  (cpg : cpg)
  (sources : set node_id)
  (sanitizers : set node_id)
: ifds_problem taint_fact = {
  supergraph = cpg;
  domain = all_taint_facts cpg;
  zero = TaintedVar ""; (* Dummy *)

  flow_function = fun edge d →
    let n = edge.source in
    (* Source: introduce taint *)
    if Set.mem n sources then
      match get_assigned_var cpg n with
      | Some v → Set.add (TaintedVar v) (Set.singleton d)
      | None → Set.singleton d
    (* Sanitizer: remove taint *)
    else if Set.mem n sanitizers then
      match get_assigned_var cpg n with
      | Some v when d = TaintedVar v → Set.empty
      | _ → Set.singleton d
    (* Assignment: propagate taint *)
    else match get_assignment cpg n with
    | Some (lhs, rhs_vars) →
      let rhs_tainted = List.∃ (fun v → d = TaintedVar v) rhs_vars in
      if rhs_tainted then
        Set.add (TaintedVar lhs) (Set.singleton d)
      else if d = TaintedVar lhs then
        Set.empty (* Overwritten with clean value *)
      else
        Set.singleton d
    | None → Set.singleton d;

  call_flow = fun call_site d →
    (* Map tainted actuals to tainted formals *)
    match d with
    | TaintedVar v →
      let param_index = get_arg_index cpg call_site v in
      begin match param_index with
      | Some i → Set.singleton (TaintedVar (get_formal cpg call_site i))
      | None → Set.empty
      end
    | _ → Set.empty;

  return_flow = fun call_site return_site d_call d_exit →
    (* Map tainted return to tainted result variable *)
    match d_exit with
    | TaintedReturn →
      begin match get_result_var cpg return_site with
      | Some v → Set.singleton (TaintedVar v)
      | None → Set.empty
      end
    | _ → Set.empty;

```

```

call_to_return_flow = fun call_site d →
  (* Taint of locals not affected by call *)
  match d with
  | TaintedVar v when not (is_arg cpg call_site v) → Set.singleton d
  | _ → Set.empty;
}

(* LIMITATION: This basic IFDS taint is FLOW-INSENSITIVE for heap.
   For PRECISE heap taint with flow-sensitivity, see Section 8.1.6.1
   (FlowDroid activation statements) which extends IFDS with:
   - Access paths (x.f.g) instead of simple variables
   - Bidirectional analysis (forward taint + backward alias)
   - Activation statements for flow-sensitive heap precision

   For CROSS-LANGUAGE taint, see Section 9.1.4 (PolyCruise) which uses
   LISR for language-agnostic def/use analysis. *)

(* -----
   UNINITIALIZED VARIABLE ANALYSIS
   ----- *)

type uninit_fact =
  | Uninitialized : var:string → uninit_fact

let uninitialized_analysis_problem (cpg : cpg) : ifds_problem uninit_fact = {
  supergraph = cpg;
  domain = Set.map (fun v → Uninitialized v) (all_variables cpg);
  zero = Uninitialized "";

  flow_function = fun edge d →
    let n = edge.source in
    match d with
    | Uninitialized v →
      (* Declaration without init: add uninit fact *)
      if is_decl_without_init cpg n v then
        Set.singleton (Uninitialized v)
      (* Assignment: remove uninit fact *)
      else if assigns_to cpg n v then
        Set.empty
      (* Use of uninit var: REPORT BUG *)
      else if uses_var cpg n v then
        let _ = report_uninit_use cpg n v in
        Set.singleton d (* Keep tracking *)
      else
        Set.singleton d;
    (* ... call/return flows ... *)
  }
}

```

7.5 IFDS Implementation Optimizations

Sources: Reps, Horwitz, Sagiv 1995; Naeem et al. 2010

Cross-References

- IFDS is DISTRIBUTIVE fragment only — see Section 12.18 (Set Constraints) and Appendix D.10.1 for non-distributive alternatives
- Taint analysis is classic IFDS application — see Section 8.1
- IFDS is OVER-approximate; for under-approximate bug finding use Eval algorithm — see Section 9
- For SOURCE-SINK problems (leak detection), consider SVF (Section 5.6) as alternative — SVF uses sparse value-flow graphs instead of exploded supergraph, which can be more

efficient when memory regions $R \ll \text{domain } D$

7.5.1 Representation Relations

Representation Relations (RHS95 Section 4)

```
(* =====
Key insight: Distributive functions can be compactly represented.
===== *)

(* -----
THEOREM: A function  $f : 2^D \rightarrow 2^D$  is distributive iff:
 $f(X \cup Y) = f(X) \cup f(Y)$ 

Such functions can be encoded as a RELATION  $R \subseteq (D \cup \{0\}) \times D$ 
where  $(d_1, d_2) \in R$  means "if  $d_1$  holds, then  $d_2$  is generated"

SPACE COMPLEXITY:  $O(D^2)$  instead of  $O(2^D)$  for arbitrary functions
COMPOSITION:  $R_1 ; R_2 = \{ (a,c) \mid \exists b. (a,b) \in R_1 \text{ and } (b,c) \in R_2 \}$ 
Relation composition = Function composition!
----- *)

type repr_relation (d : Type) = set (option d * d)
(* None represents the "zero" fact - unconditional generation *)

(* Convert a transfer function to its representation relation *)
val repr_of_transfer : #d:Type → all_facts:set d → (set d → set d) →
    repr_relation d
let repr_of_transfer #d all_facts f =
  (* For each input fact (including 0), compute generated facts *)
  let zero_gen = f Set.empty in (* Unconditional generation:  $f(\{0\})$  *)
  let zero_edges = Set.map (fun d2 → (None, d2)) zero_gen in
  (* For each fact d1, compute what d1 generates *)
  let fact_edges = Set.concat_map (fun d1 →
    let out = f (Set.singleton d1) in
    Set.map (fun d2 → (Some d1, d2)) out
  ) all_facts in
  Set.union zero_edges fact_edges

(* Compose two representation relations *)
val compose_repr : #d:Type → repr_relation d → repr_relation d →
    repr_relation d
let compose_repr #d r1 r2 =
  (*  $R_1 ; R_2 = \text{relational composition}$  *)
  Set.concat_map (fun (a, b) →
    let matching = Set.filter (fun (b', c) →
      match (b, b') with
      | (Some x, Some y) → x = y
      | (None, None) → true
      | _ → false
    ) r2 in
    Set.map (fun (_, c) → (a, c)) matching
  ) r1

(* Apply a representation relation to get output facts *)
val apply_repr : #d:Type → repr_relation d → set d → set d
let apply_repr #d r input =
  (* Output =  $\{ d2 \mid (0, d2) \in R \} \cup \{ d2 \mid \exists d1 \in \text{input}. (d1, d2) \in R \}$  *)
  let from_zero = Set.filter_map (fun (src, tgt) →
    match src with None → Some tgt | Some _ → None
  ) r in
  let from_facts = Set.concat_map (fun d1 →
    Set.filter_map (fun (src, tgt) →
```

```

    match src with Some d when d = d1 → Some tgt | _ → None
  ) r
) input in
Set.union from_zero from_facts

(* -----
THEOREM (Correctness of Representation)
For any distributive f and its representation R:
  apply_repr R input = f input
----- *)

val repr_correct :
#d:Type → all_facts:set d → f:(set d → set d) →
Lemma (requires is_distributive f)
  (ensures ∀ input. apply_repr (repr_of_transfer all_facts f) input
    = f input)

```

7.5.2 H-Sparse Optimization

H-Sparse Optimization (Naeem et al. 2010)

```

(* =====
Observation: Most transfer functions affect only  $h \ll D$  facts.
"Sparse" = few facts generated or killed per edge.
===== *)

(*
COMPLEXITY IMPROVEMENT:
Standard IFDS:  $O(E \cdot D^3)$ 
H-sparse IFDS:  $O(Call \cdot D^3 + h \cdot E \cdot D^2)$ 
where  $h = \max$  facts affected per edge (the "sparsity parameter")

PRACTICAL EXAMPLES:
For taint analysis:  $h \approx 2-3$  (one source, one propagation per stmt)
For null analysis:  $h \approx 1-2$  (one variable nullified per stmt)
For live variables:  $h \approx 2$  (one def, one use per stmt typically)

INTUITION: Most statements touch few variables, so most transfer
functions are nearly identity. Exploit this sparsity!
*)

type sparse_repr (d : Type) = {
  gen : set d; (* Facts generated unconditionally *)
  kill : set d; (* Facts killed (removed from input) *)
  propagate : set (d * d); (* Conditional propagation:  $d1 \rightarrow d2$  *)
  (* INVARIANT:  $|gen| + |kill| + |propagate| \leq h$  *)
}

(* Check if a representation relation is h-sparse *)
val is_h_sparse : #d:Type → repr_relation d → h:nat → bool
let is_h_sparse #d r h =
  Set.size r ≤ h * h (* At most  $h^2$  edges in representation *)

(* Convert full representation to sparse form (if possible) *)
val to_sparse_repr : #d:Type → all_facts:set d → repr_relation d →
  option (sparse_repr d)
let to_sparse_repr #d all_facts r =
  (* gen = facts generated from zero (unconditional) *)
  let gen = Set.filter_map (fun (src, tgt) →
    match src with None → Some tgt | _ → None
  ) r in
  (* identity = facts that map to themselves *)
  let identity_facts = Set.filter (fun d →
    Set.mem (Some d, d) r
  )

```

```

) all_facts in
(* kill = facts in domain but NOT in identity *)
let kill = Set.diff all_facts identity_facts in
(* propagate = non-identity, non-zero mappings *)
let propagate = Set.filter (fun (src, tgt) →
  match src with
  | Some d when d ≠ tgt → true
  | _ → false
) r in
Some { gen; kill;
      propagate = Set.map (fun (Some s, t) → (s, t)) propagate }

(* Sparse propagation - only process affected facts *)
val propagate_sparse : #d:Type → sparse_repr d → set d → set d
let propagate_sparse #d sr input =
  (* 1. Remove killed facts *)
  let after_kill = Set.diff input sr.kill in
  (* 2. Apply conditional propagations *)
  let propagated = Set.concat_map (fun d →
    let targets = Set.filter_map (fun (d', d'') →
      if d' = d then Some d'' else None
    ) sr.propagate in
    if Set.is_empty targets then Set.singleton d else targets
  ) after_kill in
  (* 3. Add generated facts *)
  Set.union sr.gen propagated

(* -----
THEOREM (Sparse Complexity)
For h-sparse problems:
  Time:  $O(\text{Call} \cdot D^3 + h \cdot E \cdot D^2)$ 
  Space:  $O(N \cdot D^2 + \text{Call} \cdot D^2)$ 
When  $h \ll D$ , this is significantly better than  $O(E \cdot D^3)$ 
----- *)

```

7.5.3 Locally Separable Problems (Gen/Kill)

Locally Separable Problems (RHS95 Corollary)

```

(* =====
A problem is "locally separable" if ALL transfer functions are gen/kill:
  f(X) = gen ∪ (X \ kill)
For such problems:  $O(E \cdot D)$  instead of  $O(E \cdot D^3)$  !!!
===== *)

(*
EXAMPLES OF LOCALLY SEPARABLE PROBLEMS:
1. REACHING DEFINITIONS:
  gen = { def } -- this definition reaches
  kill = { prev defs of same var } -- previous defs killed

2. LIVE VARIABLES:
  gen = { used vars } -- used variables are live
  kill = { defined var } -- defined var no longer live before

3. AVAILABLE EXPRESSIONS:
  gen = { computed expr } -- expression now available
  kill = { exprs using modified var } -- invalidated expressions

4. VERY BUSY EXPRESSIONS:
  gen = { expr used on all paths } -- expression busy
  kill = { exprs with modified vars } -- no longer busy

WHAT IS NOT LOCALLY SEPARABLE:

```

```

- Pointer analysis (aliasing creates dependencies)
- Constant propagation (values depend on other values)
- Shape analysis (heap structure depends on operations)
*)

type gen_kill_function (d : Type) = {
  gen : set d;
  kill : set d;
}

(* Check if a transfer function is gen/kill form *)
val is_gen_kill : #d:Type → (set d → set d) → option (gen_kill_function d)
let is_gen_kill #d f =
  (* A function is gen/kill iff:
    1. f({}) gives us the gen set
    2. For all d, either d \in f({d}) (preserved) or d \notin f({d}) (killed) *)
  let gen = f Set.empty in
  (* To find kill: facts d where d \notin f({d}) *)
  (* This requires knowing the domain, simplified here *)
  Some { gen; kill = Set.empty (* computed from domain *) }

(* Check if an IFDS problem is locally separable *)
val is_locally_separable : #d:Type → ifds_problem d → bool
let is_locally_separable #d problem =
  (* All flow functions must be gen/kill *)
  ∀_edges problem.supergraph (fun edge →
    match is_gen_kill (problem.flow_function edge) with
    | Some _ → true
    | None → false
  )

(* -----
   FAST SOLVER FOR LOCALLY SEPARABLE PROBLEMS
   Complexity:  $O(E \cdot D)$  instead of  $O(E \cdot D^3)$ 
   KEY INSIGHT: For gen/kill, we don't need the full IFDS machinery.
   Each fact can be tracked independently!
   ----- *)

val solve_gen_kill : #d:Type →
  problem:ifds_problem d →
  Lemma (requires is_locally_separable problem) →
  (node_id → set d)
let solve_gen_kill #d problem _ =
  (* For gen/kill, facts are independent - track each separately *)
  (* This is essentially running |D| separate bit-vector dataflows *)
  let solve_for_fact (fact : d) : set node_id =
    (* Standard worklist for single-fact reachability *)
    let rec worklist visited frontier =
      match frontier with
      | [] → visited
      | n :: rest →
        if Set.mem n visited then worklist visited rest
        else
          let visited' = Set.add n visited in
          let succs = get_successors problem.supergraph n in
          let live_succs = List.filter (fun succ →
            let gk = get_gen_kill problem n succ in
            not (Set.mem fact gk.kill) (* Not killed on this edge *)
          ) succs in
          worklist visited' (live_succs @ rest)
    in
    (* Also include nodes where fact is generated *)
    let gen_nodes = get_gen_nodes problem fact in

```

```

    worklist Set.empty (Set.to_list gen_nodes)
in
  (* Combine results for all facts *)
  fun node →
    Set.filter (fun fact →
      Set.mem node (solve_for_fact fact)
    ) problem.domain

  (*
    COMPLEXITY ANALYSIS:
    - For each fact:  $O(E)$  traversal
    - Total:  $O(D \cdot E) = O(E \cdot D)$ 
    - Compare to IFDS:  $O(E \cdot D^3)$ 

    PRACTICAL SPEEDUP:
    For  $D = 1000$  facts:  $1000^3 = 10^9$  vs  $10^3 =$  factor of  $10^6$  improvement!
  *)

```

7.5.4 Demand-Driven IFDS

Demand-Driven IFDS

```

  (* =====
    Standard IFDS: Compute ALL reachable facts (exhaustive, whole-program)
    Demand-driven: Only compute facts needed to answer a SPECIFIC QUERY
    ===== *)

  (*
    USE CASE: "Is variable x tainted at line 42?"

    STANDARD IFDS:
    1. Run full taint analysis on entire program
    2. Look up result for (line 42, tainted(x))
    3. Cost:  $O(E \cdot D^3)$  even if we only care about ONE fact at ONE location

    DEMAND-DRIVEN IFDS:
    1. Start from query point (line 42, tainted(x))
    2. Search BACKWARD: "how could this fact become true?"
    3. Stop when we reach a source or prove unreachable
    4. Cost:  $O(\text{relevant subgraph}) \ll O(\text{full program})$ 

    RELATION TO CFL-REACHABILITY (Section 4.2):
    Demand-driven IFDS is backward CFL-reachability in the exploded supergraph.
    The Dyck language ensures matched call/return context.

    DIFFERENCES FROM PURE CFL:
    - IFDS has transfer functions that transform facts
    - CFL just tracks reachability with grammar rules
    - Demand-driven IFDS = CFL + inverse transfer functions
  *)

  (* Query type: does a fact hold at a specific program point? *)
  type ifds_query (d : Type) = {
    query_node : node_id;
    query_fact : d;
  }

  (* Demand-driven solver *)
  val demand_driven_ifds :
    #d:Type →
    problem : ifds_problem d →
    query : ifds_query d →
    bool (* Does fact hold at node? *)

```

```

let demand_driven_ifds #d problem query =
  (* ALGORITHM: Backward search in exploded supergraph from query point *)
  let target = (query.query_node, query.query_fact) in
  let source = (start_main problem.supergraph, Zero) in

  (* Compute inverse transfer functions for backward search *)
  let inverse_flow edge d_out =
    (* Find d_in such that d_out \in flow(edge)(d_in) *)
    Set.filter (fun d_in →
      Set.mem d_out (problem.flow_function edge (Set.singleton d_in))
    ) (Set.add Zero problem.domain)
  in

  (* Backward reachability with context sensitivity *)
  let rec backward_search visited frontier =
    match frontier with
    | [] → false (* Query fact is not reachable from any source *)
    | (node, fact, call_stack) :: rest →
      (* Check if we reached a source (entry of main with Zero fact) *)
      if node = fst source && fact = Zero then
        true (* Found a valid path! *)
      else if Set.mem (node, fact, call_stack) visited then
        backward_search visited rest
      else
        let visited' = Set.add (node, fact, call_stack) visited in
        (* Get predecessors and inverse-flow facts *)
        let preds = get_predecessors problem.supergraph node in
        let new_frontier = List.concat_map (fun pred →
          let edge = { source = pred; target = node } in
          let d_ins = inverse_flow edge fact in
          (* Handle call/return context *)
          match node_type problem.supergraph pred with
          | NCall call_site →
            (* Push call site onto stack *)
            List.map (fun d → (pred, d, call_site :: call_stack))
              (Set.to_list d_ins)
          | NReturn ret_site →
            (* Must match top of stack *)
            (match call_stack with
             | call :: rest_stack when matches_return call ret_site →
              List.map (fun d → (pred, d, rest_stack))
                (Set.to_list d_ins)
             | _ → []) (* Invalid context - prune this path *)
          | _ →
            List.map (fun d → (pred, d, call_stack)) (Set.to_list d_ins)
        ) preds in
        backward_search visited' (new_frontier @ rest)
  in
  backward_search Set.empty [(query.query_node, query.query_fact, [])]

(* -----
  WHEN TO USE DEMAND-DRIVEN VS EXHAUSTIVE

  USE EXHAUSTIVE (standard IFDS) WHEN:
  - Need results for ALL program points
  - Building a whole-program summary
  - Analysis is cheap relative to program size

  USE DEMAND-DRIVEN WHEN:
  - Only need specific query answers
  - Interactive tools (IDE integration)
  - Incremental analysis after edits
  - Query is localized (few relevant paths)

```

HYBRID APPROACH:

1. Run fast over-approximate analysis (exhaustive)
 2. For reported issues, use demand-driven to verify
 3. See Section 4.3.3 for IFDS + Eval combination
- *)

7.5.5 Summary: IFDS Complexity Variants

| Variant | Time Complexity | When to Use |
|-------------------|--|-------------------------------|
| Standard IFDS | $\mathcal{O}E \cdot D^3$ | General distributive problems |
| H-sparse IFDS | $\mathcal{O}\text{Call} \cdot D^3 + h \cdot E \cdot D^2$ | Sparse facts (taint, null) |
| Locally separable | $\mathcal{O}E \cdot D$ | Gen/kill problems |
| Demand-driven | $\mathcal{O}\text{relevant paths}$ | Specific queries, IDE tools |

Where: E = edges in supergraph, D = size of dataflow domain, h = sparsity parameter (max facts affected per edge), Call = number of call sites.

Reminder: IFDS requires DISTRIBUTIVE transfer functions! For non-distributive (pointer analysis): Use set constraints (Section 12.18). For under-approximate bug finding: Use Eval algorithm (Section 9).

7.6 Datalog Compilation Strategy (Souffle)

Paper: Jordan, Scholz, Subotic 2016 (CAV)

For high-performance deployment, IFDS problems expressed as Datalog can be COMPILED to specialized code rather than interpreted.

Key Insight: Compile, Don't Interpret

Jordan 2016 shows: Datalog-based analysis is elegant but SLOW when interpreted. Compilation achieves 50x+ speedup over bddbldb/muZ.

Benchmark (OpenJDK7: 1.4M variables, 350K objects, 160K methods):

- Souffle (compiled): 35 seconds (CI points-to)
- bddbldb: 30 minutes
- SQLite: 6 hours 20 minutes
- muZ: Did not finish

7.6.1 Staged Compilation via Futamura Projections

Souffle Compilation Pipeline:

Stage 1: Datalog \rightarrow RAM (Relational Algebra Machine)

$$P_{\text{RAM}} = \text{Mix}(\text{Int}_{\text{dl}}, I_{\text{db}})$$

Where: Int_{dl} = Semi-naive evaluation interpreter, I_{db} = Intensional database (analysis rules). Result: Relational Algebra Machine program. Optimizations: Rule reordering, semi-naive transformation.

Stage 2: RAM \rightarrow Optimized RAM. Index selection via DILWORTH'S THEOREM (see Section 7.6.2). Join reordering for cache efficiency. Stratum computation for stratified negation.

Stage 3: RAM \rightarrow C++/Rust

$$P_{\text{C++}} = \text{Mix}(\text{Int}_{\text{RAM}}, P_{\text{RAM}})$$

Via template metaprogramming: inlined comparison functions, specialized B-tree iterators, OpenMP parallel annotations.

Theorem 7.2 (Semantic Preservation).

$$\text{result} = \llbracket \text{Int} \rrbracket(\text{Source}, \text{Input}) = \llbracket \text{Source} \rrbracket_{\text{Int}}(\text{Input}) = \llbracket \text{Mix}(\text{Int}, \text{Source}) \rrbracket(\text{Input}) = \llbracket \text{Compiled} \rrbracket(\text{Input})$$

7.6.2 Optimal Index Selection (Dilworth's Theorem)

Problem: Given N index requirements from queries, find MINIMAL set of indices such that each requirement is subsumed by at least one selected index.

Insight: Index subsumption forms a partial order. A lexicographic index $[a, b, c]$ subsumes $[a, b]$ and $[a]$. That is: if I can look up by (a, b, c) , I can also look up by (a) or (a, b) .

Theorem 7.3 (Dilworth's Theorem). *The width of a partial order equals the minimum number of chains needed to cover all elements.*

Algorithm (polynomial time):

1. Build lattice of required indices under subsumption
2. Compute minimum chain partition (via bipartite matching)
3. Select maximum element from each chain

Result: For analyses with 100s of index requirements, reduces to minimal set. Each chain's maximum subsumes all requirements in that chain.

Example:

- Required indices: $[a], [a, b], [a, c], [a, b, c], [b], [b, c]$
- Chain 1: $[a] < [a, b] < [a, b, c] \rightarrow$ select $[a, b, c]$
- Chain 2: $[a, c] \rightarrow$ select $[a, c]$
- Chain 3: $[b] < [b, c] \rightarrow$ select $[b, c]$
- Minimal cover: $\{[a, b, c], [a, c], [b, c]\}$ instead of all 6 indices

Datalog Compilation Types (Jordan et al. 2016)

```
module BrrrMachine.DatalogCompile

(* Relational Algebra Machine operations *)
type ram_expr =
| RAMScan : relation:string → ram_expr
| RAMFilter : expr:ram_expr → column:nat → value:string → ram_expr
| RAMProject : expr:ram_expr → columns:list nat → ram_expr
| RAMJoin : left:ram_expr → right:ram_expr →
    left_col:nat → right_col:nat → ram_expr
| RAMUnion : left:ram_expr → right:ram_expr → ram_expr
| RAMDiff : left:ram_expr → right:ram_expr → ram_expr

type ram_stmt =
| RAMInsert : target:string → source:ram_expr → ram_stmt
| RAMLoop : δ:string → body:list ram_stmt → ram_stmt (* Semi-naive *)
| RAMSeq : stmts:list ram_stmt → ram_stmt

(* Index specification *)
type index_spec = {
  relation : string;
  key_columns : list nat; (* Lexicographic order *)
}

(* Index subsumption: [a,b,c] subsumes [a,b] and [a] *)
let subsumes (super sub : index_spec) : bool =
  super.relation = sub.relation &&
  List.length sub.key_columns ≤ List.length super.key_columns &&
  List.for_all2 (=) sub.key_columns
    (List.take (List.length sub.key_columns) super.key_columns)

(* Dilworth-based minimal index selection *)
val compute_minimal_indices :
  required:list index_spec →
  minimal:list index_spec
  (* Every required index is subsumed by some minimal index *)
```

```

    ∀ r. List.mem r required ==>
      ∃ m. List.mem m minimal ∧ subsumes m r
  }

```

7.6.3 Implementation Strategy for brrr-machine

Recommended Approach

Development Mode: Use interpreted Datalog for rapid iteration

- Crepe (Rust embedded Datalog) for prototyping
- Souffle interpreter mode for debugging
- Modify rules without recompilation

Production Mode: Compile analysis rules to specialized Rust

- Express IFDS/IDE as Datalog rules
- Compile via Souffle → C++ or custom Datalog→Rust pipeline
- Link compiled analysis into brrr-machine

Why not just use Souffle directly?

- Souffle generates C++, we want Rust integration
- We need lattice predicates (Section 7.7) which Souffle lacks
- But Souffle's TECHNIQUES (RAM, Dilworth indices) are essential

7.7 Lattice-Extended Datalog and IDE (Flix)

Paper: Madsen, Yee, Lhotak 2016 (PLDI)

Standard Datalog operates on relations (finite sets). Many analyses require lattices with potentially infinite domains. Flix extends Datalog with lattice predicates, enabling elegant expression of IFDS, IDE, and value analyses.

The Flix Insight: Relations vs Lattices

Standard Datalog (rel):

```

rel Edge(x, y)
Multiple tuples = set
{Edge(1,2), Edge(1,3)} -> keeps both

```

Flix Extension (lat):

```

lat LocalVar(x: Var, v: Constant)
Same key = JOIN lattice values
{LocalVar(x, Cst(1)), LocalVar(x, Cst(2))} -> {LocalVar(x, Top)}

```

What this enables:

- Constant propagation (lattice = constants with Top)
- Interval analysis (lattice = intervals)
- IDE algorithm (lattice = micro-function space)
- Strong update analysis (hybrid flow-sensitive/insensitive)

7.7.1 Lattice Predicates and Transfer Functions

```

// Standard Datalog relation - set semantics
rel AddExp(r: Var, x: Var, y: Var)    // r = x + y in source code

// Flix lattice predicate - join semantics
lat LocalVar(x: Var, v: Constant)    // x has abstract value v
//      ~~~~~

```

```

//          key      lattice element (joined on same key)

// User-defined lattice
enum Constant {
  case Top,          // Unknown (join of all)
  case Cst(Int),     // Known constant
  case Bot           // Unreachable (bottom)
}

// Transfer function: MUST BE MONOTONE
def sum(e1: Constant, e2: Constant): Constant =
  match (e1, e2) with {
    case (Bot, _) | (_, Bot) => Bot      // Unreachable stays unreachable
    case (Cst(n1), Cst(n2)) => Cst(n1 + n2)
    case _                  => Top      // Unknown
  }

// Filter function: guards rule application
def isMaybeZero(e: Constant): Bool =
  match e with {
    case Bot      => false    // Unreachable - no error
    case Cst(n)   => n == 0
    case Top      => true     // Could be zero
  }

// Flix rule with transfer function
LocalVar(r, sum(x, y)) :- AddExp(r, v1, v2),
                          LocalVar(v1, x),
                          LocalVar(v2, y).

// Flix rule with filter function
ArithmeticError(r) :- isMaybeZero(y),
                     DivExp(r, n, d),
                     LocalVar(d, y).

```

7.7.2 IDE as Flix with Micro-Function Lattice

IDE Is Not Complex—It's Just Flix with a Specific Lattice

IFDS path edge: $\text{PathEdge}(d1, n, d2)$ — fact $d2$ holds at n

IDE path edge: $\text{PathEdge}(d1, n, d2, f)$ — with environment transformer f

The micro-function space forms a lattice under pointwise ordering:

$$\perp = \lambda x. \perp \quad (\text{constant bottom function})$$

$$\top = \lambda x. \top \quad (\text{constant top function})$$

$$f \sqcup g = \lambda x. (f\ x) \sqcup (g\ x) \quad (\text{pointwise join})$$

IDE key insight: transformers COMPOSE along paths:

```

PathEdge(d1, m, d3, compose(f, g)) :-
  PathEdge(d1, n, d2, f),
  CFG(n, m),
  EdgeFunction(n, d2, m, d3, g).

```

```

module BrrrMachine.IDE

(* Micro-function: environment transformer *)
type micro_fn (v : Type) = v → v

(* Micro-functions form a lattice under pointwise ordering *)
instance micro_fn_lattice (v : Type) { | complete_lattice v | } :
  complete_lattice (micro_fn v) = {
    bot = (fun _ → bot);
    top = (fun _ → top);
    join = (fun f g → fun x → join (f x) (g x));
    meet = (fun f g → fun x → meet (f x) (g x));
    leq = (fun f g → ∀ x. leq (f x) (g x));
  }

(* IDE path edge: track micro-function along path *)
type ide_path_edge (d : Type) (v : Type) = {
  entry_fact : d;
  current_node : node_id;
  current_fact : d;
  transformer : micro_fn v;
}

(* Composition is the IDE transfer function *)
let ide_compose (#v:Type) (f g : micro_fn v) : micro_fn v =
  fun x → f (g x)

(* Composition is monotone in both arguments *)
val ide_compose_monotone : #v:Type → { | complete_lattice v | } →
  f1:micro_fn v → f2:micro_fn v → g1:micro_fn v → g2:micro_fn v →
  Lemma (requires micro_fn_leq f1 f2 ∧ micro_fn_leq g1 g2)
    (ensures micro_fn_leq (ide_compose f1 g1) (ide_compose f2 g2))

(* -----
WHEN TO USE IDE vs IFDS

USE IFDS (Section 4.1):
- Taint analysis (binary: tainted/untainted)
- Uninitialized variables (set of "maybe uninitialized")
- Typestate (finite state machine)
- Any analysis where facts are BINARY or small finite set

USE IDE:
- Constant propagation (need to track actual constant values)
- Linear constant propagation (x = a*y + b transformers)
- Copy constant analysis (track "which variable copied from")
- Any analysis where you need VALUE TRANSFORMATIONS along paths
----- *)

```

7.7.3 Semi-Naive Evaluation for Lattices

Standard Semi-Naive (for relations):

$$\delta_R = \text{new TUPLES added to } R$$

Flix Semi-Naive (for lattices):

$$\delta_L = (\text{key}, \text{old_value}, \text{new_value}) \text{ where } \text{new_value} \sqsupset \text{old_value}$$

The iteration terminates when no lattice element strictly increases.

Theorem 7.4 (Madsen 2016, Theorem 1). *Every Flix program has a unique minimal model, computable via iteration, PROVIDED all transfer functions are monotone.*

Monotonicity Requirement: For soundness, all transfer functions f must satisfy:

$$x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$$

This is the Flix equivalent of IFDS distributivity requirement.

```
Semi-Naive Evaluation for Lattice Predicates
(* Transfer function monotonicity - critical for soundness *)
type monotone_transfer (#l:Type) {l complete_lattice 1 |} =
  f:(1 → 1){ ∀ x y. leq x y ==> leq (f x) (f y) }

(* Binary monotone transfer (e.g., addition in constant lattice) *)
type monotone_transfer2 (#l:Type) {l complete_lattice 1 |} =
  f:(1 → 1 → 1){
    ∀ x1 x2 y1 y2.
      leq x1 x2 ∧ leq y1 y2 ==> leq (f x1 y1) (f x2 y2)
  }

(* Semi-naive iteration for lattice predicates *)
type lat_δ (k l : Type) = {
  changed_keys : set k;
  old_values : k → l;
  new_values : k → l;
  (* Invariant: ∀ k in changed_keys. old_values k \sqsubset new_values k *)
}

val flix_semi_naive :
  #k:Type → #l:Type → {l complete_lattice 1 |} →
  rules:list flix_rule →
  initial:lat_predicate k l →
  lat_predicate k l (* Returns minimal model *)
```

8. CFL-Reachability

Paper: Reps 1997

CFL-reachability generalizes IFDS by allowing arbitrary context-free grammars to describe valid paths, not just matched parentheses.

8.1 The Framework

Definition 8.1 (CFL-Reachability). Given:

- Graph $G = (V, E)$ with edge labels from alphabet Σ
- Context-free grammar \mathcal{G} with terminals Σ
- Start symbol S

Question: For which pairs (u, v) is there a path from u to v whose edge labels form a string in $L(\mathcal{G})$?

The Dyck Language (matched parentheses):

$$S \rightarrow SS \mid ({}_i S)_i \mid \varepsilon \quad \text{for each call site } i$$

This is exactly what IFDS uses for context sensitivity.

Other Useful Grammars:

Field-sensitive alias analysis:

$$\begin{aligned} \text{FlowsTo} &\rightarrow \text{new} \\ &\mid \text{FlowsTo assign} \\ &\mid \text{FlowsTo load FlowsTo store} \end{aligned}$$

“ x flows to y if there’s a path: new, then assigns, then matched load/store on same field”

Typestate analysis:

$$\text{Valid} \rightarrow \text{open Valid (read | write)* close} \\ | \text{Valid Valid}$$

“Valid use is: open, then reads/writes, then close”

8.2 Demand-Driven Analysis via CFL

Paper: Sridharan 2005

Demand-driven analysis answers specific queries without computing the full solution. For points-to analysis, instead of computing $pts(v)$ for all variables (expensive), we compute on-demand for a single variable at a specific program point. This is implemented as *backward* CFL-reachability: starting from the query (p, x) , we search backward through assignments and field accesses until we reach allocation sites.

The CFL grammar for demand-driven points-to is: $\text{FlowsTo} \rightarrow \text{new} \mid \text{assign FlowsTo} \mid \text{load}_f \text{ PointsTo store}_f \text{ FlowsTo}$. The matched parentheses ensure field-sensitivity (only matching load/store on the same field). This approach achieves significant speedups when few queries are needed, as is common in IDE tools and incremental analysis.

Demand-Driven Points-To Analysis (Sridharan et al. 2005)

```
(*
  KEY INSIGHT:
  Instead of computing points-to for ALL variables (expensive),
  compute on-demand for specific queries.

  Query: "What does variable x point to at program point p?"

  Algorithm:
  1. Start backward search from (p, x)
  2. Follow assign edges backward: x = y means search for y
  3. Follow load/store with matched fields: x = y.f, z.f = w
  4. Stop at allocation sites: new T

  This is CFL-reachability with grammar:
  FlowsTo → new
           | assign FlowsTo
           | load_f PointsTo store_f FlowsTo
*)

type pts_edge_label =
| New : alloc_site:node_id → pts_edge_label
| Assign : pts_edge_label
| Load : field:string → pts_edge_label
| Store : field:string → pts_edge_label

(* Demand-driven points-to query *)
val points_to_query : cpg → node_id → string → set node_id
let points_to_query cpg point var =
  (* Build PEG (Pointer Expression Graph) *)
  let peg = build_peg cpg in
  (* Backward CFL-reachability from (point, var) *)
  let rec search visited (node, var) =
    if Set.mem (node, var) visited then Set.empty
    else
      let visited' = Set.add (node, var) visited in
      (* Check incoming edges *)
      let edges = get_incoming_peg_edges peg (node, var) in
      Set.concat_map (fun edge →
        match edge.label with
        | New alloc_site →
```

```

    Set.singleton alloc_site  (* Found an allocation! *)
  | Assign →
    search visited' edge.source
  | Load field →
    (* Need to find matching store *)
    let base = get_base_var edge in
    let base_pts = search visited' (edge.source_node, base) in
    Set.concat_map (fun alloc →
      (* Find stores to this field on this allocation *)
      let stores = find_stores peg alloc field in
      Set.concat_map (fun store_edge →
        search visited' store_edge.source
      ) stores
    ) base_pts
  | Store _ →
    Set.empty  (* Stores are targets, not sources *)
) edges
in
search Set.empty (point, var)

```

8.3 Interleaved Dyck and the Combined Context+Field Problem

Sources: Reps 2000, Conrado et al. 2025

CFL-reachability (Section 8.1) uses a **single Dyck language** for matched call/return. Field-sensitive alias analysis uses another Dyck language for matched load/store. **Combining both** requires **interleaved Dyck reachability**, which is **undecidable**.

The Interleaved Dyck Problem

Context sensitivity: D_α = matched parentheses ()

Field sensitivity: D_β = matched brackets []

Interleaved Language $I_{\alpha,\beta}$: A string s is in $I_{\alpha,\beta}$ iff:

- s restricted to () is in D_α (valid call/return)
- s restricted to [] is in D_β (valid load/store)

Example valid string: $(_1 [_f (_2] _f) _2 [_g] _g) _1$

Projections: $(_1 (_2) _2) _1 = \text{valid } D_\alpha$; $[_f] _f [_g] _g = \text{valid } D_\beta$

Theorem 8.2 (Reps 2000 — Undecidability). *Interleaved Dyck reachability is UNDECIDABLE. Reduction from Post Correspondence Problem.*

Practical Consequence: Cannot have BOTH context-sensitivity AND field-sensitivity with: (1) Soundness (no false negatives), (2) Completeness (no false positives), (3) Decidability (termination). Must sacrifice ONE property.

Two principled solutions exist: **MCFL underapproximation** (Section 8.4) sacrifices completeness, while **SPDS overapproximation** (Section 8.5) sacrifices soundness of the precision claim.

8.4 Multiple Context-Free Language Reachability (MCFL)

Paper: Conrado, Kjelstrom, Pavlogiannis, van de Pol 2025

MCFL provides a **decidable underapproximation** of interleaved Dyck via a hierarchy of increasingly expressive languages.

MCFL Hierarchy

- 1-MCFL = CFL (context-free languages, $\mathcal{O}n^3$ reachability)
- 2-MCFL strictly contains 1-MCFL ($\mathcal{O}n^4$ reachability)
- d -MCFL strictly contains $(d - 1)$ -MCFL ($\mathcal{O}n^{2d}$ reachability)

Completeness in Limit: For any string $s \in I_{\alpha,\beta}$, there exists d such that s is in d -MCFL. Union over all d gives exactly $I_{\alpha,\beta}$.

Practical Finding (Conrado 2025): 2-MCFL matches overapproximation in 8/11 benchmarks. 2-MCFL confirms 94.3% of paths on remaining benchmarks. $d = 2$ suffices for most real taint analysis problems.

The following F* code formalizes MCFGs. The key insight is that MCFG nonterminals have *arity*: they derive *tuples* of strings rather than single strings. A d -MCFG(r) grammar has dimension d (maximum arity) and rank r (maximum nonterminals per rule). The type signatures enforce well-formedness invariants: `wf_arity` ensures each production produces the correct tuple size, `wf_rank` bounds rule complexity, and `wf_dimension` bounds nonterminal arity.

Multiple Context-Free Grammar (MCFG) Definition

```
module BrrrMachine.MCFG

(* A nonterminal with arity k derives k-tuples of strings *)
type mcfg_nonterminal = {
  name: string;
  arity: pos (* Number of string components it derives, ≥ 1 *)
}

(* Right-hand side element: terminal or variable reference *)
type mcfg_rhs_elem =
| MCFGTerminal : char → mcfg_rhs_elem
| MCFGVariable : nt_idx:nat → component:nat → mcfg_rhs_elem

(* Production rule in d-MCFG(r) *)
type mcfg_rule (d r: pos) = {
  lhs: mcfg_nonterminal;
  lhs_patterns: list (list mcfg_rhs_elem); (* arity-many patterns *)
  rhs_nts: list mcfg_nonterminal;          (* up to r nonterminals *)

  (* Well-formedness invariants *)
  wf_arity: List.length lhs_patterns = lhs.arity;
  wf_rank: List.length rhs_nts ≤ r;
  wf_dimension: lhs.arity ≤ d
}

(* d-MCFG(r): dimension d, rank r *)
type mcfg (d r: pos) = {
  nonterminals: list mcfg_nonterminal;
  terminals: set char;
  rules: list (mcfg_rule d r);
  start: mcfg_nonterminal;
  start_arity_1: start.arity = 1
}

(* The Gd+ grammar for approximating interleaved Dyck *)
val construct_interleaved_grammar : d:pos → k:pos → mcfg d 2

(* Soundness: derived strings are in interleaved Dyck *)
val grammar_sound : d:pos → k:pos → s:string →
  Lemma (requires s 'in_language' (construct_interleaved_grammar d k))
    (ensures is_interleaved_dyck s)

(* Completeness in limit: every interleaved Dyck string is captured *)
val grammar_complete_limit : k:pos → s:string →
  Lemma (requires is_interleaved_dyck s)
    (ensures ∃ d. s 'in_language' (construct_interleaved_grammar d k))
```

Complexity Bounds (Conrado 2025):

| Language Class | Reachability Complexity | Lower Bound |
|------------------|-------------------------|---------------|
| CFL / 1-MCFL | $\mathcal{O}n^3$ | BMM (n^3) |
| d -MCFL(1) | $\mathcal{O}n^{2d}$ | SETH |
| d -MCFL(r) | $\mathcal{O}n^{d(r+1)}$ | SETH |
| Interleaved | Undecidable | PCP |

The SETH lower bound shows $\mathcal{O}n^{2d}$ is essentially optimal for d -MCFL(1).

8.5 Synchronized Pushdown Systems (SPDS)

Paper: Spath, Ali, Bodden 2019

SPDS provides an **automaton-based approach** to combined context+field sensitivity via synchronized pushdown systems, achieving polynomial complexity under a **precision hypothesis**.

Synchronized Pushdown Systems (SPDS)

Idea: Use TWO pushdown automata that operate synchronously:

- P_F : Field PDS (stack encodes field access sequence)
- P_S : Stack PDS (stack encodes calling context)
- Sync: Synchronization predicate on transitions

Encoding:

- Field store $x.f = y$: push rule $(l, \varepsilon) \rightarrow (l', f)$
- Field load $y = x.f$: pop rule $(l, f) \rightarrow (l', \varepsilon)$
- Method call: push rule $(l, \varepsilon) \rightarrow (l', c)$
- Method return: pop rule $(l, c) \rightarrow (l', \varepsilon)$

Key Innovation: Field automaton A_F represents set of valid access paths IMPLICITLY. Avoids exponential enumeration of access paths. $\mathcal{O}N$ automaton states vs $\mathcal{O}2^N$ explicit access paths.

Precision Hypothesis (Spath 2019): An improperly matched call site does not induce a properly matched field access, and vice versa. Under this hypothesis, SPDS = infinite access path precision.

Synchronized Pushdown Systems (Spath et al. 2019)

```

module BrrrMachine.SPDS

(* Stack symbols *)
type field_symbol = | FieldSym : string → field_symbol | FEps
type call_symbol = | CallSiteSym : nat → call_symbol | CEps

(* Pushdown configuration *)
type field_config = { f_state: nat; f_stack: list field_symbol }
type call_config = { c_state: nat; c_stack: list call_symbol }
type spds_config = field_config * call_config

(* Pushdown rules *)
type field_rule =
  | FPush : nat → field_symbol → nat → field_symbol → field_rule
  | FPop : nat → field_symbol → nat → field_rule

type call_rule =
  | CPush : nat → call_symbol → nat → call_symbol → call_rule
  | CPop : nat → call_symbol → nat → call_rule

(* Synchronization predicate *)
type sync_spec = field_rule → call_rule → bool

(* The full SPDS *)

```

```

type spds = {
  field_rules: set field_rule;
  call_rules: set call_rule;
  sync: sync_spec;
  initial: spds_config
}

(* Field automaton: implicitly represents set of valid access paths *)
type field_automaton = {
  fa_states: set nat;
  fa_initial: nat;
  fa_final: set nat;
  fa_transitions: map (nat * field_symbol) (set nat)
}

(* Post* computation via saturation *)
val post_star : set field_rule → field_config → field_automaton

(* SPDS synchronized reachability *)
val spds_post_star : spds → set spds_config
let spds_post_star sys =
  let fa = post_star sys.field_rules (fst sys.initial) in
  let ca = post_star sys.call_rules (snd sys.initial) in
  (* Intersect and filter by synchronization *)
  filter_by_sync sys.sync (cartesian fa ca)

(* Complexity:  $O(|P_F|^3 * |P_S|^3 * |Q|^3)$  *)
val spds_complexity :
  sys:spds →
  Lemma (time_complexity (spds_post_star sys) ≤
    pow3 (size sys.field_rules) * pow3 (size sys.call_rules))

```

Performance Results (Spath 2019, DaCapo Benchmark):

| Analysis | SPDS Speedup | Timeouts (Before/After) |
|-------------------|--------------|-------------------------|
| IO Property | 64x | 160 → 28 |
| Iterator Property | 83x | 137 → 3 |
| Vector Property | 1.8x | 57 → 25 |

SPDS matches access-graph precision on all benchmarks with no false positives.

Tension Resolution: CFL vs MCFL vs SPDS

MCFL (Conrado 2025):

- Grammar-based formalism
- Underapproximation (sound)
- First decidable interleaved Dyck
- Tight complexity bounds (SETH)
- Optimal for verification

SPDS (Spath 2019):

- Automaton-based formalism
- Overapproximation under hypothesis
- Practical WPDS encoding
- Proven speedups (64–83x)
- Optimal for bug finding

Combined Use:

- Run MCFL underapprox: paths found are DEFINITELY reachable
- Run SPDS overapprox: paths NOT found are DEFINITELY unreachable

- Intersection: sound AND complete when they agree
- Conrado 2025 shows agreement in 8/11 benchmarks

Recommendation:

- For taint verification (need soundness): Use MCFL
- For bug finding (need precision): Use SPDS
- For maximum precision: Combine both

8.6 Cross-Reference: Sparse Value-Flow Analysis (SVF)

For **source-sink reachability** problems (memory leaks, use-after-free, taint-to-sink), an alternative to IFDS-based dataflow is **Sparse Value-Flow Analysis** (Section 5.6).

SVF vs IFDS for Source-Sink Problems

Both use CFL-reachability for context-sensitivity (Section 8.1). The difference is in REPRESENTATION and PREREQUISITES:

IFDS (Section 7):

- Exploded supergraph: (program_point, dataflow_fact) pairs
- Self-contained: no pre-analysis required
- Best for: general dataflow, small finite domains

SVF (Section 5.6):

- Sparse VFG: nodes are variable definitions, edges are def-use chains
- REQUIRES pointer analysis for Memory SSA construction
- Best for: source-sink problems, address-taken variable tracking

For leak detection in C/C++, SVF typically achieves 10–50x speedup over dense IFDS approaches (ISSTA 2012) due to sparse traversal.

See Section 5.6 for complete SVF formalization including Memory SSA, SVFG construction rules, leak detection algorithms, and comparison with IFDS.

9. Under-Approximate Analysis (Bug Finding)

Sources: Le et al. 2022 (ISL), Vanegue 2025 (Pulse-infinity), O’Hearn 2020

Critical: IFDS is Over-Approximate. For Bug Finding, We Need Under-Approx.

Over-Approximation (IFDS, Section 7):

- Sound for ABSENCE: “No taint found” means truly safe
- May have FALSE POSITIVES: Reports bugs that don’t exist

Under-Approximation (This section):

- Sound for PRESENCE: “Bug found” means truly a bug
- May have FALSE NEGATIVES: Misses some bugs

Combined: Use IFDS to find candidates, then under-approx to verify.

9.1 The Eval Algorithm (Pulse/Infer)

Paper: Calcagno et al. 2009, 2011 (Bi-Abduction, Infer)

The Eval algorithm performs forward symbolic execution with bi-abduction to discover both bugs AND missing preconditions.

Eval: Forward Symbolic Execution with Bi-Abduction

```
(* -----
   Unlike IFDS which propagates dataflow facts, Eval propagates
   separation logic assertions through the program.
   ----- *)
```

```

type eval_state = {
  pre : assertion;      (* Discovered precondition (accumulated anti-frame) *)
  post : assertion;     (* Current symbolic state *)
  path : list node_id;  (* Execution path for witness *)
  exit : exit_condition; (* Ok or Er *)
}

val eval_stmt : eval_state → ir_stmt → list eval_state
(* May return multiple states for conditionals/loops *)

let eval_stmt state stmt =
  match stmt with
  | Assign (x, e) →
    (* x := e --- update symbolic state *)
    let new_post = substitute state.post x (eval_expr e state.post) in
    [{ state with post = new_post }]

  | Load (x, ptr, field) →
    (* x := ptr→field --- need ptr /→ {field: v} *)
    let required = points_to ptr field (fresh_var "v") in
    match biabduct state.post required with
    | None →
      (* NULL DEREFERENCE: can't satisfy requirement *)
      [{ state with exit = Er;
        post = state.post 'star' error "null_deref" }]
    | Some { anti_frame = m; frame = f } →
      (* Add anti-frame to precondition, update post *)
      let state' = { state with
        pre = state.pre 'star' m;
        post = f 'star' (x /→ fresh_var "v");
      } in
      [state']

  | Store (ptr, field, v) →
    (* ptr→field := v *)
    let required = points_to ptr field (fresh_var "_") in
    match biabduct state.post required with
    | None →
      [{ state with exit = Er;
        post = state.post 'star' error "null_deref" }]
    | Some { anti_frame = m; frame = f } →
      [{ state with
        pre = state.pre 'star' m;
        post = f 'star' points_to ptr field v;
      }]

  | Free ptr →
    (* free(ptr) --- need ptr /→ _ with UNIQUE capability *)
    let required = points_to ptr "_" (fresh_var "_") in
    match biabduct state.post required with
    | None →
      [{ state with exit = Er; post = error "double_free_or_invalid" }]
    | Some { anti_frame = m; frame = f } →
      (* Remove the freed memory from post *)
      [{ state with pre = state.pre 'star' m; post = f }]

  | If (cond, then_branch, else_branch) →
    (* Fork execution for both branches *)
    let then_state = { state with post = state.post 'star' (cond = true) } in
    let else_state = { state with post = state.post 'star' (cond = false) } in
    (* Prune infeasible paths *)
    let then_results = if sat then_state.post
      then eval_block then_state then_branch

```

```

        else [] in
    let else_results = if sat else_state.post
        then eval_block else_state else_branch
        else [] in
    then_results @ else_results

| While (cond, body) →
    (* UNDER-APPROXIMATE: Bounded unrolling, NOT widening *)
    eval_loop_bounded state cond body 3 (* k=3 unrollings *)

| Call (ret, func, args) →
    (* Use function summary if available *)
    match get_summary func with
    | Some summary →
        apply_summary state summary args ret
    | None →
        (* Inline or skip with havoc *)
        [{ state with post = havoc ret state.post }]

(* -----
    BOUNDED LOOP UNROLLING (UNDER-APPROXIMATE)
    For BUG FINDING: unroll k times, then cut off.
    This is SOUND for finding bugs: any bug found is real.
    May MISS bugs that require more iterations.
    ----- *)

val eval_loop_bounded : eval_state → ir_expr → ir_block → nat →
    list eval_state
let eval_loop_bounded init_state cond body max_unroll =
    let rec unroll fuel state =
        if fuel = 0 then
            (* Cut off: return current state as "exited loop" *)
            [{ state with post = state.post 'star' (cond = false) }]
        else
            (* Check loop condition *)
            let continue_state = { state with
                post = state.post 'star' (cond = true) } in
            let exit_state = { state with
                post = state.post 'star' (cond = false) } in
            let continue_results =
                if sat continue_state.post then
                    let body_results = eval_block continue_state body in
                    List.concat_map (fun s → unroll (fuel - 1) s) body_results
                else []
            in
            let exit_results =
                if sat exit_state.post then [exit_state] else []
            in
            continue_results @ exit_results
    in
    unroll max_unroll init_state

```

9.2 ISL Triple Semantics

Paper: Le et al. 2022 — “Finding Real Bugs in Big Programs with ISL”

Incorrectness Separation Logic (ISL) inverts the meaning of Hoare triples. While Hoare logic’s $\{P\}C\{Q\}$ means “ALL executions from P end in Q ” (over-approximation), ISL’s $[p] C [q]$ means “SOME execution from p ends in q ” (under-approximation). This makes ISL ideal for bug finding: if we prove $[p] C [\text{error}]$, we have demonstrated that an error is *actually reachable*.

The following code shows how to convert Eval’s exploration results into ISL triples. The function `eval_to_isl_triple` runs Eval on a function and collects all states that terminate with errors. The `presumption` field captures what conditions were required (accumulated anti-

frames), and the **result** field describes the error state.

Incorrectness Separation Logic (ISL) Triples

```
(* -----
   ISL Triple: [p] C [q; exit]
   Meaning: If execution starts in state satisfying p,
             and terminates with exit condition,
             then final state satisfies q.

   Key difference from Hoare logic:
   - Hoare: {P} C {Q} --- ALL executions from P end in Q (over-approx)
   - ISL: [p] C [q] --- SOME execution from p ends in q (under-approx)
   ----- *)

(* ISL triple computed by Eval *)
val eval_to_isl_triple : cpg → func_id → isl_triple
let eval_to_isl_triple cpg func =
  let init_state = {
    pre = emp; (* Start with empty precondition *)
    post = emp; (* Start with empty postcondition *)
    path = [];
    exit = Ok;
  } in
  let final_states = eval_func cpg func init_state in
  (* Collect error states *)
  let error_states = List.filter (fun s → s.exit = Er) final_states in
  match error_states with
  | [] →
    (* No bugs found *)
    { presumption = emp; code = func; result = emp; exit_cond = Ok }
  | errors →
    (* Combine error states *)
    let combined_pre = List.fold_left star emp
      (List.map (fun s → s.pre) errors) in
    let combined_post = List.fold_left disj bot
      (List.map (fun s → s.post) errors) in
    { presumption = combined_pre;
      code = func;
      result = combined_post;
      exit_cond = Er }
```

9.3 Latent vs Manifest Errors

Paper: Le, Raad, Villard, Berdine, Dreyer, O’Hearn 2022 “Finding Real Bugs in Big Programs with ISL”

Key Contribution: Compositional Bug Reporting via Manifest/Latent Split

Problem: ISL generates many error specs. Which to report?

Example: `deref(x) { *x = 10; }`

ISL triple: `[x = NULL] deref(x) [er : x = NULL]`

Is this a bug? Only if called with NULL. But we don’t know callers.

Solution: Distinguish MANIFEST from LATENT errors.

- **Manifest:** Bug reachable from ANY calling context
- **Latent:** Bug only reachable under specific preconditions

Policy: Report manifest bugs unconditionally. Use latent specs compositionally to find manifest bugs in callers.

Definition 9.1 (Manifest Error (Le 2022, Section 3.2)). An error triple $\models [p] C [\text{er} : q]$ denotes a MANIFEST error if:

1. The presumption is trivial: $p \equiv \text{emp} \wedge \text{true}$

2. The result is satisfiable: $\text{sat}(q)$
3. All heap locations in q are existentially quantified (fresh)
4. All pure constraints in q are satisfiable under any valuation

Formally, for $q = \exists X_q. \kappa_q \wedge \pi_q$:

1. $p \equiv \text{emp} \wedge \text{true}$ — No precondition requirements
2. $\text{sat}(q)$ holds — Error state is reachable
3. $\text{locs}(\kappa_q) \subseteq X_q$ — Heap is fresh (not from caller)
4. $\forall v. \text{sat}(\pi_q[v/Y \cup \text{locs}(\kappa_q)])$ — Pure part always satisfiable

Intuition: Manifest = error reachable regardless of how function is called. The heap portion is freshly allocated (condition 3), so caller can't prevent error by providing different inputs.

Theorem 9.2 (True Positives Property (Le 2022, Theorem 3.4)). *If procedure $f()$ in a complete program has a MANIFEST error, then either:*

- (a) $f()$ is dead code (not reachable from $\text{main}()$), OR
- (b) There exists a concrete trace from $\text{main}()$ to the error.

Consequence: Manifest errors have 0% false positive rate. If we report it, it's a real bug (unless dead code).

Manifest Error Detection (Algorithmic)

```

type error_classification =
  | Manifest : error_classification      (* Report unconditionally *)
  | Latent : precondition:assertion → error_classification
    (* Report at call site *)
  | LatentLeak : error_classification
    (* Report anyway - leaks are caller's fault rarely *)

val classify_error : isl_triple → error_classification
let classify_error triple =
  match triple.exit_cond with
  | Ok → failwith "Not an error triple"
  | Er →
    let p = triple.presumption in
    let q = triple.result in
    (* Condition 1: Trivial presumption *)
    let trivial_pre = is_emp_and_true p in
    (* Condition 2: Satisfiable result *)
    let sat_result = sat q in
    (* Condition 3: Fresh heap locations *)
    let fresh_heap = all_locs_existentially_quantified q in
    (* Condition 4: Pure constraints satisfiable *)
    let pure_sat = pure_always_satisfiable q in

    if trivial_pre && sat_result && fresh_heap && pure_sat then
      Manifest
    else if is_memory_leak_error triple then
      LatentLeak (* Report leaks even when latent *)
    else
      Latent triple.presumption

(* BUG REPORTING POLICY (Le 2022 Section 2.3) *)
val should_report : func_id → isl_triple → bool
let should_report func triple =
  match classify_error triple with
  | Manifest → true
  | LatentLeak → true (* Always report leaks *)
  | Latent _ →
    (* Report latent NPE only in main() *)
    is_main_function func && is_null_deref_error triple

```

Practical Results (Pulse-X tool):

OpenSSL-1.0.1h (2015, 2.83M bytes IR, 8,658 procedures):

- Pulse-X: 26 bugs reported, 19 fixed (73% fix rate)
- Infer: 80 bugs reported, 39 fixed (49% fix rate)
- Pulse-X has HIGHER fix rate due to manifest filtering

OpenSSL-3.0.0 (2021):

- Pulse-X found 15 NEW bugs, all confirmed and fixed by maintainers
- Including NPEs and memory leaks in core crypto code

Comparison to Infer:

| INFER (Over-Approximate) | PULSE-X (Under-Approximate) |
|-------------------------------|-----------------------------|
| Based on separation logic | Based on ISL |
| Proves ABSENCE of bugs | Proves PRESENCE of bugs |
| Uses heuristics for reporting | Uses manifest criterion |
| May have false positives | 0% FP for manifest bugs |
| Wider coverage (all paths) | Targeted (bounded paths) |
| Needs widening for loops | Simple bounded unrolling |

Algorithmic Simplification: Le 2022 Remark 1: The ISL-based algorithm is “strikingly simple” compared to over-approximate biabduction. Key reason: NO LOOP INVARIANTS NEEDED. Under-approximation uses bounded unrolling—sound for finding bugs that trigger within k iterations.

9.4 Integration: IFDS + Eval Hybrid

Hybrid Analysis Architecture

Phase 1: IFDS (Fast, Over-Approximate)

Run IFDS with complexity $\mathcal{O}ED^3$ to produce candidate bugs (may include false positives), e.g., “taint reaches sink at line 42”.

Phase 2: Eval (Precise, Under-Approximate)

Run Eval on backward slice to produce ISL triples with paths, e.g., `[emp] slice [err; Er]`.

Phase 3: Classification

For each IFDS finding:

1. Compute backward slice from finding
2. Run Eval on slice
3. Check manifest conditions (Le 2022 Definition 3.3)
4. Classify as Manifest/Latent/Refuted

Results:

- **Manifest** → TRUE BUG (0% FP by theorem)
- **Latent** → TRUE BUG with required context
- **Refuted** → FALSE POSITIVE (don’t report)

Hybrid Analysis Implementation

```
type hybrid_result =  
  | Confirmed : classification:bug_classification →  
    witness:list node_id → hybrid_result  
  | Refuted : reason:string → hybrid_result  
  | Timeout : partial:option eval_state → hybrid_result  
  
val verify_ifds_finding_with_eval :  
  cpg:cpg → finding:dataflow_fact → timeout_ms:nat → hybrid_result
```

```

let verify_ifds_finding_with_eval cpg finding timeout =
  (* Step 1: Compute backward slice *)
  let slice = backward_slice cpg finding.sink_node in
  (* Step 2: Run Eval on slice with timeout *)
  match eval_with_timeout slice timeout with
  | Timeout partial → Timeout partial
  | Complete final_states →
    (* Step 3: Check if any state confirms the finding *)
    let confirming = List.filter (confirms_finding finding) final_states in
    match confirming with
    | [] → Refuted "Eval found no path to error"
    | states →
      (* Step 4: Build ISL triple and classify *)
      let triple = states_to_isl_triple states in
      let classification = classify_bug triple in
      Confirmed classification (extract_witness states)

```

10. Symbolic Execution (Path-Sensitive)

Source: King 1976 – “Symbolic Execution and Program Testing”

Symbolic Execution vs Eval vs IFDS

IFDS (Section ??):

- Path-*insensitive* (merges at join points)
- Tracks DATAFLOW FACTS (finite domain)
- $\mathcal{O}(ED^3)$ complexity

Eval (Section 9.1):

- Path-*sensitive* (explores paths separately)
- Tracks SEPARATION LOGIC ASSERTIONS
- Bi-abduction for compositionality

Symbolic Execution (This chapter):

- Path-*sensitive* with **path condition**
- Tracks SYMBOLIC VALUES (expressions over inputs)
- SMT solver for path feasibility

Use Case Guidance:

- Taint analysis → IFDS(fast, whole-program)
- Memory bugs → Eval (separation logic)
- Numeric bugs → Symbolic Execution (precise arithmetic)

Historical Evolution: Symbolic Execution Foundations

1976: King – Pure Symbolic Execution

- Original formulation: execute with symbolic values
- Path conditions track branch decisions
- **Limitation:** Path explosion, constraint complexity

2005: DART (Godefroid) & CUTE (Sen) – Concolic Execution

- **Key insight:** Run concrete + symbolic *simultaneously*
- Concrete guides symbolic; fallback when symbolic fails
- **Result:** Practical symbolic execution for real programs

2008: KLEE (Cadar) – Scalable Symbolic Execution

- LLVM bitcode as symbolic target (language-independent)
- Constraint solver optimizations: 95% query reduction
- **Result:** 90%+ coverage on COREUTILS, 56 bugs found

2018: QSYM (Yun) – Optimistic Concolic for Hybrid Fuzzing

- Native instrumentation ($2\text{--}5\times$ vs $10\text{--}100\times$ IR interpretation)
- Optimistic solving: drop hard constraints, fuzzer validates
- **Result:** 13 new bugs in heavily-fuzzed software

10.1 Execution Tree vs CPG

| CPG(Static) | Execution Tree (Dynamic) |
|-------------------|--------------------------|
| Static structure | Dynamic exploration |
| All edges exist | Forking at branches |
| No path condition | PC at each node |
| Finite | Potentially infinite |

Relationship: Execution tree is *computed from* CPG on-demand. CPG provides structure; execution tree provides path-sensitive state.

10.2 On-Demand Symbolic Execution

Unlike IFDS which merges states at join points, symbolic execution maintains *path-sensitive* state by exploring each execution path separately. The key data structure is a `symbolic_state` containing: (1) a symbolic environment mapping variables to symbolic expressions, (2) a path condition (PC) accumulating branch decisions, and (3) the current program location.

The algorithm uses a worklist of symbolic states. At each branch, it *forks* into two states with extended path conditions. The SMT solver checks path feasibility: infeasible paths (unsatisfiable PC) are pruned. This approach is *exact* for reachability but may not terminate due to loops.

The following code demonstrates on-demand symbolic execution from a CPG. The function `symbolic_execute_from_cpg` explores paths from `entry` to `target`, returning all feasible symbolic states that reach the target. The depth bound prevents infinite exploration of loops.

Symbolic Execution: On-Demand from CPG

```
(* We don't store the execution tree. We compute it lazily when needed
   to verify specific findings or explore specific paths. *)
val symbolic_execute_from_cpg :
  cpg:cpg →
  entry:node_id →
  target:node_id →          (* Stop when we reach target *)
  max_depth:nat →
  list symbolic_state       (* All paths that reach target *)

let symbolic_execute_from_cpg cpg entry target max_depth =
  let init = {
    env = init_symbolic_env cpg entry;
    pc = []; (* Empty path condition *)
    stmt = entry;
    depth = 0;
  } in
  let rec explore worklist results =
    match worklist with
    | [] → results
    | state :: rest →
      if state.stmt = target then
        (* Reached target - add to results if path is feasible *)
        if pc_satisfiable state.pc then
          explore rest (state :: results)
        else
          explore rest results
      else if state.depth ≥ max_depth then
        (* Depth bound - cut off *)
        explore rest results
      else
        explore rest (state :: results)
  in
  explore [init] []
```

```

        explore rest results
    else
        (* Get successors from CPG *)
        let succs = cpg_successors cpg state.stmt in
        let new_states = List.concat_map (step_symbolic state) succs in
        explore (new_states @ rest) results
    in
    explore [init] []

```

SMT Integration for Path Feasibility

```

(* King 1976: "The symbolic execution of IF statements requires theorem
proving which, even for modest programming languages, is mechanically
impossible."

REALITY: SMT solvers (Z3, CVC5) are incomplete but practical.
We accept Unknown as a valid result. *)
type smt_result = Sat of model | Unsat | Unknown of string

val pc_satisfiable : path_condition → bool
let pc_satisfiable pc =
    match smt_check pc with
    | Sat _ → true
    | Unsat → false
    | Unknown _ → true (* Conservatively assume satisfiable *)

val pc_implies : path_condition → symbolic_expr → trilean
let pc_implies pc expr =
    (* Check if pc ==> expr *)
    match smt_check (pc @ [NegAtom expr]) with
    | Unsat → Definitely (* pc ∧ not expr is unsat ==> pc ==> expr *)
    | Sat _ → DefinitelyNot (* Found counterexample *)
    | Unknown _ → Unknown (* SMT timeout or incomplete *)

```

Speculative Symbolic Execution Variant (SPECTECTOR)

```

(* Source: Guarnieri et al. 2020

For detecting speculative execution vulnerabilities (Spectre), we need
a variant that forks on BOTH branch outcomes, modeling misprediction.

KEY DIFFERENCES from standard symbolic execution:
1. At branches, fork on BOTH outcomes (not just feasible paths)
2. Track speculation depth (bounded by hardware window ~200)
3. Collect observation traces (memory accesses, jumps)
4. Check Speculative Non-Interference (SNI) property *)

type symex_mode =
| StandardSymex (* Fork on feasible paths only *)
| SpeculativeSymex (* Fork on BOTH outcomes up to window *)

val symbolic_execute_speculative :
    cpg:cpg →
    entry:node_id →
    spec_window:nat →
    list (symbolic_state * obs_trace)

let symbolic_execute_speculative cpg entry spec_window =
    let init = {
        env = init_symbolic_env cpg entry;
        pc = [];
        stmt = entry;
        depth = 0;
        spec_depth = 0;
        obs_trace = [];
    }

```

```

} in
let rec explore worklist results =
  match worklist with
  | [] → results
  | state :: rest →
    match get_node_kind cpg state.stmt with
    | NBranch cond t_target f_target →
      if state.spec_depth < spec_window then
        (* SPECULATIVE: fork on BOTH outcomes *)
        let obs = ObsJumpTarget state.stmt in
        let state_true = { state with
          stmt = t_target;
          obs_trace = obs :: state.obs_trace
        } in
        let state_false = { state with
          stmt = f_target;
          spec_depth = state.spec_depth + 1;
          obs_trace = obs :: state.obs_trace
        } in
        explore (state_true :: state_false :: rest) results
      else
        (* Speculation window exhausted *)
        explore rest ((state, state.obs_trace) :: results)
    | NLoad addr _ →
      let obs = ObsMemAccess (eval_addr state addr) in
      let state' = { state with obs_trace = obs :: state.obs_trace } in
      explore (advance state' :: rest) results
    | NTerminal →
      explore rest ((state, state.obs_trace) :: results)
    | _ →
      explore (advance state :: rest) results
in
explore [init] []

```

10.3 Witness Generation

A critical capability of symbolic execution is *witness generation*: converting a symbolic execution path into a concrete test case that triggers the bug. When symbolic execution finds a path to an error state, the path condition (PC) encodes exactly which inputs lead there. By asking the SMT solver for a satisfying assignment to the PC, we obtain concrete input values.

The following code defines `concrete_witness` containing the input assignments, execution path, and final variable values. The function `generate_witness` queries the SMT solver for a model satisfying the path condition, then extracts concrete values for all input variables. The function `confirm_bug_with_witness` integrates witness generation with bug classification: manifest bugs should always yield witnesses, while latent bugs require the specified preconditions to be satisfied.

Witness Generation: Convert Symbolic Path to Concrete Test Case

```

type concrete_witness = {
  inputs : map string int;      (* Variable assignments *)
  path : list node_id;         (* Execution path *)
  final_state : map string int; (* Final variable values *)
}

val generate_witness : symbolic_state → option concrete_witness
let generate_witness sym_state =
  match smt_check sym_state.pc with
  | Sat model →
    (* Extract concrete values from model *)
    let inputs = extract_input_values model sym_state.env in
    let final = instantiate_env sym_state.env model in

```

```

    Some { inputs = inputs; path = sym_state.path; final_state = final }
  | Unsat → None
  | Unknown _ → None

(* Integration with bug classification *)
val confirm_bug_with_witness :
  cpg:cpg → finding:bug_classification → option concrete_witness
let confirm_bug_with_witness cpg finding =
  match finding with
  | Manifest proof →
    (* Manifest bugs should always have witnesses *)
    let sym_states = symbolic_execute_from_cpg cpg
      proof.triple.entry proof.triple.error_loc 100 in
    List.find_map generate_witness sym_states
  | Latent ctx →
    (* Latent bugs need context satisfied *)
    let sym_states = symbolic_execute_with_precond cpg ctx in
    List.find_map generate_witness sym_states
  | _ → None

```

10.4 Constraint Solver Optimizations

Source: Cadar, Dunbar, Engler 2008 – “KLEE: Unassisted and Automatic Generation of High-Coverage Tests”

KLEE Constraint Optimization: 95% Query Reduction, 10×+ Speedup

SMT solving is the **bottleneck** of symbolic execution. KLEE demonstrates that simple optimizations can dramatically reduce solver load:

Optimization Stack (in order of application):

1. **Expression rewriting** – Simplify before sending to solver
2. **Constraint independence** – Partition unrelated constraints
3. **Counter-example cache** – Reuse previous solver results
4. **Implied value concretization** – Extract concrete values when possible

Empirical Results (from KLEE paper, Table 3):

- STP queries reduced by 95% on COREUTILS
- 10× speedup in overall analysis time
- Counter-example cache hit rate: 92–98%

Optimization 1: Expression Rewriting

```

(* Simplify symbolic expressions before sending to SMT solver.
   Many expressions simplify to constants or simpler forms. *)
val simplify_expr : symbolic_expr → symbolic_expr
let rec simplify_expr e =
  match e with
  (* Additive identity: x + 0 = 0 + x = x *)
  | SymAdd (e1, SymConst 0) → simplify_expr e1
  | SymAdd (SymConst 0, e2) → simplify_expr e2

  (* Multiplicative identity: x * 1 = 1 * x = x *)
  | SymMul (e1, SymConst 1) → simplify_expr e1
  | SymMul (SymConst 1, e2) → simplify_expr e2

  (* Multiplicative zero: x * 0 = 0 * x = 0 *)
  | SymMul (_, SymConst 0) → SymConst 0
  | SymMul (SymConst 0, _) → SymConst 0

  (* Power-of-two multiplication to shift: x * 2^n = x << n *)
  | SymMul (e1, SymConst c) when is_power_of_two c →
    SymShl (simplify_expr e1, SymConst (log2 c))

```

```

(* Constant folding: combine adjacent constants *)
| SymAdd (SymConst c1, SymConst c2) → SymConst (c1 + c2)
| SymMul (SymConst c1, SymConst c2) → SymConst (c1 * c2)
| SymSub (SymConst c1, SymConst c2) → SymConst (c1 - c2)

(* Self-subtraction:  $x - x = 0$  *)
| SymSub (e1, e2) when expr_equal e1 e2 → SymConst 0

(* Boolean simplifications *)
| SymAnd (SymTrue, e2) → simplify_expr e2
| SymAnd (e1, SymTrue) → simplify_expr e1
| SymAnd (SymFalse, _) → SymFalse
| SymOr (SymTrue, _) → SymTrue
| SymNot (SymNot e1) → simplify_expr e1

(* Recursive simplification *)
| SymAdd (e1, e2) → SymAdd (simplify_expr e1, simplify_expr e2)
| _ → e

```

Optimization 2: Constraint Independence

```

(* Partition constraints by shared variables. If constraints share no
variables, they can be solved INDEPENDENTLY. This dramatically reduces
solver complexity since SAT is exponential in variable count.

KLEE INSIGHT: Most path conditions decompose into independent subsets.
A 50-constraint problem might split into 10 independent 5-constraint
problems --- exponentially easier to solve. *)

type constraint_set = {
  constraints : list symbolic_expr;
  variables : set var_id;
}

(* Extract variables from a symbolic expression *)
val get_variables : symbolic_expr → set var_id
let rec get_variables e =
  match e with
  | SymVar v → Set.singleton v
  | SymConst _ → Set.empty
  | SymAdd (e1, e2) | SymSub (e1, e2) | SymMul (e1, e2) →
    Set.union (get_variables e1) (get_variables e2)
  | SymIte (c, t, f) →
    Set.union (get_variables c)
      (Set.union (get_variables t) (get_variables f))
  | SymRead (arr, idx) →
    Set.add arr (get_variables idx)
  | _ → Set.empty

(* Partition constraints into independent sets using union-find *)
val partition_independent : list symbolic_expr → list constraint_set
let partition_independent constraints =
  (* Build union-find over variables *)
  let uf = UnionFind.create () in

  (* For each constraint, union all its variables together *)
  List.iter (fun c →
    let vars = Set.to_list (get_variables c) in
    match vars with
    | [] → ()
    | v :: rest →
      List.iter (fun v' → UnionFind.union uf v v') rest
  ) constraints;

```

```

(* Group constraints by representative variable *)
(* ... partition logic ... *)

```

Optimization 3: Counter-Example Cache

(Cache solver results and exploit subset/superset relationships: *)*

KEY INSIGHTS (from KLEE):

1. If constraint set S is UNSATISFIABLE, any SUPERSET S' is also unsat
2. If constraint set S is SATISFIABLE with model M , any SUBSET S' is also satisfiable (possibly with M)
3. If S has solution M , try M on superset S' --- it might work!

*This exploits the incremental nature of symbolic execution where path conditions grow by adding constraints one at a time. *)*

```

type cex_cache = {
  (* Map from constraint hash to satisfying assignment *)
  sat_cache : map constraint_hash (model * set constraint_hash);
  (* Set of unsatisfiable constraint set hashes *)
  unsat_cache : set constraint_hash;
  (* Statistics *)
  mutable hits : nat;
  mutable misses : nat;
}

val check_with_cache : cex_cache → list symbolic_expr → smt_result
let check_with_cache cache constraints =
  let constraint_set = Set.of_list (List.map hash_constraint constraints) in

  (* CHECK 1: Is any subset unsatisfiable? *)
  let subset_unsat = Set.∃ (fun h →
    Set.mem h cache.unsat_cache
  ) (power_set constraint_set) in
  if subset_unsat then begin
    cache.hits ← cache.hits + 1;
    Unsat
  end
  else
    (* CHECK 2: Does a superset have a solution that works? *)
    (* ... cache lookup logic ... *)

    (* CACHE MISS: Call actual SMT solver *)
    cache.misses ← cache.misses + 1;
    let result = smt_check constraints in
    (* Update cache with result *)
    result

```

Optimization 4: Implied Value Concretization

(When a constraint directly implies a variable's value, extract and substitute it. This reduces symbolic expression complexity. *)*

EXAMPLE: If path condition contains $(x + 1 = 10)$, we can derive $x = 9$ and replace all occurrences of x with concrete value 9.

*KLEE: Writes concretized values back to memory, avoiding future symbolic operations on those values entirely. *)*

```

val extract_implied_values : list symbolic_expr → map var_id int
let extract_implied_values constraints =
  List.fold_left (fun implied c →
    match c with
    (* Direct equality:  $x = c$  *)

```

```

| SymEq (SymVar v, SymConst c) → Map.add v c implied
| SymEq (SymConst c, SymVar v) → Map.add v c implied

(* Linear equation:  $x + c1 = c2 \Rightarrow x = c2 - c1$  *)
| SymEq (SymAdd (SymVar v, SymConst c1), SymConst c2) →
  Map.add v (c2 - c1) implied

| _ → implied
) Map.empty constraints

```

10.5 Concolic Testing Optimizations

Sources:

- Godefroid, Klarlund, Sen 2005 – “DART: Directed Automated Random Testing” (PLDI)
- Sen, Marinov, Agha 2005 – “CUTE: A Concolic Unit Testing Engine for C” (ESEC/FSE)

Concolic Execution: Concrete + Symbolic (DART/CUTE 2005)

Key Insight: Run concrete execution *alongside* symbolic tracking. When symbolic reasoning becomes intractable (e.g., pointers to arrays), **substitute** concrete values and continue.

CUTE Optimizations:

1. **Logical input map** – Decouple memory layout from logical structure
2. **Constraint separation** – Pointer vs arithmetic solved *separately*
3. **Fast unsat check** – Syntactic contradiction detection (60–95% skip)
4. **Incremental solving** – Only re-solve dependent constraints

Empirical Results (from CUTE paper):

- Fast unsat check eliminates 60–95% of SMT calls
- Common sub-constraint elimination: 64–90% reduction
- Incremental solving: only 1/8 constraints re-solved on average

Optimization 1: Logical Input Map

```

(* Decouples LOGICAL structure from PHYSICAL memory layout.

Traditional approach: symbolic pointers are PHYSICAL addresses
Problem: Address arithmetic is undecidable with arrays

CUTE approach: logical input map  $I : N \rightarrow N \cup V$ 
- Maps LOGICAL addresses to values or other logical addresses
- Pointer operations become SIMPLE equality/disequality
- Memory allocation creates NEW logical addresses

EXAMPLE: For input struct { int x; Node* next; }
  I = { 1 ↦ 42,           // p→x = 42
        2 ↦ 3,           // p→next points to logical addr 3
        3 ↦ 17,          // p→next→x = 17
        4 ↦ NULL }      // p→next→next = NULL *)

type logical_address = nat
type primitive_value = int

type logical_value =
| LVPrimitive : v:primitive_value → logical_value
| LVPointer   : addr:logical_address → logical_value
| LVNull      : logical_value

type logical_input_map = {
  mapping : map logical_address logical_value;
  next_addr : logical_address;
  type_info : map logical_address ir_type;
}

```

Optimization 2: Constraint Separation

```
(* CUTE separates constraints into TWO independent classes:

    POINTER CONSTRAINTS:  $p = q, p \neq q, p = \text{NULL}, p \neq \text{NULL}$ 
    - Solved via EQUIVALENCE GRAPH (union-find + diseq edges)
    - Decidable in near-linear time
    - No SMT solver needed!

    ARITHMETIC CONSTRAINTS: linear arithmetic over integers
    -  $a_1x_1 + a_2x_2 + \dots + a_nx_n + c \text{ <cmp> } 0$  where  $\text{<cmp>} \in \{=, \neq, <, \leq, >, \geq\}$ 
    - Solved via ILP solver (lp_solve) or SMT

    KEY INSIGHT: Pointer and arithmetic constraints share NO variables
    (pointers are logical addresses, arithmetic is over values).
    They can be solved COMPLETELY INDEPENDENTLY. *)

type ptr_constraint =
| PCEq : p1:symbolic_ptr → p2:symbolic_ptr → ptr_constraint
| PCNeq : p1:symbolic_ptr → p2:symbolic_ptr → ptr_constraint

type arith_constraint =
(* Linear:  $\sum(a_i * x_i) + c \text{ <cmp> } 0$  *)
| ACLinear : coeffs:list (int * var_id) → constant:int →
    cmp:comparison → arith_constraint

type separated_constraints = {
  ptr_constraints : list ptr_constraint;
  arith_constraints : list arith_constraint;
}

(* Solve pointer constraints via equivalence graph *)
val solve_ptr_constraints : list ptr_constraint → option (map var_id logical_address)
(* Near-linear time via union-find! *)
```

Optimization 3: Fast Unsatisfiability Check

```
(* Before calling the SMT solver, check for SYNTACTIC contradictions.
    If we can detect unsatisfiability cheaply, skip the expensive SMT call.

    CUTE reports: 60-95% of negated constraints are syntactically unsat!

    CHECK 1: Negated constraint equals existing constraint
    Path:  $[x > 0, y < 5]$ 
    Negate  $x > 0$  to get  $x \leq 0$ 
    If path contains  $x \leq 0$ , trivially unsat

    CHECK 2: Implied contradiction
    Path:  $[x = 5]$ 
    Negate to get  $x \neq 5$ 
    Combined with  $x = 5$ , trivially unsat *)

val fast_unsat_check : list symbolic_expr → symbolic_expr → bool
let fast_unsat_check path_condition negated =
  (* CHECK 1: Is negated the negation of something in path? *)
  let syntactic_contradiction = List.∃ (fun c →
    is_negation_of c negated || is_negation_of negated c
  ) path_condition in
  if syntactic_contradiction then true
  else
    (* CHECK 2: Simple propagation for equality chains *)
    match negated with
    | SymNeq (SymVar v, SymConst c) →
      List.∃ (fun pc →
```

```

    match pc with
    | SymEq (SymVar v', SymConst c') when v = v' && c = c' → true
    | _ → false
  ) path_condition
| _ → false

```

Optimization 4: Incremental Solving

(When we negate a constraint to explore a new path, we only need to re-solve the constraints that DEPEND on the changed predicate.*

DEPENDENCY: Constraint C1 depends on C2 if they share variables.

INCREMENTAL ALGORITHM:

- 1. Find constraints dependent on negated predicate*
- 2. Solve ONLY the dependent subset*
- 3. If satisfiable, try to extend old solution*
- 4. Only full re-solve if extension fails*

*CUTE reports: On average, only 1/8 of constraints need re-solving! *)*

```

type incremental_solver_state = {
  constraints : list symbolic_expr;
  solution : option model;
  dependency_graph : map var_id (set nat);
}

val solve_incremental :
  incremental_solver_state →
  negated_idx:nat →
  (incremental_solver_state * option model)
(* Try old model first, only re-solve dependent subset *)

```

10.6 Chopped Symbolic Execution

Source: Trabish, Mattavelli, Rinetzky, Cadar 2018 – “Chopped Symbolic Execution”

Chopped Symbolic Execution: Lazy Recovery for Path Explosion

The Problem: Despite KLEE optimizations (Section 10.4), symbolic execution still suffers from exponential path explosion. Many explored paths are *irrelevant* to the analysis goal.

Example: Hunting heap overflow in libtasn1 requires traversing 2,945 calls to 98 functions (386,727 instructions) – most unrelated to the vulnerability.

The Insight: Allow *skipping* irrelevant code, but don’t simply ignore it. Execute skipped code *lazily* when side effects become *observable*.

Results: 10–100× speedup over baseline KLEE on failure reproduction. CVE-2012-1569: KLEE runs out of memory; Chopper reproduces in < 4 minutes.

10.6.1 The Path Explosion Problem

KLEE (Section 10.4) reduces *constraint solving* cost. CSE addresses *number of paths*:

- Exponential growth: 2^N paths for N branches
- Most paths are *irrelevant* to analysis goal
- Skip irrelevant code, recover on-demand

Complementary: Use *both* for maximum effectiveness:

1. CSE to skip irrelevant code regions
2. KLEE optimizations for constraint solving in relevant regions

10.6.2 The Four State Types

Chopped Symbolic Execution State Types

```
(* STATE KIND HIERARCHY:
   Normal → [call skip] → Snapshot created
   Normal → [load mayMod] → Dependent (suspended)
   Snapshot → [recovery initiated] → Recovery
   Recovery → [return] → Dependent resumes as Normal *)

type cse_state_kind =
| CseNormal      (* Standard symbolic execution state *)
| CseSnapshot    (* Clone saved before entering skip region *)
| CseDependent   (* Suspended state awaiting recovery *)
| CseRecovery    (* Executing sliced skipped function *)

type skip_entry = {
  skipped_func : func_id;
  snapshot : cse_snapshot_state;
  call_context : path_condition;
}

type cse_state = {
  store : symbolic_store;
  heap : symbolic_heap;
  pc : path_condition;
  kind : cse_state_kind;
  skipped : list skip_entry;

  (* For Dependent states *)
  snapshot_ref : option cse_snapshot_state;
  guiding_constraints : path_condition;
  dep_addr : option address;

  (* For Recovery states *)
  recovery_link : option cse_state;
  target_addr : option address;
  sliced_func : option (list ir_stmt);

  (* Optimization: track addresses written after skip *)
  overwritten_set : set address;
}
```

Normal States: Standard symbolic execution state (King 1976). No special handling, code executed symbolically.

Snapshot States: Clone of Normal state created *before* entering skip region. Preserves complete symbolic state at call site boundary. Immutable after creation.

Dependent States: Suspended Normal state that encountered a load where the address *may* have been modified by a skipped function.

Recovery States: State forked from Snapshot with guiding constraints, executing a *statically sliced* version of skipped function to compute value at dependent load address.

10.6.3 Guiding Constraints

Problem: Recovery state forked from snapshot might explore paths *inconsistent* with the dependent state's execution context.

Solution: Add path constraints accumulated *since snapshot* to Recovery state.

$$\text{guiding_constraints} = \text{dependent.pc} - \text{snapshot.pc}$$

$$\text{recovery.pc} = \text{snapshot.pc} \wedge \text{guiding_constraints}$$

Guiding Constraints Computation

```
val get_guiding_constraints :
  current:cse_state{current.kind = CseDependent} →
  snapshot:cse_snapshot_state →
  path_condition
let get_guiding_constraints current snapshot =
  (* Filter: constraints in current.pc but not in snapshot.snap_pc *)
  List.filter (fun c → not (List.mem c snapshot.snap_pc)) current.pc
```

10.6.4 The Recovery Mechanism

When Recovery Triggers:

1. Normal state executes load instruction
2. mayMod(state, state.skipped, addr) returns true
3. Recovery initiated for dependent load

May-Mod Analysis: Uses Andersen-style pointer analysis to compute set of allocation sites each function may modify.

Backward Slicing: Recovery executes *sliced* version of skipped function using PDG.

Recovery State Creation

```
type recovery_result =
  | RecoveryCreated : recovery:cse_state → recovery_result
  | RecoveryInfeasible : recovery_result

val create_recovery_state :
  dependent:cse_state{dependent.kind = CseDependent} →
  addr:address →
  entry:skip_entry →
  recovery_result
let create_recovery_state dependent addr entry =
  let gc = get_guiding_constraints dependent entry.snapshot in
  let recovery_pc = concat entry.snapshot.snap_pc gc in

  (* Check feasibility: guiding constraints must not contradict snapshot *)
  if not (smt_satisfiable recovery_pc) then
    RecoveryInfeasible
  else
    let sliced = static_backward_slice entry.skipped_func addr in
    RecoveryCreated {
      store = entry.snapshot.snap_store;
      heap = entry.snapshot.snap_heap;
      pc = recovery_pc;
      kind = CseRecovery;
      skipped = entry.snapshot.snap_skipped;
      recovery_link = Some dependent;
      target_addr = Some addr;
      sliced_func = Some sliced;
      (* ... other fields ... *)
    }
```

10.6.5 Correctness Properties

Theorem 10.1 (CSE Soundness). *All paths explored by CSE are feasible (satisfiable PC). Exception: Non-termination in skipped functions not detected.*

Theorem 10.2 (Guiding Constraints Correctness). *For all models M : $M \models \text{recovery.pc} \Rightarrow M \models \text{dependent.pc}$*

Theorem 10.3 (Recovery Equivalence). *After recovery completes, dependent has correct value at target address:*

$$\text{lookup}(s'.\text{heap}, \text{addr}) = \text{lookup}(s''.\text{heap}, \text{addr})$$

where s' is full execution and s'' is skip-and-recover execution.

Theorem 10.4 (Relative Completeness). *For bugs not depending on skip internals, CSE will find them (assuming skipped functions terminate).*

10.6.6 Integration with Outcome Logic

| Aspect | Outcome Logic | Chopped SE |
|---|--------------------------|---------------------------|
| Goal | Find bugs (under-approx) | Find bugs (directed) |
| Path selection | Witness paths to bugs | Skip irrelevant code |
| Completeness | Partial (by design) | Relative to skip regions |
| False positives | None (manifest bugs) | None (concrete execution) |
| Recovery | N/A (permanent drop) | On-demand when relevant |
| <i>Both focus on relevant paths, not exhaustive exploration</i> | | |

10.7 Optimistic Concolic Execution (QSYM)

Source: Yun, Lee, Xu, Jang, Kim 2018 – “QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing” (USENIX Security)

Optimistic Concolic Execution: 10–100× Faster via Validation-Based Soundness

QSYM Key Insight: In hybrid fuzzing, the fuzzer provides *validation*. The concolic executor doesn’t need to be perfectly sound – generate candidate inputs optimistically, let the fuzzer validate them by execution.

Scalability Wall (why traditional concolic fails):

1. Symbolic emulation overhead: IR translation (VEX/LLVM) is 10–100× slow
2. Environment modeling: syscalls, libraries often unsupported
3. Constraint explosion: full soundness = massive constraint sets
4. Path explosion: even with fuzzer help, too many paths

QSYM Solutions:

1. Native execution with Intel Pin instrumentation (2–5× vs 10–100×)
2. Optimistic constraint solving with fuzzer validation
3. Basic block pruning by CFG distance to targets
4. Concrete fallback for complex operations (FP, syscalls)

Empirical Results: 13 new bugs in heavily-fuzzed software (ffmpeg, OpenJPEG)

Execution Models: IR Interpretation vs Native Instrumentation

(Traditional (KLEE, Driller): Binary → IR → Interpret symbolically (SLOW)*
*QSYM: Binary → Native execution + selective instrumentation (FAST) *)*

```
type execution_model =
| IRInterpretation      (* KLEE-style: lift to IR, interpret symbolically *)
| NativeInstrumented    (* QSYM-style: native exec + symbolic tracking *)
| FullEmulation         (* S2E-style: full system emulation *)
```

```
type instrumentation_granularity =
| AllInstructions        (* Every instruction runs symbolic *)
| SymbolicDataOnly      (* Only when symbolic data touched - QSYM default *)
| CFGRelevantOnly       (* Only at branch points affecting targets *)
```

(QSYM instruments ONLY instructions touching symbolic data.*
*Non-symbolic code runs at near-native speed. *)*

Optimistic Constraint Simplification

```
(* QSYM drops constraints that are expensive to solve, relying on
   the fuzzer to validate generated inputs by concrete execution. *)

type simplification_strategy =
| DropFloatingPoint      (* FP operations are hard for SMT *)
| DropUnmodeledCalls     (* Use return value from concrete exec *)
| ConcreteSymbolicPointers (* Resolve symbolic ptrs to concrete addrs *)
| PruneDistantBlocks     (* Drop constraints from far-away code *)

let qsym_optimistic_config : optimistic_config = {
  strategies = [
    DropFloatingPoint;
    DropUnmodeledCalls;
    ConcreteSymbolicPointers;
    PruneDistantBlocks;
  ];
  max_constraint_size = 1000;
  solver_timeout_ms = 1000;      (* 1 second per query - fail fast *)
  validation_required = true;
}

val simplify_optimistically :
  config:optimistic_config →
  constraints:list symbolic_expr →
  concrete:concrete_state →
  list symbolic_expr
(* Apply strategies, drop oversized constraints *)
```

Basic Block Pruning by CFG Distance

```
(* Not all code is equally relevant. QSYM prioritizes blocks CLOSE to
   uncovered branches, pruning constraints from distant code. *)

type block_relevance = {
  cfg_distance : nat;          (* Hops to nearest uncovered branch *)
  data_flow_score : float;     (* How much data flows to branches *)
  execution_count : nat;       (* Times hit in concrete trace *)
}

val compute_block_relevance :
  cfg:cfg →
  block:basic_block_id →
  uncovered:set basic_block_id →
  block_relevance

type pruning_config = {
  max_cfg_distance : nat;      (* Prune if further than this *)
  prune_unexecuted : bool;    (* Prune blocks not in concrete trace *)
}

val prune_constraints_by_distance :
  constraints:list (basic_block_id * symbolic_expr) →
  cfg:cfg →
  targets:set basic_block_id →
  config:pruning_config →
  list symbolic_expr
```

Hybrid Fuzzing Integration

```
(* Fuzzer and concolic executor work cooperatively:
   - Fuzzer finds easy paths quickly (loose constraints)
   - Concolic solves hard constraints (tight like x == 0xdeadbeef)
   - Fuzzer validates concolic-generated inputs *)
```

```

type hybrid_result =
  | NewCoverage : inputs:list bytes → new_blocks:set basic_block_id → hybrid_result
  | CrashFound : input:bytes → crash_info:crash_info → hybrid_result
  | NoProgress : hybrid_result

val hybrid_iteration :
  program:string →
  inputs:list bytes →
  coverage:set basic_block_id →
  hybrid_result

let hybrid_iteration program inputs coverage =
  (* 1. Run fuzzer for a while *)
  let (fuzz_inputs, fuzz_coverage) = run_fuzzer program inputs in

  (* 2. Find branches where fuzzer is stuck *)
  let uncovered = Set.diff (get_all_branches program) fuzz_coverage in
  let stuck = find_most_promising uncovered 5 in

  (* 3. Run optimistic concolic on stuck branches *)
  let concolic_inputs = List.concat_map (fun target →
    let trace = get_concrete_trace program inputs target in
    let constraints = collect_path_constraints trace in
    let simplified = simplify_optimistically
      qsym_optimistic_config constraints trace in
    match optimistic_solve simplified 1000 with
    | Sat model → [model_to_input model]
    | _ → []
  ) stuck in

  (* 4. Validate concolic inputs with fuzzer *)
  let validated = List.filter_map (fun input →
    match run_with_coverage program input with
    | (cov, NormalExit) when not (Set.is_empty (Set.diff cov coverage)) →
      Some input (* New coverage! *)
    | (_, Crash crash) →
      Some input (* Crashes are valuable! *)
    | _ → None (* Didn't help - optimistic solving produced junk *)
  ) concolic_inputs in
  (* ... return result ... *)

```

10.7.1 Soundness Model: Validation-Based

QSYM is *not* sound for path reachability (may generate invalid inputs). QSYM *is* sound for bug finding: every reported crash is validated.

| | Sound (KLEE) | Best-Effort (CUTE) | Optimistic (QSYM) |
|-----------------|--------------------|--------------------|-----------------------------|
| Guarantees | Correct + Complete | Correctness | Neither (validated) |
| False positives | None | Rare | Frequent but validated away |
| Scalability | ~100K LOC | Medium | Millions of LOC |
| Use case | Verification | Testing | Bug hunting |

Complementary Use:

- KLEE for small, critical components (crypto, parsers) – proof quality
- QSYM for large codebases (ffmpeg, Chrome) – bug hunting at scale

Cross-References:

- Section 10.4: KLEE optimizations (sound – apply first)
- Section 10.5: CUTE optimizations (constraint separation)
- Section 10.6: Chopped Symbolic Execution (function-level pruning)

- Section 9.4: Hybrid IFDS+ Eval architecture (add fuzzer cooperation)

Part V

Pointer Analysis — The Precision Foundation

Tension Resolution: Steensgaard Default vs Language-Specific

Source: [Rupta24], [Lattner07] DSA. See Appendix D.10.4 for full analysis.

OLD RECOMMENDATION: “Steensgaard for speed, Andersen for precision”

UPDATED: Language AND codebase size matter!

Language-Specific Recommendations:

- **Rust:** 1-callsite-sensitive + stack filtering (Section 12.22) — *FASTER AND more precise than Steensgaard for Rust!*
- **C/C++ (< 50K LOC):** Andersen (most precise, subset-based)
- **C/C++ (> 100K LOC):** DSA (Section 5.2.5) — unification + heap cloning. Linux kernel (355K LOC) in 3.1 seconds, < 46MB memory. Precision *comparable* to Andersen via context-sensitivity.
- **C/C++ source-sink:** SVF (Section 5.6) — uses DSA/Andersen as input
- **C/C++ flow-sensitive:** Demand-driven ([Sridharan05])
- **Java/OOP:** Qilin + ZIPPER (Sections 5.3, 5.3.2) for selective CS
- **Python/JavaScript:** Type-based + dynamic traces

Key Insight: Rust’s ownership model changes aliasing patterns dramatically. Stack filtering (Section 5.3, 12.22) eliminates spurious stack aliasing. DSA’s heap cloning distinguishes data structure *instances* at scale.

Tension Resolution: Field Index vs Projection Path

Source: [Rupta24]. See Appendix D.10.3 for full analysis.

This Section uses field INDEX: `FieldLoad(dst, base, field_index)`

Rupta 2024 uses PROJECTION PATH: `(base, [field1, field2, field3])`

Trade-off:

- **Index-based:** Simpler, sufficient for flat structs
- **Projection-based:** More precise for *nested* structs — `x.a.b.c` and `y.a.b.c` distinguished by full path. Required for Rust where nested borrows are common.

Recommendation: Use projection-based for Rust/C++ with deep nesting. Section 12.22 provides `typed_loc` with `type_view` for cast handling.

Empirical Finding: Field-Dereference Depth Bound

Source: [TAJ09]

When tracking taint through field accesses (e.g., `x.a.b.c`), how deep should analysis go? TAJ found an empirical bound:

$k = 2$ is SUFFICIENT for 95%+ of real vulnerabilities

Meaning: Most taint flows involve at most 2 levels of field dereference:

- **COMMON:** `x.field` ($k = 1$)
- **COMMON:** `x.outer.inner` ($k = 2$)
- **RARE:** `x.a.b.c.d` ($k = 4$) — diminishing returns

Recommendation:

- Default to $k = 2$ for field-dereference depth
- Make configurable for specific analyses requiring deeper tracking
- Cost grows exponentially with k ; keep it small

Cross-reference: TAJ taint analysis (Section 8.1.3, 8.1.5)

11. Andersen’s Analysis

Foundational Reference: Andersen 1994

[Andersen94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

Andersen’s analysis is the gold standard for precise points-to analysis. It models pointer operations as set constraints.

11.1 The Constraint Language

Pointer operations translate to set inclusion constraints:

| Statement | Constraint | Meaning |
|-----------|---|---|
| $x = \&y$ | $\{y\} \subseteq pts(x)$ | x points to y ’s location |
| $x = y$ | $pts(y) \subseteq pts(x)$ | x points to everything y points to |
| $x = *y$ | $\forall v \in pts(y). pts(v) \subseteq pts(x)$ | x points to what y ’s targets point to |
| $*x = y$ | $\forall v \in pts(x). pts(y) \subseteq pts(v)$ | y ’s targets added to x ’s targets’ pts |

Remark 11.1 (Example Constraint Solving). Consider:

```

p = &x          {x} <= pts(p)
q = &y          {y} <= pts(q)
r = p           pts(p) <= pts(r)
s = *r          forall v in pts(r). pts(v) <= pts(s)

```

Solution:

```

pts(p) = {x}
pts(q) = {y}
pts(r) = {x} (from pts(p))
pts(s) = pts(x) (from dereferencing r which points to x)

```

11.2 Constraint Solving

The F* formalization below defines the core types for Andersen’s analysis. The `abstract_loc` type distinguishes between stack, heap, and global memory locations—this stratification is essential for precision, as different storage classes have different aliasing behaviors. The `constraint_` type directly encodes the four fundamental pointer operations (address-of, copy, load, store) plus field operations for struct handling.

Andersen’s Points-To Analysis

```

(* =====
   ANDERSEN’S POINTS-TO ANALYSIS
   Source: Andersen 1994
   ===== *)
module BrrrMachine.PointerAnalysis.Andersen

type abstract_loc =
| StackLoc : func:string → var:string → abstract_loc
| HeapLoc  : alloc_site:node_id → abstract_loc
| GlobalLoc : var:string → abstract_loc
| UnknownLoc : abstract_loc

type constraint_ =

```

```

| AddrOf : lhs:string → rhs:abstract_loc → constraint_
  (* lhs = &rhs: {rhs} ≤ pts(lhs) *)
| Copy : lhs:string → rhs:string → constraint_
  (* lhs = rhs: pts(rhs) ≤ pts(lhs) *)
| Load : lhs:string → rhs:string → constraint_
  (* lhs = *rhs: ∀ v in pts(rhs). pts(v) ≤ pts(lhs) *)
| Store : lhs:string → rhs:string → constraint_
  (* *lhs = rhs: ∀ v in pts(lhs). pts(rhs) ≤ pts(v) *)
| FieldLoad : lhs:string → base:string → field:string → constraint_
  (* lhs = base→field *)
| FieldStore : base:string → field:string → rhs:string → constraint_
  (* base→field = rhs *)

type pts_solution = map string (set abstract_loc)

```

The `pts_solution` type maps each variable name to its points-to set—a set of abstract locations that the variable may reference at runtime. This is the central data structure computed by the analysis; all client analyses (taint tracking, use-after-free detection, etc.) query this map to determine aliasing relationships.

The following code demonstrates how constraints are extracted from the Code Property Graph (CPG). Each assignment node is classified by its pointer operation type, and the corresponding constraint is generated. This extraction is language-independent; the CPG abstraction (Section 3.1) normalizes language-specific syntax into a uniform representation.

Constraint Extraction from CPG

```

val extract_constraints : cpg → list constraint_
let extract_constraints cpg =
  fold_nodes cpg (fun constraints node →
    match node.kind with
    (* x = &y *)
    | NAssign when is_addr_of cpg node.id →
      let lhs = get_lhs cpg node.id in
      let rhs = get_addr_of_target cpg node.id in
      AddrOf lhs rhs :: constraints
    (* x = y *)
    | NAssign when is_simple_copy cpg node.id →
      let lhs = get_lhs cpg node.id in
      let rhs = get_rhs_var cpg node.id in
      Copy lhs rhs :: constraints
    (* x = *y *)
    | NAssign when is_load cpg node.id →
      let lhs = get_lhs cpg node.id in
      let rhs = get_deref_target cpg node.id in
      Load lhs rhs :: constraints
    (* *x = y *)
    | NAssign when is_store cpg node.id →
      let lhs = get_store_target cpg node.id in
      let rhs = get_rhs_var cpg node.id in
      Store lhs rhs :: constraints
    (* x = y→f *)
    | NAssign when is_field_load cpg node.id →
      let (lhs, base, field) = get_field_load_info cpg node.id in
      FieldLoad lhs base field :: constraints
    (* x→f = y *)
    | NAssign when is_field_store cpg node.id →
      let (base, field, rhs) = get_field_store_info cpg node.id in
      FieldStore base field rhs :: constraints
    | _ → constraints
  ) []

```

The constraint extraction function `extract_constraints` performs a single pass over the CPG, producing a list of constraints. The helper functions (`is_addr_of`, `get_lhs`, etc.) are CPG query operations that inspect node attributes. This design separates the constraint *language*

from the constraint *extraction*, enabling the same solver to work with constraints from different front-ends.

The worklist algorithm below is the heart of Andersen’s analysis. It iteratively propagates points-to information until a fixpoint is reached. The key insight is that only variables whose points-to sets have *changed* need to be re-examined—the worklist tracks these “dirty” variables. For load/store constraints, the algorithm must handle the indirection: when $\text{pts}(y)$ changes, any constraint of the form $x = *y$ must be re-evaluated.

Iterative Worklist Algorithm

```
val solve_andersen : list constraint_ → pts_solution
let solve_andersen constraints =
  (* Initialize with base constraints (AddrOf) *)
  let init = List.fold_left (fun sol c →
    match c with
    | AddrOf lhs rhs →
      Map.update lhs (fun s →
        Set.add rhs (Option.value s ~default:Set.empty)) sol
    | _ → sol
  ) Map.empty constraints in

  (* Build constraint graph for worklist *)
  let copy_edges = List.filter_map (fun c →
    match c with Copy lhs rhs → Some (rhs, lhs) | _ → None
  ) constraints in
  let load_constraints = List.filter (fun c →
    match c with Load _ _ → true | _ → false
  ) constraints in
  let store_constraints = List.filter (fun c →
    match c with Store _ _ → true | _ → false
  ) constraints in

  (* Worklist: variables whose pts changed *)
  let worklist = ref (Map.keys init) in
  let solution = ref init in

  while not (List.is_empty !worklist) do
    let var = List.hd !worklist in
    worklist := List.tl !worklist;
    let pts_var = Map.find_default var Set.empty !solution in

    (* Propagate through copy edges *)
    List.iter (fun (src, dst) →
      if src = var then
        let old_pts = Map.find_default dst Set.empty !solution in
        let new_pts = Set.union old_pts pts_var in
        if not (Set.equal old_pts new_pts) then begin
          solution := Map.add dst new_pts !solution;
          worklist := dst :: !worklist
        end
      end
    ) copy_edges;

    (* Handle complex constraints *)
    List.iter (fun c →
      match c with
      | Load lhs rhs when rhs = var →
        Set.iter (fun loc →
          let loc_pts = Map.find_default (loc_to_var loc) Set.empty !solution in
          let old_pts = Map.find_default lhs Set.empty !solution in
          let new_pts = Set.union old_pts loc_pts in
          if not (Set.equal old_pts new_pts) then begin
            solution := Map.add lhs new_pts !solution;
            worklist := lhs :: !worklist
          end
        ) rhs
      | _ → ()
    ) store_constraints
```

```

        end
      ) pts_var
    | Store lhs rhs when lhs = var →
      let rhs_pts = Map.find_default rhs Set.empty !solution in
      Set.iter (fun loc →
        let loc_var = loc_to_var loc in
        let old_pts = Map.find_default loc_var Set.empty !solution in
        let new_pts = Set.union old_pts rhs_pts in
        if not (Set.equal old_pts new_pts) then begin
          solution := Map.add loc_var new_pts !solution;
          worklist := loc_var :: !worklist
        end
      ) pts_var
    | _ → ()
  ) (load_constraints @ store_constraints)
done;
!solution

```

Remark 11.2 (Complexity Analysis). Let n = number of variables/locations.

Worst Case: $\mathcal{O}n^3$

- Each variable can point to $\mathcal{O}n$ locations
- Each change propagates to $\mathcal{O}n$ dependents
- $\mathcal{O}n$ variables can change

Practical: Much better on real programs due to sparse constraint graphs.

Comparison (language-dependent — see Part V tension box):

- **Andersen:** $\mathcal{O}n^3$, precise (inclusion-based)
- **Steensgaard:** $\mathcal{O}n \cdot \alpha(n)$, less precise (unification-based)
- **Rust:** 1-callsite + stack filtering (Section 12.22) — *faster* than Steensgaard!

The traditional “Steensgaard=fast, Andersen=precise” trade-off does **not** hold for all languages. Rust’s ownership model changes aliasing patterns.

11.3 Limitations from Andersen 1994

Critical: Limitations from Andersen 1994 — Avoid These Extensions

The paper explicitly identifies problems with no good solutions:

1. Flow-Sensitive Constraint-Based Analysis

“Currently, we have no good solution to this problem.” (Section 4.10)

Problem: When $x = y$ updates abstract location p at point 3, but p does not syntactically appear in $*q = \&y$ at point 4, flow-sensitive constraints cannot express this update.

Recommendation: Use flow-INSENSITIVE analysis. For most programs with many small functions, this is sufficient.

2. Must-Alias Analysis

“We shall only consider may point-to analysis in this chapter.”

Must-alias requires flow-sensitive reasoning to determine when two pointers *definitely* alias. This is significantly more expensive.

Recommendation: Start with may-alias. Add must-alias only when needed for specific optimizations.

3. Unknown Explosion from LHS Store Dereferences

“If the analysis reveals that an Unknown pointer may be dereferenced in the left hand side of an assignment, the analysis stops with ‘worst-case’ message.” (Section 4.3.3)

Problem: $*x = y$ where $pts(x)$ contains Unknown is CATASTROPHIC. Everything becomes Unknown transitively.

Recommendation:

- Track which constraints come from LHS of stores

- Only abort when LHS dereference involves Unknown
- RHS dereferences with Unknown are safe (just propagate Unknown)

4. Full Array Element Tracking

“Our analysis treats arrays as aggregates.” (Section 4.12.1)

Tracking individual array elements requires array dependence analysis from parallelizing compilers.

5. Recursive Data Structure Unrolling

“In our experience elements in a recursive data structure are used ‘the same way’” — merge all nodes into single abstract location. For precise tracking, use Shape Analysis (Section 5.4).

11.4 Inter-Procedural Context Sensitivity

Definition 11.3 (Context-Sensitive Extension via Constraint Vectors). **Source:** [Ander-
sen94], Section 4.6

Instead of copying functions for each call site (exponential blowup), use *vectors* of type variables indexed by call-site variants:

$$\langle T^0, T^1, \dots, T^n \rangle$$

where:

- T^0 = summary (context-insensitive)
- T^i = specific call context i

At each call site, the Static Call Graph determines which variant index to use. This achieves context sensitivity **without** exponential cloning of constraints.

IMPLEMENTATION:

```
struct ContextSensitiveVar {
    base: VarId,
    variant: u32, // 0 = summary, 1..n = specific call contexts
}

fn get_callee_variant(
    call_site: u32,
    caller_variant: u32,
    scg: &StaticCallGraph
) -> u32 {
    scg.lookup(call_site, caller_variant)
}
```

Trade-off:

- More precise than context-insensitive (fewer spurious flows)
- Less precise than full context cloning (variants merge at recursion)
- Polynomial complexity preserved

11.5 Cross-References: Pointer Analysis Dependencies

What Depends on Pointer Analysis ($pts()$):

1. **Effect Resolution** (Section 6.1): $ERead(loc)$, $EWrite(loc)$ need $pts()$ to determine which locations are accessed. Without $pts()$, $Read(*p)$ affects “everything”.
2. **Use-After-Free Detection** (Section 6.1.4): $Free(ptr)$ marks $pts(ptr)$ as freed. Read-/Write checks if any target in $pts()$ is in freed set.
3. **Null Dereference Detection** (Section 2.1.7): Track nullable sources through pointer

assignments. $pts()$ determines which pointers may contain null.

4. **Data Race Detection** (Section 6.1.4): Two accesses conflict if $pts()$ sets overlap AND at least one is a write AND not synchronized.
5. **Taint Analysis** (Section 8.1): $Source(tainted) \rightarrow^* Sink(tainted)$ through pointer indirection. $*p = tainted_value$ taints $pts(p)$.
6. **Call Graph Construction** (Section 5.3): Virtual/indirect calls: $target = pts(function_pointer)$.
7. **IFDS Precision** (Section 4.1): IFDS is **not** applicable to pointer analysis (non-distributive), but IFDS analyses (taint, uninit) *use* $pts()$ for precision.
8. **Sparse Value-Flow Analysis** (Section 5.6): SVF **REQUIRES** $pts()$ for Memory SSA construction. μ/χ annotations use $pts()$ to determine affected regions.
9. **Alternative for Large C/C++ Codebases** (Section 5.2.5): DSA provides $\mathcal{O}n \cdot \alpha(n)$ pointer analysis with heap cloning. For $> 100K$ LOC: DSA faster than Andersen, precision via context-sensitivity.

12. Steensgaard's Analysis

Foundational Reference: Steensgaard 1996

[Steensgaard96] B. Steensgaard. *Points-to Analysis in Almost Linear Time*. POPL 1996. Steensgaard's analysis trades precision for speed using unification. Key insight: Use **equality** instead of **subset**. $x = y$ means $pts(x) = pts(y)$, not $pts(y) \subseteq pts(x)$. This enables union-find for $\mathcal{O}n \cdot \alpha(n)$ complexity.

Remark 12.1 (Language-Dependent Trade-off). This trade-off is **language-dependent**. For Rust, context-sensitive analysis with stack filtering (Section 12.22) is often *faster* due to reduced spurious aliasing. See tension resolution at start of Part V.

The F* formalization below implements Steensgaard's unification-based analysis using the classic union-find data structure. The `uf_node` type stores both the union-find parent pointer and the points-to set for each equivalence class representative. Path compression (`find`) and union-by-rank (`union`) ensure near-linear time complexity.

The key difference from Andersen is in `union`: when two variables are assigned ($x = y$), their equivalence classes are *merged*, causing all members to share the same points-to set. This loses precision but enables the $\mathcal{O}n \cdot \alpha(n)$ complexity that makes Steensgaard practical for very large codebases.

Steensgaard's Points-To Analysis

```
(* =====
STEENSGAARD'S POINTS-TO ANALYSIS
Source: Steensgaard 1996
===== *)
module BrrrMachine.PointerAnalysis.Steensgaard

(* Union-find with path compression *)
type uf_node = {
  mutable parent : int;
  mutable rank   : int;
  mutable pts    : set abstract_loc;
}

type union_find = array uf_node

val find : union_find → int → int
let rec find uf i =
  if uf.(i).parent = i then i
  else begin
```

```

    uf.(i).parent ← find uf uf.(i).parent;  (* Path compression *)
    uf.(i).parent
end

val union : union_find → int → int → unit
let union uf i j =
  let ri = find uf i in
  let rj = find uf j in
  if ri ≠ rj then begin
    (* Union by rank *)
    if uf.(ri).rank < uf.(rj).rank then
      uf.(ri).parent ← rj
    else if uf.(ri).rank > uf.(rj).rank then
      uf.(rj).parent ← ri
    else begin
      uf.(rj).parent ← ri;
      uf.(ri).rank ← uf.(ri).rank + 1
    end;
    (* Merge points-to sets *)
    let root = find uf i in
    uf.(root).pts ← Set.union uf.(ri).pts uf.(rj).pts
  end
end

```

The union-find operations above maintain the invariant that equivalent variables share a single representative. When points-to sets are merged during `union`, the combined set is stored only at the new root—this avoids duplicating storage across the equivalence class.

The solving algorithm below processes each constraint type differently. `AddrOf` constraints directly add to points-to sets. `Copy` constraints trigger unification. `Load` and `Store` constraints are more complex: they unify through the indirection, which can cause significant precision loss when many variables flow together. The final solution is extracted by finding the root of each variable's equivalence class.

Steensgaard Solving Algorithm

```

val solve_steensgaard : list constraint_ → pts_solution
let solve_steensgaard constraints =
  let vars = collect_all_vars constraints in
  let var_to_id = Map.of_list (List.mapi (fun i v → (v, i)) vars) in
  let n = List.length vars in

  (* Initialize union-find *)
  let uf = Array.init n (fun i → { parent = i; rank = 0; pts = Set.empty }) in

  (* Process constraints *)
  List.iter (fun c →
    match c with
    | AddrOf lhs rhs →
      let id = Map.find lhs var_to_id in
      uf.(id).pts ← Set.add rhs uf.(id).pts
    | Copy lhs rhs →
      (* Key difference from Andersen: UNIFY instead of subset *)
      let id_lhs = Map.find lhs var_to_id in
      let id_rhs = Map.find rhs var_to_id in
      union uf id_lhs id_rhs
    | Load lhs rhs →
      let id_lhs = Map.find lhs var_to_id in
      let id_rhs = find uf (Map.find rhs var_to_id) in
      Set.iter (fun loc →
        let id_loc = Map.find (loc_to_var loc) var_to_id in
        union uf id_lhs id_loc
      ) uf.(id_rhs).pts
    | Store lhs rhs →
      let id_lhs = find uf (Map.find lhs var_to_id) in

```

```

    let id_rhs = Map.find rhs var_to_id in
  Set.iter (fun loc →
    let id_loc = Map.find (loc_to_var loc) var_to_id in
    union uf id_loc id_rhs
  ) uf.(id_lhs).pts
| _ → ()
) constraints;

(* Extract solution *)
Map.mapi (fun var id →
  let root = find uf id in
  uf.(root).pts
) var_to_id

```

Remark 12.2 (Complexity and Precision Loss). **Complexity:** $\mathcal{O}n \cdot \alpha(n) \approx \mathcal{O}n$ where α is the inverse Ackermann function (effectively constant).

Precision Loss Example:

```

p = &x
q = &y
r = p
s = q

```

Andersen: $pts(p) = \{x\}$, $pts(q) = \{y\}$, $pts(r) = \{x\}$, $pts(s) = \{y\}$

Steensgaard (with $r = q$ somewhere): All unified: $pts(p) = pts(q) = pts(r) = pts(s) = \{x, y\}$

Use Cases:

- Fast pre-analysis to identify “interesting” pointers
- Scalability-first scenarios (millions of LOC)
- When precision loss is acceptable

12.1 Conditional Join (cjoin)

Definition 12.3 (Conditional Join). **Source:** [Steensgaard96], Figures 5 & 6

Key insight: When joining two ECRs where one has type \perp (unknown), **defer** the join until the type becomes known. This prevents premature merging that loses precision.

- **Without cjoin:** $x = y$ immediately unifies $pts(x)$ and $pts(y)$
- **With cjoin:** If $pts(y) = \perp$, record “join with x when y gets a type”

This is critical for handling uninitialized pointers and forward references.

The implementation below extends the basic Steensgaard analysis with conditional joins. The `steens_type` distinguishes between `TBot` (type unknown) and `TRef` (pointer with known target). The `pending` map tracks deferred joins: when we try to join with a variable whose type is still \perp , we record the pending join and execute it later when `settype` assigns a concrete type.

This deferred execution pattern is essential for handling forward references in the program—a pointer may be used before its allocation site is analyzed. The `cjoin` function implements the conditional logic: if the target type is known, join immediately; otherwise, record the pending join.

Conditional Join Implementation

```

(* Steensgaard type:  $\tau$  (location)  $x$  lambda (function signature) *)
type steens_type =
| TBot                                     (* Bottom: type unknown *)
| TRef :  $\tau$ :ecr_id → lam:ecr_id → steens_type (* Pointer to  $\tau$ , function lam *)

(* Extended state with pending conditional joins *)

```

```

type steens_state = {
  uf : union_find;
  types : map ecr_id steens_type;
  pending : map ecr_id (list ecr_id);  (* Pending joins: ecr → waiting ECRs *)
}

(* Conditional join: defer if target type is bottom *)
val cjoin : steens_state → ecr_id → ecr_id → steens_state
let cjoin st ecr1 ecr2 =
  let ecr2' = find st.uf ecr2 in
  match Map.find ecr2' st.types with
  | None | Some TBot →
    (* Type unknown: DEFER the join until type is set *)
    let waiting = Map.find_default ecr2' [] st.pending in
    let pending' = Map.add ecr2' (ecr1 :: waiting) st.pending in
    { st with pending = pending' }
  | Some _ →
    (* Type known: perform immediate join via union *)
    { st with uf = union st.uf ecr1 ecr2 }

(* Set type and trigger all pending conditional joins *)
val settype : steens_state → ecr_id → steens_type → steens_state
let settype st ecr ty =
  let ecr' = find st.uf ecr in
  let st' = { st with types = Map.add ecr' ty st.types } in
  let waiting = Map.find_default ecr' [] st'.pending in
  let st'' = List.fold_left (fun s pending_ecr →
    { s with uf = union s.uf ecr' pending_ecr })
    st' waiting in
  { st'' with pending = Map.remove ecr' st''.pending }

```

12.2 Data Structure Analysis (DSA)

Foundational Reference: Lattner, Lenharth, Adve 2007

[Lattner07] C. Lattner, A. Lenharth, V. Adve. *Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World*. PLDI 2007.

DSA: Unification + Context-Sensitivity + Heap Cloning

Key Insight: The “fast vs precise” dichotomy is **false** at scale.

DSA combines:

- **Unification-based** (like Steensgaard) for $\mathcal{O}n \cdot \alpha(n)$ base complexity
- **Context-sensitivity** to recover precision lost by unification
- **Heap cloning by acyclic call paths** to distinguish data structure *instances*

Result: Precision comparable to Andersen, speed of Steensgaard

- Linux kernel (355K LOC): 3.1 seconds, < 46MB memory
- Analysis time < 5% of GCC compile time

When to Use DSA (vs Andersen):

- Codebase > 100K LOC where Andersen is too slow
- Heavy use of wrapper functions around malloc
- Generic data structure libraries (`std::vector`, linked lists)
- Need to distinguish instances created through same allocator

12.2.1 DS Graph Structure

DSA represents points-to information using **Data Structure Graphs (DS Graphs)**:

CROSS-REFERENCE: The F* type definitions for DSA are in Section 12.33.

See Section 12.33.1 for:

- `ds_flag`: Node flags (Heap, Stack, Global, Unknown, Array, Modified, Read, Complete, Collapsed)
- `ds_node`: Memory object representation with fields and flags
- `ds_graph`: Per-function graph with nodes, edges, and variable mapping
- `call_node`: Unresolved indirect call representation
- `dsa_state`: Full analysis state across all functions

CONCEPTUAL SUMMARY:

DS Graphs use disjoint-set data structures where:

- Nodes represent abstract memory locations (heap, stack, global)
- Fields provide field-sensitive pointer edges
- Flags track provenance (H/S/G/U) and analysis state (M/R/C/O/A)
- The Complete (C) flag is CRITICAL - see below

12.2.2 The Completeness Flag

Definition 12.4 (Completeness Flag Semantics). A node is **Complete** when all operations on it have been analyzed:

- All callers/callees have been incorporated
- No unknown external code can modify it
- Node will **never** be merged with other nodes in subsequent phases

Aliasing Rules:

- Two nodes **without** C flag may represent common objects \Rightarrow must assume they may alias (conservative)
- Two nodes **with** C flag \Rightarrow guaranteed distinct (precise)

Use Cases:

1. **Incomplete Programs**: External functions create incomplete nodes
2. **Incremental Analysis**: Only reanalyze incomplete portions
3. **Speculative Field-Sensitivity**: Assume type-safe until proven wrong
4. **Library Analysis**: Mark library boundaries as incomplete

12.2.3 Three-Phase Algorithm

DSA ALGORITHM STRUCTURE:

PHASE 1: LOCAL ANALYSIS (per function)

- Build DS graph using only intraprocedural information
- Track loads, stores, allocations, direct calls
- Create call nodes for unresolved indirect calls
- Mark all nodes as INCOMPLETE (no C flag)

PHASE 2: BOTTOM-UP (BU) ANALYSIS

- Traverse call graph SCCs in POSTORDER
- Clone callee graphs into caller at call sites
- Merge argument/return cells
- Incrementally resolve indirect calls as function pointers become known
- Use extended Tarjan's SCC algorithm with revisitation

PHASE 3: TOP-DOWN (TD) ANALYSIS

- Traverse call graph SCCs in REVERSE POSTORDER
- Merge caller context into callees

- Mark nodes COMPLETE when all callers incorporated
- Final completeness determination

COMPLEXITY:

Time: $O(n \cdot \alpha(n) + K \cdot \alpha(K) \cdot e)$

Space: $O(f \cdot K)$

where:

n = number of instructions

K = maximum DS graph size

e = number of call graph edges

f = number of functions

alpha = inverse Ackermann function (effectively constant)

12.2.4 Heap Cloning: Distinguishing Data Structure Instances

DSA's heap cloning by acyclic call paths distinguishes instances of data structures:

HEAP CLONING EXAMPLE:

```
// Two disjoint linked lists created via same wrapper
list *X = makeList(10); // Call context: main -> makeList
list *Y = makeList(100); // Call context: main -> makeList (different site)

void addToList(list *L) {
    // Without heap cloning: X and Y conflated (same malloc site)
    // With heap cloning: X and Y DISTINGUISHED by call context
}
```

DSA proves X and Y are DISJOINT because:

- They reach malloc through DIFFERENT acyclic call paths
- Each path creates a separate DS node
- No aliasing between nodes with different call contexts

COMPARISON WITH ANDERSEN:

- Andersen uses allocation-site naming: ONE abstract object per malloc
- DSA uses call-path naming: SEPARATE objects per call context
- For wrapper-heavy code: DSA more precise AND faster

12.2.5 Engineering Optimizations

$O(N^2)$ ELIMINATION TECHNIQUES:

1. GLOBALS GRAPH

Problem: Propagating globals through call tree is $O(N^2)$

Solution: Separate graph for global-reachable nodes

- Functions reference the globals graph, not copy it
- Speedup: 2-3.5x on large programs

2. GLOBAL EQUIVALENCE CLASSES

Problem: Many globals point to same DS node

Solution: Keep only representative global per DS node

- Reduces EV entries from $O(N^2)$ to $O(N)$
- Speedup: up to 21.8x on Linux kernel

3. EFFICIENT GRAPH INLINING

Problem: Cloning allocates nodes only to discard them

Solution: Recursive traversal from common pointers only

- Only visit nodes that will be reflected in target
- Avoid allocate-then-discard pattern

GENERALIZABLE PATTERNS:

- Separate representation for global entities
- Representative-based compression
- Lazy/demand-driven cloning

12.2.6 Precision Comparison with Andersen

| Benchmark | DSA May-Alias % | Andersen May-Alias % | Winner |
|------------|-----------------|----------------------|----------|
| 181.mcf | 1.8% | 20.5% | DSA |
| 175.vpr | 3.2% | 18.1% | DSA |
| 186.crafty | 8.2% | 23.4% | DSA |
| 300.twolf | 4.1% | 15.8% | DSA |
| 197.parser | 22.1% | 18.5% | Andersen |
| 255.vortex | 19.8% | 16.2% | Andersen |

Key Finding:

- **DSA better when:** wrapper functions, data structure libraries
- **Andersen better when:** no wrappers, direct malloc usage
- Context-sensitivity + heap cloning compensates for unification

Hybrid Strategy:

- Use DSA for initial fast analysis
- Demand-driven refinement for specific queries where DSA is imprecise

13. On-the-Fly Call Graph Construction

Foundational Reference: Qilin

[Qilin22] D. He, J. Lu, J. Xue. *Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis*. ECOOP 2022.

The Chicken-and-Egg Problem

For OOP languages with virtual dispatch:

- Call graph construction needs pointer analysis (to resolve receivers)
- Pointer analysis needs call graph (to know which methods to analyze)

Solution: Solve them TOGETHER, on-the-fly.

Result: Qilin (single-threaded) is 2.4x faster than Doop (8-threaded) while achieving identical precision.

The Qilin framework parameterizes pointer analysis over three orthogonal dimensions, enabling fine-grained control over the precision/performance trade-off. The `context_constructor` determines *how* calling contexts are represented: call-site strings (*k*-CFA), receiver objects (object-sensitivity), or receiver types (type-sensitivity). The `context_selector` determines *which* methods receive context-sensitive treatment—ZIPPER (Section 13.2) uses this to selectively apply context-sensitivity only where it improves precision. The `heap_abstractor` controls allocation-site naming granularity.

Parameterized Context-Sensitivity

```
(* =====
ON-THE-FLY CALL GRAPH + POINTER ANALYSIS
Source: Qilin (He, Lu, Xue 2022)
Three parameters control precision/performance tradeoff:
1. Context Constructor (Cons): HOW to build contexts
2. Context Selector (Sel): WHEN to apply context-sensitivity
3. Heap Abtractor (HeapAbs): HOW to abstract allocation sites
===== *)

module BrrrMachine.PointerAnalysis.OnTheFly

type context_constructor =
| ConsInsens      (* No context - fastest, least precise *)
| ConsCallsite    (* k-CFA: use call sites as context *)
| ConsObject      (* Object-sensitivity: use receiver objects *)
| ConsType        (* Type-sensitivity: use receiver types *)
| ConsHybrid      (* Selective based on heuristics *)

type context_selector =
| SelUniform : k:nat → hk:nat → context_selector
| SelSelective : cs_methods:set method_id → context_selector
| SelPartial : cs_vars:set var_id → context_selector

type heap_abtractor =
| HeapAllocationSite (* Standard: one abstract loc per new *)
| HeapTypeConsistent (* Merge by type *)
| HeapHeuristic      (* Exclude certain types like String, StringBuilder *)
```

These type definitions encode the configuration space for context-sensitive analysis. The `SelSelective` variant enables ZIPPER-style selective context-sensitivity, where only methods in `cs_methods` receive context-sensitive treatment. The `SelPartial` variant provides even finer granularity at the variable level.

The core algorithm below interleaves call graph construction with points-to propagation. The key insight is maintaining *old* and *new* partitions of both reachable methods and points-to facts. When a virtual call's receiver type becomes more precise, new callees are discovered and added to `new_part`. The algorithm continues until both partitions stabilize (fixpoint).

Core Algorithm: Interleaved CG + PTS Construction

```
val solve_on_the_fly :
  cpg →
  cons:context_constructor →
  sel:context_selector →
  heap:heap_abtractor →
  (call_graph * pts_solution)
let solve_on_the_fly cpg cons sel heap =
  let state = {
    pts = Map.empty;
    call_graph = { old_part = Set.empty; new_part = Set.empty };
    reachable = { old_part = Set.empty;
                  new_part = Set.singleton (entry_method cpg, empty_ctx) };
    worklist = [];
  } in

  let rec process state =
    (* Step 1: Process newly reachable methods *)
    let state = process_new_reachable state cons sel heap in
    (* Step 2: Propagate points-to through worklist *)
    match pop_worklist state with
    | None → state (* Fixpoint reached *)
    | Some ((v, ctx), state') →
      let pts_new = get_new_pts state' v ctx in
      (* Step 3: Propagate through direct constraints *)
```

```

let state' = propagate_direct state' v ctx pts_new in
(* Step 4: Propagate through indirect constraints *)
let state' = propagate_indirect state' v ctx pts_new in
(* Step 5: Resolve virtual calls based on new receiver types *)
let state' = resolve_virtual_calls state' v ctx pts_new cons sel in
(* Step 6: Flush and continue *)
let state' = flush_pts state' v ctx in
process state'
in
let final = process state in
(extract_call_graph final, extract_pts final)

```

Theorem 13.1 (Variable-Level CS Subtlety). *Source: Qilin Theorem 1*

Under method-level CS: When method m is analyzed under new context c , $PTS(v, Sel(v, c))_{old} = \emptyset$ always holds.

Under variable-level CS: $PTS(v, Sel(v, c))_{old} \neq \emptyset$ may hold because $Sel(v, c) = Sel(v, c')$ can be true for $c \neq c'$.

Implication: Must propagate through NEW edges using OLD points-to facts.

13.1 When to Use Each Approach

DECISION TREE: Phased vs On-the-Fly

Is the language...

```

|
+-- Purely procedural (C, Fortran)?
|   -> Use PHASED (Section 3.1.7) - simpler, sufficient
|
+-- Has virtual dispatch (Java, Python, C++, JS)?
|   -> Use ON-THE-FLY (Section 5.3) - REQUIRED for soundness
|
+-- Mixed (Rust, Go)?
    -> Use ON-THE-FLY for trait objects / interfaces
        Phased is OK for static dispatch portions

```

13.2 Selective Context Sensitivity (ZIPPER)

Foundational Reference: ZIPPER

[ZIPPER20] Y. Li, T. Tan, A. Møller, Y. Smaragdakis. *A Principled Approach to Selective Context Sensitivity for Pointer Analysis*. OOPSLA 2020.

Context sensitivity dramatically improves pointer analysis precision but is expensive. The fundamental insight from ZIPPER is that **only ~38% of methods actually benefit from context sensitivity**. The remaining 62% receive context-sensitive treatment that consumes analysis time without improving results.

13.2.1 The Uniform Context Sensitivity Problem

OBSERVATION: Applying context sensitivity uniformly is wasteful.

Traditional approaches:

- 2-object-sensitive: ALL methods get 2-level context
- 2-callsite-sensitive: ALL methods get 2-level call string

Reality:

- ~62% of methods: CS provides NO precision benefit

- ~38% of methods: CS is precision-critical
- Wasted analysis time on the 62% can be 3-25x the useful work

13.2.2 The Three Precision-Loss Patterns

ZIPPER identifies **why** context-insensitive analysis loses precision through three observable value-flow patterns:

Definition 13.2 (Pattern 1: Direct Flow). Object O enters via IN method parameter, flows through assignments and field operations, exits via OUT method return of SAME class.

```
class Person {
  String name;
  void setName(String nm) { this.name = nm; } // In method
  String getName() { return this.name; }      // Out method
}
```

Without CS: Objects from different `setName()` calls merge in `nm` parameter.

With CS: Each call site distinguished, precise points-to.

Definition 13.3 (Pattern 2: Wrapped Flow). Object O enters via IN method, gets stored in wrapper object W , wrapper W flows out via OUT method return.

```
void add(Object el) { this.elem = el; } // Object wrapped
Iterator iterator() { return new Iterator(elem); } // Wrapper flows out
```

Imprecision manifests when wrapper is later accessed.

Definition 13.4 (Pattern 3: Unwrapped Flow). Object O (a carrier) enters via IN method, contents are loaded from O , loaded contents flow out via OUT method return.

```
SyncBox(Box box) { this.box = box; }
Object getItem() {
  Box b = this.box; // Carrier loaded
  return b.getItem(); // Unwrapped content flows out
}
```

13.2.3 The ZIPPER Algorithm

ALGORITHM: Zipper(program, OFG)

INPUT: program - Source program

OFG - Object Flow Graph from CI pre-analysis

1. `PCM <- {}` // Precision-Critical Methods
2. FOR EACH class c in program:
3. `PFG_c <- build_pfg(c, OFG)` // Add wrap/unwrap edges
- 4.
5. // Backward reachability from Out method returns
6. `FlowNodes <- {}`
7. FOR EACH Out method m of c :
8. FOR EACH return var r of m :
9. `FlowNodes <- FlowNodes U backward_reach(PFG_c, r)`
- 10.

```

11. // Extract methods containing flow nodes
12. FOR EACH node n in FlowNodes:
13.   PCM <- PCM U {method_of(n)}

14. RETURN PCM

```

COMPLEXITY: $O(|\text{classes}| * |\text{OFG}|)$

In practice: ~11 seconds for large Java programs

13.2.4 Empirical Results

| Configuration | Precision Preserved | Speedup | PCM Coverage |
|-----------------------------------|---------------------|------------------------|--------------|
| Zipper-2obj | 98.8% | 3.4x (avg), 9.4x (max) | ~38% methods |
| Zipper _e -2obj (PV=5%) | 94.7% | 25.5x (avg), 88x (max) | ~14% methods |

Surprising Result: For 5 programs, Zipper_e-guided CS is *faster* than context-insensitive analysis while being *more precise*. This occurs because the reduced method set enables better cache locality.

13.3 Python-Specific Call Graph via Type Inference

Foundational Reference: JARVIS

[JARVIS23] M. Huang et al. *JARVIS: Scalable and Precise Application-Centered Call Graph Construction for Python*. ICSE 2023.

Why Qilin Is Insufficient for Python

Qilin (Section 5.3) is designed for **class-based OOP** with:

- Explicit type declarations (Java, C++)
- Receiver-based dispatch (`obj.method(args)`)
- Static method signatures

Python requires **different handling** for:

- Duck typing (no static type at receiver)
- Module-level functions (no receiver)
- First-class functions and closures
- Magic methods (`__getattr__`, `__call__`, descriptors)
- MRO (Method Resolution Order) via C3 linearization
- Decorator wrapping

Solution: Function Type Graph (FTG) — per-function type inference with flow-sensitive strong updates.

Results: 84% higher precision than flow-insensitive (PyCG), 67% faster on exhaustive whole-program analysis. Handles 200k+ LOC codebases (PyCG runs OOM).

The Function Type Graph (FTG) represents Python’s type inference results. Unlike Java where types are declared, Python types must be inferred from usage patterns. The `python_type_element` discriminates between modules, classes, functions, variables, and builtins—each has different scoping and attribute lookup rules. The `python_namespace` captures the name-to-type bindings visible at each scope level.

The `ftg_relation` type encodes three kinds of type facts: `FTGPointsTo` for pointer-like relationships, `FTGTypeOf` for expression types, and `FTGFieldOf` for attribute access. These relations form a graph that can be queried to resolve method calls.

Function Type Graph Structure

```
module BrrrMachine.Python.FTG

type python_type_element =
| PTEModule : name:string → is_external:bool → python_type_element
| PTEClass : name:string → module:string → python_type_element
| PTEFunction : name:string → module:string → python_type_element
| PTEVariable : name:string → scope:string → python_type_element
| PTEBuiltin : name:string → python_type_element

type python_namespace = list (string * python_type_element)

type python_type = {
  element : python_type_element;
  namespace : python_namespace;
  name : string;
}

type ftg_relation =
| FTGPointsTo : src:python_type → dst:python_type → ftg_relation
| FTGTypeOf : expr:ir_expr → ty:python_type → ftg_relation
| FTGFieldOf : base:python_type → field:string → value:python_type → ftg_relation

type function_type_graph = {
  types : set python_type;
  exprs : set ir_expr;
  relations : set ftg_relation;
}
```

13.3.1 C3 Linearization for Python MRO

Python’s multiple inheritance uses the C3 linearization algorithm to compute the Method Resolution Order (MRO). This determines the order in which base classes are searched when resolving a method call. The algorithm must satisfy three properties: (1) children come before parents, (2) the order respects the left-to-right ordering in the class definition, and (3) the result is consistent across all classes. If no linearization satisfying these properties exists, Python raises a `TypeError` at class definition time.

The `c3_merge` function implements the merge step: it finds a class that appears at the head of some list but not in the tail of any other list (the “good head” property). The recursion terminates when all lists are empty. A `None` result indicates an inconsistent hierarchy that Python would reject.

C3 Linearization

(Python uses C3 linearization for method lookup in multiple inheritance. *)*

Example:

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass # MRO: D → B → C → A → object *)
```

```
val c3_merge : list (list class_id) → option (list class_id)
let rec c3_merge seqs =
  let non_empty = List.filter (fun l → not (List.is_empty l)) seqs in
  if List.is_empty non_empty then Some []
  else
    let find_candidate () =
      List.find_opt (fun seq →
        let head = List.hd seq in
        List.for_all (fun other_seq →
          not (List.mem head (List.tl other_seq))
        ) non_empty
      ) non_empty
    in
```

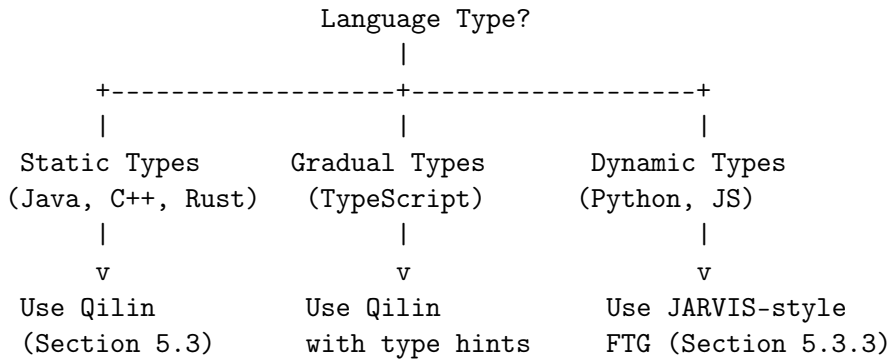
```

in
match find_candidate () with
| None → None  (* C3 merge failure - inconsistent hierarchy *)
| Some winner_seq →
    let head = List.hd winner_seq in
    let remaining = List.map (fun seq →
        List.filter (fun c → c ≠ head) seq
    ) non_empty in
    match c3_merge remaining with
    | None → None
    | Some rest → Some (head :: rest)

val compute_mro : class_summary → class_id → list class_id
let compute_mro summ cls =
    let parents = get_direct_parents summ cls in
    if List.is_empty parents then [cls; "object"]
    else
        let parent_mros = List.map (compute_mro summ) parents in
        match c3_merge (parent_mros @ [parents]) with
        | Some mro → cls :: List.concat parent_mros  (* Fallback *)
        | None → cls :: List.concat parent_mros

```

13.3.2 Call Graph Algorithm Selection



SUMMARY:

- Java/C++: Qilin + ZIPPER (object-sensitive, selective CS)
- Rust: Qilin + Stack Filtering (Section 12.22)
- TypeScript: Qilin with type annotations
- Python: JARVIS FTG (flow-sensitive type inference)
- JavaScript: Hybrid FTG + prototype chain analysis
- Mixed: Use appropriate algorithm per module

14. Shape Analysis via Three-Valued Logic

Foundational Reference: TVLA

[SRW02] M. Sagiv, T. Reps, R. Wilhelm. *Parametric Shape Analysis via 3-Valued Logic*. TOPLAS 2002.

Shape analysis determines structural properties of heap data: lists, trees, DAGs, cycles. Unlike pointer analysis which tracks aliasing, shape analysis tracks *data structure invariants*.

14.1 Three-Valued Foundation

Definition 14.1 (Three-Valued Logic). Two-valued logic is **too imprecise** for heap analysis. Consider: after $x = y \rightarrow \text{next}$ in a list traversal, x might point to a summary node (represents multiple cells). Does $x \rightarrow \text{next} = y$? Unknown! Neither yes nor no is safe.

Three-Valued Logic:

- 1 = definitely true
- 0 = definitely false
- $\frac{1}{2}$ = unknown/indefinite

Information ordering: $\frac{1}{2} \sqsubseteq 0$ and $\frac{1}{2} \sqsubseteq 1$ (Unknown is *less* informative than definite).

14.2 Shape Structures

Shape structures represent abstract heaps using three-valued predicates. The `universe` contains abstract locations, some of which are *summary nodes* representing multiple concrete heap cells. The `predicates` map assigns three-valued truth values to predicate applications—for example, `n(u,v)` might be 1 (definitely an edge), 0 (definitely no edge), or $\frac{1}{2}$ (unknown).

The predicates below are standard for linked data structure analysis. Core predicates like `x` (variable points here) and `n` (next-pointer edge) describe the heap graph structure. Instrumentation predicates like `r_x` (reachable from `x`), `c` (on a cycle), and `is` (shared/aliased) are derived predicates that dramatically improve precision by encoding properties that would otherwise be lost to abstraction.

Shape Structure Types

```
type shape_structure = {
  universe : set abstract_loc;
  predicates : map pred_name (list loc → tv);
  is_summary : loc → tv;
}

(* Core predicates *)
val x : loc → tv          (* Variable x points here *)
val n : loc → loc → tv    (* n(u,v) = u.next points to v *)
val sm : loc → tv         (* Summary: represents multiple concrete *)

(* Derived instrumentation predicates - CRITICAL for precision *)
val r_x : loc → tv        (* Reachable from x: r_x(v) = TC(x,v) via n *)
val c : loc → tv          (* On cycle: c(v) = ∃ u. n*(v,u) ∧ n(u,v) *)
val is : loc → tv         (* Shared: is(v) = ∃ u,u'. u!=u' ∧ n(u,v) ∧ n(u',v) *)
```

14.3 Embedding Theorem

Theorem 14.2 (Embedding Theorem). *Source: [SRW02], Theorem 3.7*

If $S \sqsubseteq^f S'$ (S embeds in S' via f), then for every formula φ and assignment Z :

$$\llbracket \varphi \rrbracket_S^3(Z) \sqsubseteq \llbracket \varphi \rrbracket_{S'}^3(f \circ Z)$$

Meaning: If concrete heap satisfies φ (value 1), abstract shape doesn't refute it (value $\neq 0$).

14.4 Focus-Transform-Coerce Algorithm

ABSTRACT INTERPRETATION STEP:

1. FOCUS: Make relevant predicates definite
 - If $n(x, _) = 1/2$, split structure into cases
 - "Materialize" concrete nodes from summaries
 - Essential for loop analysis precision
2. TRANSFORM: Apply statement semantics
 - Update predicates according to operation
 - Standard abstract interpretation

3. COERCE: Apply compatibility constraints
 - Propagate implications: $\phi_1 \Rightarrow \phi_2$
 - Detect inconsistencies (return bottom)
 - Sharpen indefinite predicates

LOOP TERMINATION:

- Bounded structures have finite domain
- Canonical abstraction: merge by predicate values
- No explicit widening needed!

14.5 Integration with Brrr-Machine

Shape Analysis Track:

1. Build initial shape from CPG:
 - Stack/global vars \rightarrow individual nodes
 - Allocations \rightarrow summary nodes ($sm = \frac{1}{2}$)
2. Run focus-transform-coerce at each CPG node
3. Extract properties:
 - $list(x, null) = \text{valid list from } x$
 - $tree(x) = \text{valid tree rooted at } x$
 - $shared(v) = 0 \implies \text{unique ownership}$
 - $cycle(v) = 0 \implies \text{acyclic}$

Detected Bugs:

- Use-after-free: $r_x(v) = 1$ but $freed(v) = 1$
- Dangling pointer: $n(u, v) = 1$ but $freed(v) = 1$
- Memory leak: $\neg(\exists x. r_x(v) = 1)$ but $freed(v) = 0$
- List corruption: was list, now has cycle

15. Symbolic Heap Shape Analysis

Foundational Reference: Distefano, O'Hearn, Yang 2006

[DOY06] D. Distefano, P. O'Hearn, H. Yang. *A Local Shape Analysis Based on Separation Logic*. TACAS 2006.

Symbolic Heaps vs TVLA

An **alternative** to TVLA (Section 5.4) for shape analysis based on separation logic.

Key Advantages:

- Natural integration with frame rule (Section 7.4)
- Direct memory leak detection via “junk” predicate
- Simpler abstraction via canonicalization rules

Trade-off vs TVLA:

- **TVLA:** better for materialization/summarization
- **Symbolic Heaps:** better for compositional (per-procedure) analysis

Both approaches are SOUND; choice depends on analysis goals.

15.1 Symbolic Heap Structure

Definition 15.1 (Symbolic Heap). A symbolic heap has form $\Pi \vdash \Sigma$ where:

- Π (pure part): equalities and disequalities between variables
- Σ (spatial part): heap predicates in separation logic

This forms a FINITE abstract domain when combined with canonicalization, bounded by $2^{129n^2+18n+2}$ for n program variables (Theorem 14).

The F* types below encode symbolic heaps in separation logic style. Variables come in three flavors: program variables (SHProgVar), existentially-quantified primed variables (SHPrimedVar) introduced during bi-abduction, and the distinguished null value (SHNil). The pure formula captures equalities and disequalities between variables, while the spatial formula describes heap structure.

The key spatial connectives are: SPointsTo for a single heap cell, SListSeg for an inductively-defined list segment, SJunk for unreachable garbage (memory leaks), and SSep for the separating conjunction. The `abstract_state` is a *set* of symbolic heaps, representing disjunctive information from different execution paths.

Symbolic Heap Types

```
module BrrrMachine.ShapeAnalysis.SymbolicHeap

type sh_var =
| SHProgVar : v:var_id → sh_var
| SHPrimedVar : v:nat → sh_var      (* Existentially quantified x' *)
| SHNil : sh_var

type pure_formula =
| PEq : sh_var → sh_var → pure_formula
| PNeq : sh_var → sh_var → pure_formula
| PAnd : pure_formula → pure_formula → pure_formula
| PTrue

type spatial_formula =
| SEmpty                                     (* emp *)
| SPointsTo : src:sh_var → field:string → dst:sh_var → spatial_formula
| SListSeg : src:sh_var → dst:sh_var → spatial_formula (* ls(x, y) *)
| SJunk                                     (* junk - unreachable garbage *)
| SSep : spatial_formula → spatial_formula → spatial_formula (* * *)

type symbolic_heap = {
  pure : pure_formula;
  spatial : spatial_formula;
}

type abstract_state = set symbolic_heap
```

15.2 Heap Predicates

Definition 15.2 (Points-To Predicate). The cell at address $(x + \text{offset}(f))$ contains value y . This is the **atomic** spatial predicate.

Critical: Points-to is EXCLUSIVE — it *owns* the cell. Two points-to on same address cannot be separated.

Definition 15.3 (List Segment Predicate). $ls(x, y)$: There is a well-formed singly-linked list from x to y .

Defined **inductively**:

- $ls(x, x) = \text{emp}$ (empty list — base case)
- $ls(x, y) = \exists z. x.\text{next} \mapsto z * ls(z, y)$ (cons cell)

Critical: The list segment *owns* all intermediate nodes.

Definition 15.4 (Junk Predicate). Represents heap cells that are UNREACHABLE from program variables. This is garbage — a **memory leak indicator**.

Critical Property: $\text{junk} * \text{junk} = \text{junk}$ (idempotent under separation)

This allows UNBOUNDED accumulation of leaks without state explosion.

15.3 Canonicalization as Widening

Theorem 15.5 (Termination Guarantee). *Source: [DOY06], Theorem 14*

The number of CANONICAL symbolic heaps is bounded by:

$$2^{129n^2+18n+2}$$

where n = number of program variables.

This is FINITE, ensuring:

1. Canonicalization always terminates
2. The abstract domain is finite
3. Fixpoint iteration terminates without explicit widening

CANONICALIZATION RULES:

```

SubstEq:   x' = E => substitute E for x' everywhere
Garbage1:  P(x', E) where x' unreachable => junk
Garbage2:  P1(x', y') * P2(y', x') cycle => junk
Abs1:      P1(E, x') * P2(x', F) => ls(E, nil) when Pi |- F = nil
Abs2:      P1(E, x') * P2(x', F) * P3(G, H) => ls(E, F) * P3(G, H)
           (when F != G)

```

15.4 Memory Leak Detection via Junk

The **SJunk** predicate makes leak detection trivial: after canonicalization, any heap cells that become unreachable from program variables are folded into **junk**. The idempotent property ($\text{junk} * \text{junk} = \text{junk}$) means the abstract state remains bounded even with unbounded leaks.

The leak classification below distinguishes three cases: **DefiniteLeak** when all abstract states contain junk (the leak is unavoidable), **PossibleLeak** when some but not all states contain junk (path-dependent), and **NoLeak** when no states contain junk. This maps directly to the manifest/latent bug classification in incorrectness logic (Section 12.3).

```

Leak Detection
type leak_result =
  | NoLeak
  | PossibleLeak
  | DefiniteLeak of leaked_cells : nat

val detect_leaks : abstract_state → leak_result
let detect_leaks final_states =
  let has_junk sh = contains_junk sh.spatial in
  let all_have_junk = Set.for_all has_junk final_states in
  let some_have_junk = Set.∃ has_junk final_states in
  if all_have_junk then
    DefiniteLeak (min_junk_size final_states)
  else if some_have_junk then
    PossibleLeak
  else
    NoLeak

```

Remark 15.6 (Integration with Manifest Bug Classification). Cross-reference: Section 12.3 (Manifest/Latent bug classification)

DefiniteLeak is a MANIFEST bug because:

1. Junk appears on ALL execution paths
2. The leak is PROVABLE — not a false positive
3. It satisfies the “reachability condition”

PossibleLeak is a LATENT bug because:

1. Junk appears on SOME but not all paths
2. The leak depends on which path is taken
3. Caller context determines if leak manifests

15.5 Frame Rule for Compositional Analysis

Theorem 15.7 (Abstract Frame Rule). *Source: [DOY06], Theorem 11*

*If $\{P\} C \{Q\}$ and C doesn't modify variables in R , then $\{P * R\} C \{Q * R\}$.*

For all $X, Y \in \mathcal{P}(\text{CSH})$, if $\text{Vars}(Y) \cap \text{Mod}(C) = \emptyset$ then:

$$\gamma(\mathcal{A}[c](X * Y)) \subseteq \gamma((\mathcal{A}[c]X) * Y)$$

This enables COMPOSITIONAL interprocedural analysis:

- Analyze each procedure in isolation on its FOOTPRINT
- Combine via frame rule
- No need to re-analyze callee for each calling context

Definition 15.8 (Procedure Footprint). A function's FOOTPRINT is the MINIMAL heap needed to execute safely. By Theorem 12 of [DOY06], this footprint soundly approximates the full semantics:

$$\text{foot}(c)\sigma = \begin{cases} \mathcal{A}[c]\{\sigma\} & \text{if } \text{onlyaccesses}(c, \sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The footprint representation below captures a procedure's memory behavior independently of its calling context. The `input_footprint` is the minimal heap the procedure needs to execute (discovered via bi-abduction), and `output_footprints` are the possible resulting heaps. The `modified_vars` track which variables may be changed.

The `apply_footprint` function shows how footprints enable compositional analysis: at a call site, we match the caller's state against the callee's footprint, extracting the *frame*—the part of the heap untouched by the call. After the call, we combine the callee's output with the frame. This is the abstract frame rule in action.

Footprint Computation and Application

```
type procedure_footprint = {
  input_footprint : symbolic_heap;
  output_footprints : set symbolic_heap;
  modified_vars : set var_id;
}

val compute_footprint : cpg → func_id → procedure_footprint
let compute_footprint cpg func =
  let init = { pure = PTrue; spatial = SEmpty } in
  let results = symbolic_exec_function cpg func init in
  let minimal_input = collect_anti_frames results in
  let outputs = Set.map (fun r → r.post) results in
  { input_footprint = canonicalize minimal_input;
    output_footprints = Set.map canonicalize outputs;
    modified_vars = compute_mod cpg func }

val apply_footprint :
  caller_state : symbolic_heap →
  callee_footprint : procedure_footprint →
  call_site : node_id →
  set symbolic_heap
let apply_footprint caller_state footprint call_site =
  match find_matching_substate caller_state footprint.input_footprint with
  | Some (matching, frame) →
    Set.map (fun output →
      canonicalize { output with spatial = SSep output.spatial frame })
```

```

    ) footprint.output_footprints
  | None →
    Set.singleton { pure = PTrue;
                    spatial = SSep SJunk (error_marker "footprint_mismatch") }

```

15.6 Symbolic Execution for Symbolic Heaps

Symbolic execution on symbolic heaps discovers preconditions “on demand” via bi-abduction. When a statement requires a heap cell that is not currently available, the analysis does not fail—instead, it *abduces* (infers) the missing cell as part of the required precondition. This is captured in the `anti_frame` field of `exec_result`.

The code below handles key memory operations. For loads (`SLoad`), if the required cell is available, we extract its value; if not, we create a fresh existential and record the needed cell in `anti_frame`. For stores (`SStore`), we update the heap or abduce the target cell. For allocation (`SAlloc`), we extend the heap with a fresh cell. For free (`SFree`), we remove the cell. The `exit` field tracks whether the operation succeeded (`Ok`) or errored (`Er`).

Symbolic Execution with Bi-Abduction

```

(* Execute commands on symbolic heaps, discovering preconditions via
   bi-abduction when heap access requires unavailable cells. *)

type exec_result = {
  post : symbolic_heap;
  anti_frame : spatial_formula; (* Discovered precondition *)
  exit : exit_condition;
}

val symbolic_exec_stmt : symbolic_heap → ir_stmt → set exec_result
let symbolic_exec_stmt sh stmt =
  match stmt with
  (* LOAD: x := ptr→field *)
  | SLoad x ptr field →
    (match expose_points_to sh ptr field with
     | Found (sh', v) →
       Set.singleton {
         post = assign_var sh' x v;
         anti_frame = SEmpty;
         exit = Ok }
     | NeedCell →
       let fresh_v = fresh_primed_var () in
       Set.singleton {
         post = assign_var sh x fresh_v;
         anti_frame = SPointsTo (var_to_sh ptr) field fresh_v;
         exit = Ok }
     | Contradiction →
       Set.singleton { post = sh; anti_frame = SEmpty; exit = Er })
  (* STORE: ptr→field := v *)
  | SStore ptr field v →
    (match expose_points_to sh ptr field with
     | Found (sh', _) →
       Set.singleton {
         post = update_points_to sh' ptr field v;
         anti_frame = SEmpty;
         exit = Ok }
     | NeedCell →
       let fresh_v = fresh_primed_var () in
       Set.singleton {
         post = update_points_to sh ptr field v;
         anti_frame = SPointsTo (var_to_sh ptr) field fresh_v;
         exit = Ok }

```

```

| Contradiction →
  Set.singleton { post = sh; anti_frame = SEmpty; exit = Er })

(* ALLOC: x := malloc(size) *)
| SAlloc x size →
  let fresh_val = fresh_primed_var () in
  Set.singleton {
    post = { sh with
      spatial = SSep sh.spatial (SPointsTo (SHProgVar x) "data" fresh_val) };
    anti_frame = SEmpty;
    exit = Ok }

(* FREE: free(ptr) *)
| SFree ptr →
  (match remove_points_to sh ptr with
  | Some sh' →
    Set.singleton { post = sh'; anti_frame = SEmpty; exit = Ok }
  | None →
    Set.singleton { post = sh; anti_frame = SEmpty; exit = Er })

(* ASSIGN: x := y -- may cause LEAK *)
| SAssign x y →
  let sh' = update_var sh x y in
  Set.singleton {
    post = canonicalize sh'; (* Detect leaks via Garbage rules *)
    anti_frame = SEmpty;
    exit = Ok }

```

15.7 Comparison: TVLA vs Symbolic Heaps

| Aspect | TVLA (Section 5.4) | Symbolic Heaps (Section 5.5) |
|----------------|-------------------------------------|--------------------------------|
| Foundation | Three-valued logic | Separation logic |
| Abstraction | Instrumentation predicates | Canonicalization rules |
| Unknown | $\frac{1}{2}$ with Kleene semantics | Disjunction of heaps |
| Composition | Focus-transform-coerce | Frame rule |
| Leak Detection | Separate predicate needed | Built-in (junk) |
| Termination | Canonical abstraction | Theorem 14 bound |
| Best For | Materialization, complex shapes | Local reasoning, per-procedure |

Recommendation:

- Use TVLA when analyzing complex data structures where materialization of summary nodes is critical
- Use Symbolic Heaps when doing compositional analysis where frame rule enables procedure summaries
- Hybrid approach: Use symbolic heaps for interprocedural, TVLA for intraprocedural shape reasoning

16. Sparse Value-Flow Analysis (SVF)

Foundational References

- [SYX12] Y. Sui, D. Ye, J. Xue. *Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis*. ISSTA 2012.
- [SX16] Y. Sui, J. Xue. *SVF: Interprocedural Static Value-Flow Analysis in LLVM*. CC 2016.
- [CWZ96] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, P. Tu. *A New Algorithm for Partial Redundancy Elimination Based on SSA Form*. CC 1996.

SVF: Alternative to IFDS for Source-Sink Problems

SVF is a **complementary** technique to IFDS (Section 4.1), not an optimization. Both share CFL-reachability for context-sensitivity, but differ in:

- **Representation:** SVFG vs Exploded Supergraph
- **Prerequisites:** Requires pointer analysis vs self-contained
- **Applicability:** Source-sink problems vs general dataflow

Key Insight: SPARSITY — most statements affect few memory locations. By pre-computing def-use chains via Memory SSA, analysis propagates only along actual value flows, not all control-flow edges.

Pointer Analysis Input: SVF **REQUIRES** pointer analysis (*pts*) for Memory SSA construction. For large C/C++ codebases ($> 100K$ LOC):

- Use DSA (Section 5.2.5) for fast $\mathcal{O}n \cdot \alpha(n)$ pointer analysis
- DSA's heap cloning provides **better** precision than allocation-site only

For smaller codebases: Use Andersen (Section 5.1) for maximum precision.

16.1 Motivation: Dense vs Sparse Dataflow

THE DENSITY PROBLEM:

Traditional IFDS (Section 4.1) propagates dataflow facts along EVERY control-flow edge. For N statements and D dataflow facts, this is $\mathcal{O}(E * D^3)$.

For source-sink problems (memory leaks, use-after-free, taint-to-sink):

- We only care about reachability from SOURCE to SINK
- Most statements don't affect the tracked values
- Dense iteration wastes work on irrelevant paths

SVF INSIGHT:

Pre-compute def-use chains, then analyze ONLY along those chains.

This leverages the SPARSITY of actual value flows.

COMPARISON:

Dense (IFDS): Propagate facts at EVERY CFG edge

Sparse (SVF): Propagate facts only along DEF-USE edges

When to use each:

- IFDS: When you need ALL facts at each program point (reaching defs)
- SVF: When you need source-sink REACHABILITY (leak detection)

16.2 Memory SSA

Memory SSA extends classical SSA to handle indirect memory accesses using μ/χ annotations from [CWZ96].

Definition 16.1 (μ Annotation). Represents a potential USE of an address-taken variable. For load statement $x = *y$, annotate with $\mu(o)$ for each $o \in pts(y)$.

Definition 16.2 (χ Annotation). Represents a potential DEF (and use) of an address-taken variable. For store statement $*x = y$, annotate with $o = \chi(o)$ for each $o \in pts(x)$.

Definition 16.3 (Strong vs Weak Update). **Strong Update:** If ptr uniquely points to o (singleton points-to set and o is a concrete location), then new value KILLS old contents.

Weak Update: If ptr may point to multiple locations or o is abstract (summary node), new value must JOIN with old value.

This distinction is **critical** for precision: strong update enables KILL, improving precision.

ANNOTATION EXAMPLES:

1. LOAD: $x = *y$ (where $\text{pts}(y) = \{o1, o2\}$)

Before: $x = *y$
 After: $x = *y \text{ } [\mu(o1_3), \mu(o2_5)]$

Meaning: x 's value MAY come from $o1$ version 3 or $o2$ version 5

2. STORE: $*x = y$ (where $\text{pts}(x) = \{o\}$, singleton)

Before: $*x = y$
 After: $*x = y \text{ } [o_4 = \text{chi}(o_3)]$ (STRONG update)

Meaning: o gets new version 4, old value killed

3. STORE: $*x = y$ (where $\text{pts}(x) = \{o1, o2\}$, multiple)

Before: $*x = y$
 After: $*x = y \text{ } [o1_4 = \text{chi}(o1_3), o2_6 = \text{chi}(o2_5)]$ (WEAK update)

Meaning: Both regions updated but old values preserved in join

16.3 SVFG Construction

The Sparse Value-Flow Graph (SVFG) is constructed from Memory SSA by connecting def-use pairs. The F* types below encode SVFG nodes and edges. Nodes come in several flavors: `CopyNode` and `PhiNode` handle direct value-flow; `LoadNode` and `StoreNode` handle indirect value-flow via memory with μ/χ annotations; `DParamNode`/`IParamNode` and `DRetNode`/`IRetNode` handle interprocedural value-flow for direct and indirect parameters/returns; `SourceNode` and `SinkNode` mark allocation and deallocation sites for leak detection.

Edge labels distinguish between direct flow (register assignments), indirect flow (through memory), and interprocedural edges (calls and returns with callsite/callee information for context-sensitivity).

SVFG Types

```
module BrrrMachine.SVFG

type svfg_node =
  (* Direct value-flow nodes *)
  | CopyNode : def_site:node_id → dst:var_id → src:var_id → svfg_node
  | PhiNode : def_site:node_id → dst:var_id → srcs:list var_id → svfg_node

  (* Indirect value-flow nodes *)
  | LoadNode : def_site:node_id → dst:var_id → ptr:var_id →
    mu:mu_annotation → svfg_node
  | StoreNode : def_site:node_id → ptr:var_id → val_:var_id →
    chi:chi_annotation → svfg_node

  (* Interprocedural nodes *)
  | DParamNode : func:func_id → param:var_id → svfg_node
  | IParamNode : func:func_id → region:versioned_region → svfg_node
  | DRetNode : callsite:node_id → result:var_id → svfg_node
  | IRetNode : callsite:node_id → region:versioned_region → svfg_node
```

```

(* Source/sink markers *)
| SourceNode : alloc_site:node_id → svfg_node
| SinkNode : free_site:node_id → svfg_node

type svfg_edge_label =
| DirectFlow
| IndirectFlow
| CallEdge : callsite:node_id → callee:func_id → svfg_edge_label
| ReturnEdge : callsite:node_id → callee:func_id → svfg_edge_label

type svfg = {
  nodes : map svfg_node_id svfg_node;
  edges : list svfg_edge;
  sources : set svfg_node_id;
  sinks : set svfg_node_id;
}

```

16.4 Context-Sensitivity via CFL-Reachability

Context-sensitivity uses the **same** approach as IFDS (Section 4.2):

CFL FOR MATCHED CALLS AND RETURNS:

Edge labels form a Dyck language (matched parentheses):

- Call edge: (c_g (open paren for callsite c calling g)
- Return edge:)c_g (close paren)

Valid interprocedural path: balanced parentheses

Grammar:

$S \rightarrow S S \mid (i S) i \mid \text{epsilon} \quad \text{for each callsite } i$

This is IDENTICAL to IFDS context-sensitivity.

The difference is WHAT is tracked (value-flows vs dataflow facts).

16.5 IFDS vs SVF Comparison

| Aspect | IFDS ([Reps95]) | SVF ([SX16]) |
|---------------------|--------------------------------------|---|
| Problem Class | Distributive dataflow | Source-sink value-flow |
| Domain | Finite set D of facts | Def-use chains for memory |
| Graph Structure | Exploded supergraph ($N \times D$) | Sparse VFG (N nodes) |
| Context Sensitivity | CFL-reachability | CFL-reachability |
| Distributivity | REQUIRED | Not required |
| Pointer Handling | Requires separate pts | Integrated via Memory SSA |
| Pre-analysis | None required | Pointer analysis REQUIRED |
| Time Complexity | $\mathcal{O}E \cdot D^3$ | $\mathcal{O}\text{Andersen} + \mathcal{O}N \cdot R^2$ |
| Best Use Cases | Taint, reaching defs, uninit | Leak detection, UAF, DFree |

Key Insight: Context-sensitivity mechanism is IDENTICAL (CFL-reachability). The difference is REPRESENTATION and PROBLEM CLASS.

16.6 When to Use Each Approach

USE IFDS WHEN:

- Transfer functions are naturally distributive
- Domain D is small and finite (taint labels, null states)
- No pointer analysis available or needed
- Need proven $O(E * D^3)$ complexity bound
- Need complete coverage of ALL reachable facts
- Problem is NOT source-sink specific

USE SVF WHEN:

- Tracking value-flows through memory (leak detection)
- Pointer analysis is already available (from Section 5.1-5.3)
- Need to handle address-taken variables precisely
- Analysis is source-sink reachability problem
- Program has many loads/stores
- Memory regions are relatively sparse ($R \ll D$)

CRITICAL: SVF REQUIRES pointer analysis as pre-requisite.

IFDS is self-contained (no pre-analysis needed).

16.7 Leak Detection Algorithm

The SVFG enables precise leak detection through a two-phase algorithm. The first phase (*some_path_analysis*) performs a forward slice from each allocation site to find which free sites are reachable—if no free is reachable, the allocation definitely leaks. The second phase (*all_path_analysis*) computes path guards: under what conditions does the allocation reach a free? If the guard is a tautology (always true), there is no leak; if it is satisfiable but not a tautology, the leak is conditional.

The *leak_result* type distinguishes four cases: *DefinitelyLeaks* (no path to free), *ConditionalLeak* (some paths leak, some don't), *NoLeak* (all paths reach free), and *ReachesGlobal* (the pointer escapes to global state, so leak detection is inconclusive).

Leak Detection on SVFG

```
module BrrrMachine.LeakDetection

type leak_result =
  | DefinitelyLeaks : source:svfg_node_id → leak_result
  | ConditionalLeak : source:svfg_node_id → condition:guard → leak_result
  | NoLeak : source:svfg_node_id → leak_result
  | ReachesGlobal : source:svfg_node_id → leak_result

(* Phase 1: SOME-PATH ANALYSIS (Forward Slice) *)
val some_path_analysis : svfg → svfg_node_id → forward_slice_result
let some_path_analysis svfg source =
  let rec explore visited call_stack worklist =
    match worklist with
    | [] → visited
    | (node, stack) :: rest →
      if Set.mem (node, stack) visited then
        explore visited call_stack rest
      else
        let visited' = Set.add (node, stack) visited in
        let successors = get_cfl_successors svfg node stack in
        explore visited' call_stack (successors @ rest)
  in
  let slice = explore Set.empty [] [(source, [])] in
  let sinks = Set.filter (fun (n, _) → is_sink n svfg) slice in
  let reaches_global = Set.∃ (fun (n, _) → is_global_store n) slice in
  { forward_slice = Set.map fst slice;
    reachable_sinks = Set.map fst sinks;
```

```

reaches_global }

(* Phase 2: ALL-PATH ANALYSIS (Guard Computation) *)
val all_path_analysis : svfg → svfg_node_id → set svfg_node_id → guard
let all_path_analysis svfg source sinks =
  let path_guards = Set.fold (fun sink acc →
    let paths = enumerate_vf_paths svfg source sink in
    let guards = List.map (compute_path_guard svfg) paths in
    guards @ acc
  ) sinks [] in
  List.fold_left (fun acc g → GOr acc g) GFalse path_guards

(* Main Leak Detection *)
val detect_leak : svfg → svfg_node_id → leak_result
let detect_leak svfg source =
  let fsr = some_path_analysis svfg source in
  if Set.is_empty fsr.reachable_sinks then
    DefinitelyLeaks source
  else if fsr.reaches_global then
    ReachesGlobal source
  else
    let freed = all_path_analysis svfg source fsr.reachable_sinks in
    if guard_is_τtology freed then
      NoLeak source
    else
      ConditionalLeak source (GNot freed)

```

Remark 16.4 (Practical Performance). **Source:** [SYX12]

On average, only 5.75% of functions and 4.31% of SVFG nodes are traversed during leak detection. The sparse representation enables demand-driven analysis that skips irrelevant program regions.

Speedup over dense approaches: 10–50x faster than iterative dense dataflow.

16.8 Cross-References

SVF Integration Points:

Prerequisites:

- Pointer Analysis (Section 5.1–5.3): REQUIRED for Memory SSA
- CPG (Section 3.1): SVFG nodes correspond to CPG statement nodes

Related Techniques:

- IFDS (Section 4.1): Alternative for distributive problems
- CFL-Reachability (Section 4.2): Shared context-sensitivity approach
- Interleaved Dyck (Section 4.2.3): Combined context+field problem
- Shape Analysis (Section 5.4–5.5): Can use SVFG for heap queries

Client Analyses:

- Memory leak detection (primary use case)
- Use-after-free detection
- Double-free detection
- Null dereference (if null is tracked as taint)

Part VI

Effect Systems and Coeffects—Tracking Computational Behavior

17. Effects as First-Class Citizens

Foundational Papers

- [Moggi91] Notions of computation and monads.
- [Plotkin03] The semantics of algebraic effects.
- [Plotkin09] Handlers of algebraic effects.
- [Leijen14] Koka: programming with row polymorphic effect types.
- [Leijen17] Type directed compilation of row-typed algebraic effects.

Effects are central to the brrr-machine. An effect is anything a computation does besides returning a value. Types tell you what values exist; **effects tell you what happens**.

17.1 The Moggi Foundation

Moggi's Key Insight (1991)

Programs compute **computations**, not just **values**.

A computation of type T may:

- Return a value of type T
- Perform effects before returning
- Never return (diverge)

Definition 17.1 (Monad). The **monad** captures this notion:

$$\begin{aligned} \text{return} : T &\rightarrow MT && \text{(Pure value, no effect)} \\ \text{bind} : MT &\rightarrow (T \rightarrow MU) \rightarrow MU && \text{(Sequence computations)} \end{aligned}$$

subject to the monad laws:

$$\begin{aligned} \text{return } a &\gg= f = f a && \text{(Left identity)} \\ m &\gg= \text{return} = m && \text{(Right identity)} \\ (m &\gg= f) \gg= g = m \gg= (\lambda x. f x \gg= g) && \text{(Associativity)} \end{aligned}$$

Definition 17.2 (Effects as Monads). Common effects have monadic interpretations:

$$\begin{aligned} \text{State } ST &= S \rightarrow (T \times S) && \text{(Read/write state)} \\ \text{Error } ET &= T + E && \text{(May fail with error)} \\ \text{Writer } WT &= T \times W && \text{(Accumulate output)} \\ \text{Reader } RT &= R \rightarrow T && \text{(Read from environment)} \\ \text{IO } T &= \text{World} \rightarrow (T \times \text{World}) && \text{(External effects)} \\ \text{Cont } RT &= (T \rightarrow R) \rightarrow R && \text{(Continuations)} \end{aligned}$$

17.2 Algebraic Effects

Papers and Implementation

- [Plotkin03], [Plotkin09] for theory.
- [Sivaramakrishnan21] for production implementation in Multicore OCaml.

Plotkin's Contribution

Monads are powerful but don't compose well. Algebraic effects separate **what** from **how**:

- **Effect operations**: Abstract interface (what can happen)
- **Effect handlers**: Concrete implementation (how it happens)

Production Implementation (Sivaramakrishnan et al. PLDI 2021)

Multicore OCaml demonstrates that algebraic effects can be added to a production language with:

- 1% mean overhead for non-effect code (validated on 54 benchmarks)
- Full DWARF debugging support (GDB, perf work correctly)
- C FFI compatibility (with boundary limitations—see Section ??)
- Runtime-enforced continuation linearity (one-shot by default)

Definition 17.3 (Effect Signature). Effect signatures declare the operations an effect provides:

```
effect State( $S$ ){
    get : ()  $\rightarrow S$ 
    put :  $S \rightarrow ()$ 
}

effect Fail{
    fail : ()  $\rightarrow \perp$ 
}

effect IO{
    read : ()  $\rightarrow \text{String}$ 
    write :  $\text{String} \rightarrow ()$ 
}
```

Definition 17.4 (Effect Handler). Effect handlers give semantics to effect operations:

```
handler state_handler( $init : S$ ){
    return( $x$ )  $\mapsto \lambda s. (x, s)$ 
    get( $()$ ,  $k$ )  $\mapsto \lambda s. k(s)(s)$            ( $k$  is continuation)
    put( $s'$ ,  $k$ )  $\mapsto \lambda s. k(())(s')$ 
}

handler maybe_handler{
    return( $x$ )  $\mapsto \text{Some}(x)$ 
    fail( $()$ ,  $k$ )  $\mapsto \text{None}$                  (Don't invoke  $k$ )
}
```

Why This Matters for Brrr-Machine

1. **Every bug is an effect violation:**

- Use-after-free: Read effect after EFree effect on same location
- Race condition: Concurrent Write effects without synchronization
- SQL injection: IO effect (query) with tainted data

2. **Effect tracking enables:**

- Purity analysis (function has no effects)
- Effect-based optimization (pure functions can be cached)
- Cross-language boundary analysis (effect mismatch = risk)

17.3 Row-Polymorphic Effects

Paper: [Leijen14], [Leijen17]

The Problem with Simple Effect Types

Consider: `read_file : String → IO String`

This says “`read_file` has IO effect” but:

- Can’t compose with **State** functions
- Can’t be polymorphic over additional effects
- Forces monomorphic effect typing

Definition 17.5 (Leijen’s Solution: Row Polymorphism). Effect types are **rows**—unordered collections with a “tail variable”:

$$\text{read_file} : \text{String} \rightarrow \langle \text{IO} \mid \varepsilon \rangle \text{String}$$

This says:

- `read_file` definitely has IO effect
- It may have other effects (ε is the “rest”)
- Callers can instantiate ε with their effects

Definition 17.6 (Row Operations).

$$\begin{array}{ll} \text{Empty row: } \langle \rangle & \\ \text{Extend: } \langle E \mid \rho \rangle & (E \text{ plus whatever is in } \rho) \\ \text{Variable: } \varepsilon & (\text{Unknown effects}) \end{array}$$

Definition 17.7 (Subtyping via Row Extension). If $f : T \rightarrow \langle E_1 \mid \varepsilon \rangle U$ and we call f in context with effects $\langle E_1, E_2 \rangle$, then $\varepsilon = \langle E_2 \rangle$ and the call has type $\langle E_1, E_2 \rangle$.

Definition 17.8 (Effect-Polymorphic Map).

$$\text{map} : \forall \alpha \beta \varepsilon. (\alpha \rightarrow \langle \varepsilon \rangle \beta) \rightarrow \text{List } \alpha \rightarrow \langle \varepsilon \rangle \text{List } \beta$$

This map is effect-polymorphic:

- `map pure_f xs` has no effects
- `map io_f xs` has IO effect
- Effect inferred from the function argument

17.4 F* Specification: Effect System

The following F* specification formalizes the effect system described above. The key components are:

1. **Effect kinds**: A comprehensive taxonomy of effects organized by category (memory, control, I/O, concurrency, channels, resources). Each constructor captures one primitive operation that has observable side effects.
2. **Effect rows**: Row-polymorphic effect types following Leijen’s design, where effects are accumulated in extensible rows. The `RowVar` constructor enables effect polymorphism.
3. **Session types**: Following Honda 1998/2008, channel effects include session type operations (select, branch, delegate) for typed communication protocols.

Effect System for Brrr-Machine

```
module BrrrMachine.Effects
```

```
(* Effect kinds - The taxonomy of effects *)
type effect_kind =
```

```

(* Memory effects *)
| ERead      : loc:abstract_loc → effect_kind
| EWrite     : loc:abstract_loc → effect_kind
| EAlloc     : effect_kind
| EFree      : loc:abstract_loc → effect_kind
(* Control effects *)
| EDiverge   : effect_kind
| EThrow     : exn_type:string → effect_kind
| ECatch     : exn_type:string → effect_kind
(* I/O effects *)
| EInput     : source:io_source → effect_kind
| EOutput    : sink:io_sink → effect_kind
| EFileRead  : path:string → effect_kind
| EFileWrite : path:string → effect_kind
| ENetwork   : effect_kind
| ERandom    : effect_kind
| EClock     : effect_kind
(* Concurrency effects *)
| ESpawn     : effect_kind
| EJoin      : effect_kind
| ELock      : lock_id:nat → effect_kind
| EUnlock    : lock_id:nat → effect_kind
| ESend      : chan:nat → payload_type:ir_type → effect_kind
| ERecv      : chan:nat → payload_type:ir_type → effect_kind
| EAtomic    : effect_kind
(* Channel effects - Honda 1998/2008 session type operations *)
| EChanCreate : chan:nat → elem_type:ir_type → buffer_size:nat → effect_kind
| EChanClose  : chan:nat → effect_kind
| ESelect     : chan:nat → label:string → effect_kind
| EBranch     : chan:nat → labels:list string → effect_kind
| EDelegate   : chan:nat → delegated_chan:nat → effect_kind
(* Resource effects *)
| EAcquire    : resource_type:string → effect_kind
| ERelease    : resource_type:string → effect_kind
| EUse        : resource_type:string → effect_kind

and io_source = Stdin | EnvVar of string | FileInput | NetworkInput | UserInput
and io_sink = Stdout | Stderr | FileOutput | NetworkOutput | DatabaseOutput

```

Effect Rows—Row-Polymorphic Effect Types (Leijen 2014, 2017)

```

type effect_row =
| RowEmpty   : effect_row
  (* ≠ - no effects *)
| RowExtend  : effect:effect_kind → tail:effect_row → effect_row
  (* <E | ρ> - effect E plus tail *)
| RowVar     : var:effect_var → effect_row
  (* ε - unknown effects *)

and effect_var = nat (* De Bruijn index or unique ID *)

```

Critical: Effect Rows Allow Duplicate Labels (Leijen 2014)

This is NOT a set! The following are DISTINCT:

- $\langle \text{exn} \mid \mu \rangle$ — one exception effect
- $\langle \text{exn}, \text{exn} \mid \mu \rangle$ — TWO exception effects

Why this matters: Effect **handlers** consume labels one at a time. A single catch handler removes ONE exn from the row:

$$\text{catch} : \langle \text{exn} \mid \varepsilon \rangle \tau \rightarrow (\text{exn} \rightarrow \tau) \rightarrow \langle \varepsilon \rangle \tau$$

If you have $\langle \text{exn}, \text{exn} \mid \mu \rangle$, after one catch you still have $\langle \text{exn} \mid \mu \rangle$!

Effect Elimination and Row Operations

```

(* Effect elimination removes ONE instance *)
val effect_handle : effect_kind → effect_row → option effect_row
let rec effect_handle eff row =
  match row with
  | RowEmpty → None (* Effect not present *)
  | RowExtend e tail →
    if e = eff then Some tail (* Remove ONE instance *)
    else
      (match effect_handle eff tail with
       | Some tail' → Some (RowExtend e tail')
       | None → None)
  | RowVar _ → None (* Can't remove from variable *)

(* Effect row membership *)
val row_contains : effect_row → effect_kind → bool
let rec row_contains row eff =
  match row with
  | RowEmpty → false
  | RowExtend e tail → e = eff || row_contains tail eff
  | RowVar _ → true (* Conservative: assume present *)

(* Effect row union *)
val row_union : effect_row → effect_row → effect_row
let rec row_union r1 r2 =
  match r1 with
  | RowEmpty → r2
  | RowExtend e tail → RowExtend e (row_union tail r2)
  | RowVar v → RowVar v (* Variable absorbs other row *)

(* Effect row subsumption *)
val row_subsumes : effect_row → effect_row → bool
let rec row_subsumes larger smaller =
  match smaller with
  | RowEmpty → true
  | RowExtend e tail →
    row_contains larger e && row_subsumes larger tail
  | RowVar _ → true (* Variable matches anything *)

```

The following F* code models session type semantics for channel effects. Key concepts:

- **Channel linearity:** Each channel use consumes the current session type prefix
- **Commutativity:** Operations on different channels commute (can be reordered)
- **Delegation:** Transfers channel ownership to another endpoint (linear transfer)

Note: `List.assoc_opt` is a helper function (not standard F*) returning `Some v` if key found, `None` otherwise.

Channel Effect Composition Rules—Honda 1998/2008 Session Types

```

(* Channel effects are LINEAR: each channel use must be accounted for exactly once.
   This models session type linearity where each communication action consumes
   the current type prefix and advances to the continuation. *)

(* Channel effect commutativity:
   ESend(k, T); ERecv(k', T') commutes if k ≠ k'
   Operations on DIFFERENT channels are independent (parallel composition) *)
val channel_effects_commute : effect_kind → effect_kind → bool
let channel_effects_commute e1 e2 =
  match e1, e2 with
  | ESend k1 _, ESend k2 _ → k1 ≠ k2
  | ESend k1 _, ERecv k2 _ → k1 ≠ k2
  | ERecv k1 _, ESend k2 _ → k1 ≠ k2
  | ERecv k1 _, ERecv k2 _ → k1 ≠ k2
  | ESend k1 _, ESelect k2 _ → k1 ≠ k2

```

```

| ESend k1 _, EBranch k2 _ → k1 ≠ k2
| ERecv k1 _, ESelect k2 _ → k1 ≠ k2
| ERecv k1 _, EBranch k2 _ → k1 ≠ k2
| ESelect k1 _, ESelect k2 _ → k1 ≠ k2
| ESelect k1 _, EBranch k2 _ → k1 ≠ k2
| EBranch k1 _, EBranch k2 _ → k1 ≠ k2
| EChanClose k1, EChanClose k2 → k1 ≠ k2
| EChanClose k1, ESend k2 _ → k1 ≠ k2
| EChanClose k1, ERecv k2 _ → k1 ≠ k2
| _, _ → true  (* Non-channel effects always commute with channel effects *)

(* Channel linearity check:
   A channel cannot be used after being closed.
   Closing a channel CONSUMES it (linear resource). *)
type channel_state =
| ChanOpen : elem_type:ir_type → buffer_size:nat → channel_state
| ChanClosed : channel_state

type channel_context = list (nat * channel_state)

val check_channel_linearity : channel_context → effect_kind → option channel_context
let check_channel_linearity ctx eff =
  match eff with
  | EChanCreate k elem_ty buf_sz →
    if List.assoc_opt k ctx = None then
      Some ((k, ChanOpen elem_ty buf_sz) :: ctx)
    else None  (* Channel already ∃ *)
  | ESend k _ | ERecv k _ | ESelect k _ | EBranch k _ →
    (match List.assoc_opt k ctx with
     | Some (ChanOpen _ _) → Some ctx  (* Channel is open, use allowed *)
     | _ → None)  (* Channel closed or doesn't exist *)
  | EChanClose k →
    (match List.assoc_opt k ctx with
     | Some (ChanOpen _ _) →
      Some (List.map (fun (k', s) →
        if k' = k then (k', ChanClosed) else (k', s)) ctx)
     | _ → None)  (* Already closed or doesn't exist *)
  | EDelegate k delegated_k →
    (* Delegation transfers ownership: delegated channel must be open,
       becomes unavailable in current context (transferred to receiver) *)
    (match List.assoc_opt k ctx, List.assoc_opt delegated_k ctx with
     | Some (ChanOpen _ _), Some (ChanOpen _ _) →
      Some (List.filter (fun (k', _) → k' ≠ delegated_k) ctx)
     | _, _ → None)
  | _ → Some ctx  (* Non-channel effect doesn't affect channel context *)

```

Definition 17.9 (Channel Effect Typing Rules (Row-Polymorphic Style)).

| | |
|--|---------------|
| $\frac{}{\Gamma \vdash \text{chan_create}\langle T, n \rangle() : \text{Chan}\langle T \rangle \ \& \ \langle \text{EChanCreate}(k, T, n) \mid \varepsilon \rangle}$ | (Chan-Create) |
| $\frac{\Gamma \vdash ch : \text{Chan}\langle T \rangle \quad \Gamma \vdash v : T}{\Gamma \vdash ch.\text{send}(v) : () \ \& \ \langle \text{ESend}(k, T) \mid \varepsilon \rangle}$ | (Chan-Send) |
| $\frac{\Gamma \vdash ch : \text{Chan}\langle T \rangle}{\Gamma \vdash ch.\text{recv}() : T \ \& \ \langle \text{ERecv}(k, T) \mid \varepsilon \rangle}$ | (Chan-Recv) |
| $\frac{\Gamma \vdash ch : \text{Chan}\langle T \rangle}{\Gamma \vdash ch.\text{close}() : () \ \& \ \langle \text{EChanClose}(k) \mid \varepsilon \rangle}$ | (Chan-Close) |
| $\frac{\Gamma \vdash ch : \text{Chan}\langle T \rangle \quad l \in \{l_1, \dots, l_n\}}{\Gamma \vdash ch.\text{select}(l) : () \ \& \ \langle \text{ESelect}(k, l) \mid \varepsilon \rangle}$ | (Chan-Select) |
| $\frac{\Gamma \vdash ch : \text{Chan}\langle T \rangle \quad \Gamma, x : l_i \vdash e_i : U \ \& \ \langle \varepsilon_i \rangle \text{ for each } i}{\Gamma \vdash ch.\text{branch}\{l_1 \Rightarrow e_1, \dots, l_n \Rightarrow e_n\} : U \ \& \ \langle \text{EBranch}(k, \{l_1, \dots, l_n\}) \mid \varepsilon_1 \sqcup \dots \sqcup \varepsilon_n \rangle}$ | (Chan-Branch) |
| $\frac{\Gamma \vdash ch : \text{Chan}\langle S \rangle \quad \Gamma \vdash \text{delegated_ch} : \text{Chan}\langle T \rangle}{\Gamma \vdash ch.\text{delegate}(\text{delegated_ch}) : () \ \& \ \langle \text{EDelegate}(k, k') \mid \varepsilon \rangle}$ | (Delegate) |

Note: In Delegate, *delegated_ch* is consumed—linear ownership transfer.

Effect Signatures—For Functions and Procedures

```

type effect_signature = {
  (* Effect row for this function *)
  effects : effect_row;
  (* Specific categories for quick checks *)
  may_read : bool;
  may_write : bool;
  may_alloc : bool;
  may_free : bool;
  may_throw : bool;
  may_diverge : bool;
  may_io : bool;
  may_spawn : bool;
  (* Channel effects - Honda 1998/2008 session operations *)
  may_send : bool;
  may_recv : bool;
  may_create_chan : bool;
  may_close_chan : bool;
  may_select : bool;
  may_branch : bool;
  may_delegate : bool;
  (* Channel context: maps channel IDs to their current session types *)
  channel_types : list (nat * local_session_type);
  is_pure : bool; (* No effects at all *)
}

let empty_effect_sig : effect_signature = {
  effects = RowEmpty;
  may_read = false; may_write = false;
  may_alloc = false; may_free = false;
  may_throw = false; may_diverge = false;
  may_io = false; may_spawn = false;
  may_send = false; may_recv = false;
  may_create_chan = false; may_close_chan = false;
  may_select = false; may_branch = false;
}

```

```

    may_delegate = false;
    channel_types = [];
    is_pure = true;
}

```

Effect Trace—Runtime Effect Sequence

```

type effect_event = {
  kind : effect_kind;
  location : node_id;
  timestamp : nat;
  thread : option nat;
}

type effect_trace = list effect_event

```

Happens-Before Relation (Simplified for Effect Traces)

```

(* Source: Lamport 1978 (basic), extended per Batty 2011 (C11 memory model)

This is a SIMPLIFIED version for effect_event analysis.
For the FULL C11 memory model with consume semantics, release sequences,
and coherence axioms, see Section 6.5.2 happens_before_c11. *)

val happens_before : effect_event → effect_event → bool
let happens_before e1 e2 =
  (* Case 1: Sequenced-before (same thread program order) *)
  (e1.thread = e2.thread && e1.timestamp < e2.timestamp) ||
  (* Case 2: Synchronizes-with (cross-thread via release/acquire) *)
  (is_release e1 && is_acquire e2 && synchronizes e1 e2) ||
  (* Case 3: Dependency-ordered-before (consume semantics) *)
  (is_release e1 && is_consume e2 && synchronizes e1 e2)

(* Helper: Check if effect has release semantics *)
val is_release : effect_event → bool
let is_release e =
  match e.kind with
  | EUnlock _ → true          (* Mutex unlock has release semantics *)
  | EAtomic → true           (* Conservative: treat atomics as release *)
  | ESend _ → true           (* Channel send has release semantics *)
  | _ → false

(* Helper: Check if effect has acquire semantics *)
val is_acquire : effect_event → bool
let is_acquire e =
  match e.kind with
  | ELock _ → true           (* Mutex lock has acquire semantics *)
  | EAtomic → true           (* Conservative: treat atomics as acquire *)
  | ERecv _ → true           (* Channel receive has acquire semantics *)
  | _ → false

(* Helper: Check if two events synchronize *)
val synchronizes : effect_event → effect_event → bool
let synchronizes e1 e2 =
  match e1.kind, e2.kind with
  | EUnlock l1, ELock l2 → l1 = l2          (* Unlock/Lock on same mutex *)
  | ESend c1, ERecv c2 → c1 = c2           (* Send/Recv on same channel *)
  | _, EJoin → true                        (* Thread join synchronizes *)
  | EAtomic, EAtomic → true                (* Atomic operations may synchronize *)
  | _ → false

```

Effect-Based Bug Detection

```

type effect_violation =
  | UseAfterFree : loc:abstract_loc → free_site:node_id → use_site:node_id
    → effect_violation

```

```

| DoubleFree : loc:abstract_loc → first_free:node_id → second_free:node_id
               → effect_violation
| DataRace : loc:abstract_loc → access1:effect_event → access2:effect_event
               → effect_violation
| Deadlock : locks:list nat → threads:list nat → effect_violation
| ResourceLeak : resource:string → acquire_site:node_id → effect_violation
| UnhandledEffect : effect:effect_kind → site:node_id → effect_violation

(* Detect violations in effect trace *)
val detect_violations : effect_trace → list effect_violation
let detect_violations trace =
  let violations = ref [] in
  (* Track memory state *)
  let freed : set abstract_loc ref = ref Set.empty in
  List.iter (fun event →
    match event.kind with
    | EFree loc →
      if Set.mem loc !freed then
        violations := DoubleFree loc event.location event.location :: !violations;
        freed := Set.add loc !freed
    | ERead loc | EWrite loc →
      if Set.mem loc !freed then
        violations := UseAfterFree loc 0 event.location :: !violations
    | _ → ()
  ) trace;
  (* Detect data races *)
  let pairs = all_pairs trace in
  List.iter (fun (e1, e2) →
    if same_location e1 e2 &&
      (is_write e1 || is_write e2) &&
      not (happens_before e1 e2) &&
      not (happens_before e2 e1) then
      violations := DataRace (get_loc e1) e1 e2 :: !violations
  ) pairs;
  !violations

```

18. Effect Handler Limitations

Critical Limitations of Algebraic Effect Handlers ([Plotkin09], [Sivaramakrishnan21])

Plotkin–Pretnar 2009 proves that effect handlers are **homomorphisms** from free algebraic models. This has important consequences:

1. Continuations are NOT algebraic

$$\text{call/cc} : ((a \rightarrow b) \rightarrow a) \rightarrow a$$

Problem: Captured continuation escapes the handler scope. The homomorphism property does NOT hold.

2. Parallel composition CANNOT be handled

$$\text{parallel} : m\ a \rightarrow m\ b \rightarrow m\ (a \times b)$$

Problem: Requires BINARY handler (handling two computations). Plotkin–Pretnar: “it seems that parallel cannot be [implemented]”

3. C FFI boundaries BLOCK effect propagation (Sivaramakrishnan 2021)

Effects CANNOT propagate across C stack frames!

$$\text{OCaml (handler)} \rightarrow \text{C function} \rightarrow \text{OCaml (perform effect)}$$

The C frames don’t know about effect handlers. An effect performed in the inner OCaml

code cannot be handled by the outer OCaml handler. This is a **fundamental limitation** of retrofitting effects onto existing languages with C FFI.

Consequence: Use different mechanisms for these effects:

- Continuations: Delimited control operators (shift/reset)
- Parallelism: Concurrency model with explicit threads
- Cross-FFI: Exception-style handling only (no continuation capture)

Effect Handler Correctness Criterion

```
(* A handler is CORRECT iff it preserves the equations of the effect theory.
   This is the key condition from Plotkin-Pretnar 2009. *)

type effect_equation = {
  lhs : effect_term;
  rhs : effect_term;
}

type effect_theory = {
  signature : effect_signature;
  equations : list effect_equation;
}

val handler_correct : #e:effect_theory → #m:Type → handler e.signature m → bool
let handler_correct #e #m h =
  (* Handler must preserve ALL equations *)
  List.for_all (fun eq →
    interpret_with_handler h eq.lhs == interpret_with_handler h eq.rhs
  ) e.equations
```

Effects That Are NOT Algebraic—Do Not Model as Handlers

```
(* WRONG: Trying to model call/cc as an effect
   This DOES NOT WORK because the continuation (cont b) can escape
   the handler scope, violating the homomorphism property. *)

(* CORRECT: Model call/cc with delimited control *)
type delimited_control =
  | Shift : ((a → answer) → answer) → delimited_control
  | Reset : (() → answer) → answer → delimited_control

(* WRONG: Trying to handle parallel composition
   Can't handle TWO computations at once! *)

(* CORRECT: Model parallelism with explicit concurrency *)
type concurrent_effect =
  | Spawn : (() → unit) → thread_id → concurrent_effect
  | Join : thread_id → unit → concurrent_effect
  | Yield : unit → concurrent_effect
```

What CAN Be Modeled as Algebraic Effects (Safe to Use Handlers)

```
(* State: get/put - ALGEBRAIC *)
type state_eff (s:Type) =
  | Get : s
  | Put : s → unit

(* Exceptions: raise - ALGEBRAIC (arity 0, no continuation) *)
type exception_eff (e:Type) =
  | Raise : e → 'a

(* Nondeterminism: fail/choose - ALGEBRAIC *)
type nondet_eff =
  | Fail : 'a
  | Choose : unit → bool
```

```

(* Reader: ask - ALGEBRAIC *)
type reader_eff (r:Type) =
  | Ask : r

(* Writer: tell - ALGEBRAIC *)
type writer_eff (w:Type) =
  | Tell : w → unit

(* SUMMARY: Effect handlers work for sequential, single-continuation effects.
   For parallelism and first-class continuations, use other mechanisms. *)

```

18.1 Production Effect Handler Implementation

Paper: Sivaramakrishnan, Dolan, White, Kelly, Jaffer, Madhavapeddy — PLDI 2021

Fiber-Based Effect Handler Implementation

Key Insight: Separate OCaml frames into “fibers”—heap-allocated stack segments that can be captured and resumed independently.

Traditional Stack

| |
|--------------|
| C frames |
| OCaml frames |
| C frames |
| OCaml frames |

Multicore OCaml

| |
|-------------------------------|
| Fiber Stack (one per handler) |
| OCaml frames |
| C stack (unchanged) |

Fiber Operations:

1. Handler installation = Allocate new fiber (8KB default)
2. Effect perform = Capture fiber state as continuation
3. Continue = Restore fiber and resume execution
4. Discontinue = Restore and raise exception in captured context

Stack Overflow Checking (Red Zone Technique): Every function checks: $sp < fiber_end + red_zone_size$ (16 words). If true: grow fiber stack (2×) or signal overflow. Trade-off: 19% text section size increase for 1% runtime overhead.

Continuation Linearity (Runtime Enforced): Continuations are ONE-SHOT by default. Resuming a captured continuation twice = runtime error. Trade-off: Runtime check vs compile-time linear types.

Performance Characteristics:

- Mean overhead (no effects): 1%
- Median overhead: <5% for 32/54 benchmarks
- Worst case overhead: ~15% (allocation-heavy code)
- Text section size increase: 19% (with 16-word red zone)
- Exception handling: No measurable change
- External C calls: No measurable change

Fiber-Based Effect Handler Model

```

module BrrrMachine.Effects.FiberImplementation

(* Fiber: heap-allocated stack segment for effect handlers *)
type fiber_id = nat

type fiber = {
  id : fiber_id;

```

```

frames : list stack_frame;
stack_size : nat;
stack_capacity : nat; (* Default 1024 words = 8KB *)
handler : option handler_closure;
parent : option fiber_id;
}

type handler_clause =
| ReturnClause : (value → computation) → handler_clause
| EffectClause : effect_label:string → (value → continuation → computation)
               → handler_clause
| ExceptionClause : exn_label:string → (value → computation) → handler_clause

type handler_closure = {
  clauses : list handler_clause;
}

(* Continuation: captured fiber state (linear by default) *)
type continuation_state = | Fresh | Resumed | Finalized

type continuation = {
  captured_fiber : fiber;
  captured_handlers : list handler_closure;
  state : ref continuation_state; (* Runtime linearity tracking *)
}

(* CRITICAL: C FFI boundary detection *)
type stack_segment =
| OCamlSegment : fiber_id → stack_segment
| CSegment : list c_frame → stack_segment

type mixed_stack = list stack_segment

(* Effects CANNOT cross C segments *)
val effect_can_reach_handler : effect_label:string → mixed_stack → bool
let rec effect_can_reach_handler label stack =
  match stack with
  | [] → false
  | OCamlSegment fid :: rest →
    if fiber_handles_effect fid label then true
    else effect_can_reach_handler label rest
  | CSegment _ :: _ →
    (* C frames BLOCK effect propagation! *)
    false

(* Continue with linearity check *)
val continue_checked : continuation → value → result computation string
let continue_checked k v =
  match !(k.state) with
  | Fresh →
    k.state := Resumed;
    Ok (resume_fiber k.captured_fiber v)
  | Resumed →
    Error "Continuation already resumed (linearity violation)"
  | Finalized →
    Error "Continuation was finalized by GC"

```

19. C11/C++ Memory Model

C11/C++ Memory Model Formalization

Primary Source: [Batty11] — “Mathematizing C++ Concurrency”

Foundational: [Boehm08] — “Foundations of the C++ Concurrency Memory Model”

Relationship: Boehm 2008 defines the C++ memory model informally and proves the DRF-SC theorem. Batty 2011 provides the **rigorous mathematical formalization** we use here. Both papers are essential:

- Boehm 2008: Conceptual foundation, race definitions, SC-DRF theorem
- Batty 2011: Formal execution model, coherence axioms, proof framework

Key Insight: Concurrent program semantics require **execution-based** models, not just sequential effect traces. The C11 memory model defines:

1. Memory orderings (relaxed to sequentially consistent)
2. Execution witnesses (reads-from, modification order)
3. Happens-before relation with release/acquire synchronization
4. Coherence axioms ensuring per-location consistency

Race Definitions (Boehm 2008 terminology):

- **Type 1 Race:** Conflicting actions adjacent in a total interleaving order (used in operational/interleaving semantics)
- **Type 2 Race:** Conflicting data accesses unordered by happens-before (used in axiomatic/execution-based semantics)

Theorem 8.1–8.2 (Boehm): Type 1 and Type 2 races are **equivalent** for programs without undefined behavior from other sources.

Integration with Effect System:

- Atomic effect kind is REPLACED by memory-order-parameterized atomics
- Effect traces are EXTENDED to execution witnesses
- Happens-before is EXTENDED with consume/dependency

Critical Limitation: Thin-Air Problem in Relaxed Atomics

The C11 axiomatic model (Batty 2011) permits “out-of-thin-air” values in programs using relaxed atomics. This is a **soundness bug** that violates:

- Basic invariant reasoning (cannot prove $x = y = 0$ is maintained)
- DRF-SC guarantees (race-free programs may behave non-SC)
- Type safety (Java-like languages cannot guarantee type-correct values)

Litmus Test (LBd—Thin Air):

Initial: $x = y = 0$ (relaxed atomics)

Thread 1: $a := x; y := a;$

Thread 2: $x := y;$

Outcome $a = 1$: **IMPOSSIBLE** (no source) but **ALLOWED** by C11 axiomatic model!

Safe Subset: Use ONLY release/acquire synchronization (no relaxed atomics) to avoid thin-air issues.

Recommended: For sound relaxed atomic reasoning, use Promising Semantics 2.0 ([Lee20]) formalized in Section 19.7.

References:

- [Kang17]: “A Promising Semantics for Relaxed-Memory Concurrency”
- [Lee20]: “Promising 2.0” (enables global opts, fixes ARMv8)

- [Podkopaev19]: IMM intermediate model
- [Vafeiadis15]: “Common Compiler Optimisations are Invalid in C11”
- [Batty11]: “Mathematizing C++ Concurrency”

19.1 Memory Orderings and Actions

C11 Memory Orderings (Batty et al. 2011, Section 2)

```

module BrrrMachine.Effects.C11MemoryModel

(* Memory order enumeration
   Partial ordering: seq_cst > acq_rel > acquire/release > consume > relaxed *)
type memory_order =
| MO_relaxed      (* No ordering constraints - only atomicity guaranteed *)
| MO_consume      (* Data dependency ordering - weaker than acquire *)
| MO_acquire      (* Acquire semantics - synchronizes with release *)
| MO_release      (* Release semantics - synchronizes with acquire *)
| MO_acq_rel      (* Both acquire and release - for read-modify-write *)
| MO_seq_cst      (* Sequentially consistent - total order of all SC ops *)

(* Memory ordering strength comparison *)
val mo_stronger : memory_order → memory_order → bool
let mo_stronger mo1 mo2 =
  match mo1, mo2 with
  | MO_seq_cst, _ → true
  | _, MO_seq_cst → false
  | MO_acq_rel, (MO_acquire | MO_release | MO_consume | MO_relaxed) → true
  | (MO_acquire | MO_release | MO_consume | MO_relaxed), MO_acq_rel → false
  | MO_acquire, (MO_consume | MO_relaxed) → true
  | MO_release, (MO_consume | MO_relaxed) → true
  | (MO_consume | MO_relaxed), MO_acquire → false
  | (MO_consume | MO_relaxed), MO_release → false
  | MO_consume, MO_relaxed → true
  | MO_relaxed, MO_consume → false
  | MO_acquire, MO_release → false (* Incomparable *)
  | MO_release, MO_acquire → false (* Incomparable *)
  | mo1, mo2 → mo1 = mo2

(* Check if ordering provides acquire/release/consume semantics *)
val is_acquire : memory_order → bool
let is_acquire mo = match mo with
  | MO_acquire | MO_acq_rel | MO_seq_cst → true | _ → false

val is_release : memory_order → bool
let is_release mo = match mo with
  | MO_release | MO_acq_rel | MO_seq_cst → true | _ → false

val is_consume : memory_order → bool
let is_consume mo = match mo with
  | MO_consume | MO_acquire | MO_acq_rel | MO_seq_cst → true | _ → false

```

Memory Action Types (Batty et al. 2011, Figure 1)

```

type action_id = nat
type thread_id = nat
type location = abstract_loc
type value = int
type mutex_id = nat

type memory_action =
(* Non-atomic memory operations - subject to data race UB *)
| Rna : loc:location → val_read:value → memory_action
| Wna : loc:location → val_written:value → memory_action

```

```

(* Atomic memory operations - race-free by construction *)
| R : loc:location → mo:memory_order → val_read:value → memory_action
| W : loc:location → mo:memory_order → val_written:value → memory_action
| RMW : loc:location → mo:memory_order → val_read:value → val_written:value
    → memory_action

(* Fence operations *)
| Fence : mo:memory_order → memory_action

(* Lock operations - implicit acquire/release semantics *)
| Lock : mutex:mutex_id → memory_action
| Unlock : mutex:mutex_id → memory_action

(* Helper functions *)
val action_location : memory_action → option location
val action_memory_order : memory_action → option memory_order
val is_write : memory_action → bool
val is_read : memory_action → bool
val is_atomic_action : memory_action → bool
val same_location : memory_action → memory_action → bool

```

19.2 Execution Witnesses and Candidate Executions

Execution Witness Structure (Batty et al. 2011, Section 3)

```

type relation (a : Type) = a → a → bool

(* The witness contains relations NOT determined by program text *)
type execution_witness = {
  (* Reads-from: maps each read action to the write it reads from *)
  reads_from : map action_id action_id;

  (* Modification order: per-location total order of all writes *)
  modification_order : map location (list action_id);

  (* SC order: total order of all sequentially consistent operations *)
  sc_order : list action_id;
}

(* Candidate execution combines actions, relations, and witness *)
type candidate_execution = {
  actions : set memory_action;
  action_ids : map memory_action action_id;
  id_to_action : map action_id memory_action;
  thread_of : map action_id thread_id;
  sequenced_before : relation action_id;
  additional_synchronized_with : relation action_id;
  data_dependency : relation action_id;
  witness : execution_witness;
}

```

Synchronizes-With Relation (Batty 2011 Section 3.2)

```

val synchronizes_with : candidate_execution → action_id → action_id → bool
let synchronizes_with exec a b =
  match get_action exec a, get_action exec b with
  | Some act_a, Some act_b →
    (* Case 1: Release write synchronizes with acquire read via release sequence *)
    (is_write act_a && is_release (Option.get (action_memory_order act_a)) &&
     is_read act_b && is_acquire (Option.get (action_memory_order act_b)) &&
     reads_from_release_sequence exec b a) ||

    (* Case 2: Release fence synchronizes with acquire fence *)

```

```

(match act_a, act_b with
| Fence mo_a, Fence mo_b → is_release mo_a && is_acquire mo_b &&
  fence_synchronizes exec a b
| _, _ → false) ||

(* Case 3: Unlock synchronizes with subsequent lock on same mutex *)
(match act_a, act_b with
| Unlock m1, Lock m2 → m1 = m2 && unlock_before_lock exec a b
| _, _ → false) ||

(* Case 4: Additional synchronization from thread operations *)
exec.additional_synchronized_with a b
| _, _ → false

```

Happens-Before Relation (Extended C11 Semantics)

```

(* This EXTENDS the simple happens_before with full C11 semantics.
   Happens-before is the transitive closure of:
   1. Sequenced-before (program order within thread)
   2. Synchronizes-with (cross-thread via release/acquire)
   3. Dependency-ordered-before (consume semantics) *)

val happens_before_c11 : candidate_execution → action_id → action_id → bool
let rec happens_before_c11 exec a b =
  exec.sequenced_before a b ||
  synchronizes_with exec a b ||
  dependency_ordered_before exec a b ||
  Set.∃ (fun c →
    happens_before_c11 exec a c && happens_before_c11 exec c b
  ) (all_action_ids exec)

```

19.3 Release Sequences

Definition 19.1 (Release Sequence (Batty 2011 Section 3.2)). A release sequence headed by a release write A consists of:

- A itself
- Subsequent writes to the same location by the same thread
- RMW operations that read from the sequence

This enables lock-free algorithms to work correctly by allowing synchronization to “chain” through multiple atomic operations.

Release Sequence Membership

```

val in_release_sequence_batty : candidate_execution → action_id → action_id → bool
let rec in_release_sequence_batty exec head action =
  match get_action exec head with
  | None → false
  | Some head_act →
    if not (is_release_write head_act) then false
    else if not (same_location_id exec head action) then false
    else
      (* Case 1: action IS the head *)
      action = head ||
      (* Case 2: Same thread, sequenced after, and is a write *)
      (same_thread exec head action &&
        exec.sequenced_before head action &&
        (match get_action exec action with
        | Some act → is_write act | None → false)) ||
      (* Case 3: RMW that reads from something in the sequence *)
      (match get_action exec action with
      | Some act when is_rmw act →
        (match Map.find action exec.witness.reads_from with
        | Some prev_write → in_release_sequence_batty exec head prev_write

```

```

    | None → false)
  | _ → false)

```

Theorem 19.2 (Release Sequence RMW Extension). *If A is in the release sequence of H , and B reads from A , and B is an RMW, then B is also in the release sequence of H .*

19.4 Coherence Axioms

Coherence Axioms (Batty 2011 Section 4)

These four axioms ensure per-location consistency between the happens-before relation and the modification order. They are the key soundness requirements for the C11 memory model.

Intuition: Even though different threads may observe writes in different orders (relaxed atomics), the four coherence axioms guarantee that:

- Causally related observations are consistent
- No thread observes “time travel” (reading older values after newer)

Definition 19.3 (CoRR: Coherent Read-Read). If two reads are ordered by happens-before and both read from writes to the same location, the later read cannot see an earlier write (in modification order) than the earlier read saw.

Intuition: If I see value V , then anyone who observes my read (via happens-before) cannot see a value from before V .

Definition 19.4 (CoWR: Coherent Write-Read). A read cannot observe a write that happens-after the read.

Intuition: You cannot read from the future.

Definition 19.5 (CoWW: Coherent Write-Write). If two writes are ordered by happens-before, they must be in the same order in the modification order.

Intuition: Causality is respected in the write history.

Definition 19.6 (CoRW: Coherent Read-Write (Acyclicity)). The combined relation $(hb \cup rf \cup mo \cup rb)$ must be acyclic, where rb (reads-before) $= mo ; rf^{-1}$.

This prevents causal cycles that would make the execution internally inconsistent.

Coherence Implementation

```

val coherent_read_read_batty : candidate_execution → bool
val coherent_write_read_batty : candidate_execution → bool
val coherent_write_write_batty : candidate_execution → bool
val coherent_read_write_batty : candidate_execution → bool

val is_coherent_batty : candidate_execution → bool
let is_coherent_batty exec =
  coherent_read_read_batty exec &&
  coherent_write_read_batty exec &&
  coherent_write_write_batty exec &&
  coherent_read_write_batty exec

```

Theorem 19.7 (Coherence Implies Per-Location SC). *Under coherence, each individual location behaves as if accessed sequentially consistently, even if the overall execution is not SC.*

19.5 Data Race Detection

Definition 19.8 (Data Race (Batty 2011 Section 2.3)). A **data race** is the fundamental source of undefined behavior in C11. Two memory operations **race** if they:

1. Access the same location
2. At least one is a write

3. At least one is non-atomic
4. They are not ordered by happens-before

Key Insight: If a program has NO data races under SC semantics, then ALL its behaviors are sequentially consistent (DRF-SC guarantee).

Definition 19.9 (Conflict Relation). Two actions **conflict** if they access the same location and at least one is a write. This is a necessary but not sufficient condition for a data race.

Data Race Predicate

```
val is_data_race_batty : candidate_execution → action_id → action_id → bool
let is_data_race_batty exec a b =
  match get_action exec a, get_action exec b with
  | Some act_a, Some act_b →
    conflicts act_a act_b &&
    not (both_atomic act_a act_b) &&
    not (happens_before_c11 exec a b) &&
    not (happens_before_c11 exec b a)
  | _, _ → false

val no_data_races_batty : candidate_execution → bool
let no_data_races_batty exec =
  Set.for_all (fun a →
    Set.for_all (fun b →
      not (is_data_race_batty exec a b)
    ) (all_action_ids exec)
  ) (all_action_ids exec)
```

Definition 19.10 (Unsequenced Race (C-specific)). In addition to data races, C has “unsequenced races”—conflicting accesses within a **single expression** that are unsequenced. Example: `i = i++` has unsequenced race on `i`.

Definition 19.11 (Undefined Behavior Detection). A program has undefined behavior if ANY consistent execution has:

- A data race, OR
- An unsequenced race, OR
- An indeterminate read

19.6 Sequential Consistency and SC-DRF

Sequential Consistency for Data-Race-Free Programs (Batty 2011 Section 5, Boehm–Adve 2008)

The DRF-SC (Data Race Freedom implies Sequential Consistency) theorem is the **fundamental guarantee** of the C11 memory model.

Informal Statement: If your program has no data races when run with SC semantics, then all its behaviors under C11 will also be SC.

This allows programmers to reason about correctly synchronized code using the simple SC model, while the compiler/hardware can still optimize based on the relaxed C11 model.

Definition 19.12 (Sequentially Consistent Execution). An execution is SC if there exists a total order of all memory accesses that:

1. Respects program order (sequenced-before)
2. Each read sees the most recent write in the total order

Definition 19.13 (DRF Program). A program is DRF (Data Race Free) if, when executed under SC semantics, no execution has a data race.

Theorem 19.14 (DRF-SC Theorem (Main Result)). *The fundamental guarantee of the C11 memory model.*

If a program has no data races under SC semantics, then:

1. *All its C11-consistent executions are also SC, OR*
2. *All its C11-consistent executions have the same observable behavior as some SC execution.*

DRF-SC Theorem Specification

```
val drf_sc_theorem :
  is_program_execution:(candidate_execution → bool) →
  Lemma (requires is_drf_program is_program_execution)
    (ensures ∀ exec.
      is_program_execution exec && is_consistent exec ==>
      ∃ sc_exec. is_sc_execution sc_exec &&
        same_observable_behavior exec sc_exec)
```

Corollary 19.15 (Safe Concurrent Programming Model). *For DRF programs, programmers can reason using simple SC semantics, even though the implementation uses relaxed memory.*

19.7 Promising Semantics 2.0 for Sound Relaxed Atomics

Promising Semantics 2.0 for Relaxed-Memory Concurrency

Sources:

- [Kang17]: “A Promising Semantics for Relaxed-Memory Concurrency”
- [Lee20]: “Promising 2.0: Global Optimizations in Relaxed Memory”

Coq Development: <http://sf.snu.ac.kr/promise-concurrency> (~30K lines)

Why This Is Needed: The C11 axiomatic model has a fundamental flaw: it permits “out-of-thin-air” values in programs using relaxed atomics. The Promising Semantics provides an **operational model** that:

1. **Prevents** thin-air values via thread-local certification
2. **Validates** all standard compiler optimizations
3. **Proves** correct compilation to x86-TSO, ARMv8, and POWER
4. **Preserves** DRF-SC and other key theorems

PS 2.0 Improvements (Lee et al. PLDI 2020):

- Enables global value-range analysis soundness
- Enables register promotion for single-threaded locations
- Fixes ARMv8 compilation (no extra fence for relaxed RMWs)
- Replaces universal quantification with **capped memory** construction

Key Insight: Use **operational semantics** (step-by-step execution) with a **promise mechanism**, rather than **axiomatic** (check complete graphs). Promises commit to future writes but must be **certified**—the thread must be able to fulfill the promise against **capped memory** (PS 2.0), not all futures.

Timestamps, Views, and Messages

```
module BrrrMachine.PromisingSemanticsTyped

(* Timestamps for totally ordering writes per location *)
type timestamp = nat

(* Views track which messages a thread has observed *)
type timemap = location → timestamp
type view = { v_pln : timemap; v_rlx : timemap; }

(* Messages represent writes to memory with timestamp intervals *)
type message = {
  msg_loc : location;
```

```

msg_val : value;
msg_from : timestamp; (* Exclusive lower bound *)
msg_to : timestamp; (* Inclusive upper bound = "real" timestamp *)
msg_view : view; (* Release view attached to message *)
}

(* Memory is a set of pairwise disjoint messages *)
type memory = set message

```

Capped Memory Construction (PS 2.0—Lee et al. PLDI 2020)

(The key innovation of PS 2.0: replace universal quantification over ALL future memories with quantification over CAPPED memories. Capped memory represents realistic interference patterns without over-approximation. *)*

Construction:

1. Add RESERVATIONS in gaps between consecutive messages (blocks promises)
2. Add CAP MESSAGES at end of each location's chain (unless promised)

*This enables GLOBAL optimizations (value-range, register promotion) that were unsound under original PS. *)*

```

val capped_memory : memory → set message → memory
let capped_memory mem promises =
  let promised_ends = Set.filter (fun m →
    match max_timestamp_message mem m.msg_loc with
    | Some m_max → m.msg_from = m_max.msg_to
    | None → false) promises in
  let cap_locs = Set.diff (all_locations mem)
    (Set.map (fun m → m.msg_loc) promised_ends) in
  Set.union mem (Set.map (cap_message mem) cap_locs)

```

Thread-Local Certification (PS 2.0)

The Key Innovation: A promise is only allowed if thread-local certification succeeds against capped memory (not all futures).

Original PS (Kang 2017): Certify against ALL future memories

- Too conservative: blocks valid global optimizations
- Problem: future could contain impossible values

PS 2.0 (Lee 2020): Certify against CAPPED memory only

- Capped memory bounds realistic interference
- Enables value-range analysis and register promotion
- Fixes ARMv8 compilation issue

Definition: Thread configuration is **consistent** iff thread can fulfill ALL its promises against capped memory.

Why Certification Prevents Thin-Air (LBd Example):

1. T1 attempts to promise $y := 1$
2. Certification: T1 must execute alone to fulfill promise
3. T1 reads x : only value available is 0 (initial)
4. T1 computes $y := a$ where $a = 0$
5. T1 would write $y := 0$, not $y := 1$
6. Certification **fails**—promise rejected
7. Result: $a = 1$ is **forbidden** (correct!)

Theorem 19.16 (No Thin-Air Values). *Under Promising Semantics, if value $v \neq 0$ is readable at location loc , then some thread wrote it with justified certification.*

Theorem 19.17 (DRF-SC for Promise-Free Machine). *If all reachable states under promise-free execution are race-free, then full machine equals promise-free machine (behaviors identical).*

Theorem 19.18 (Release-Acquire Programs Are Safe). *Programs using only release/acquire synchronization have no thin-air values.*

Theorem 19.19 (Well-Locked Programs Are SC). *If a program is well-locked and SC-reachable states have no races on normal locations, then full machine equals SC machine.*

Theorem 19.20 (Compiler Optimizations Sound). *All standard optimizations that preserve sequential traces are sound under Promising Semantics.*

Theorem 19.21 (Compilation Correctness). *Promising behaviors are subset of hardware behaviors for x86-TSO, POWER, and ARMv8 (PS 2.0 fixes ARMv8—no extra fence for relaxed RMWs).*

Theorem 19.22 (Value-Range Analysis Soundness (PS 2.0)). *Sound value-range analysis ensures all thread states and messages satisfy the analysis in reachable states.*

Theorem 19.23 (Register Promotion Soundness (PS 2.0)). *For single-threaded locations, register promotion preserves Promising behaviors.*

19.8 Cross-Reference Summary for Memory Models

| Model/Section | Description |
|--|--|
| Section 6.5.1–6.5.6 (C11 Axiomatic) | Memory orderings, execution witnesses, coherence axioms. Limitation: Allows thin-air values with relaxed atomics. Safe for: release/acquire only programs. |
| Section 6.5.7 (Promising 2.0) | Operational model with promise mechanism. Thread-local certification prevents thin-air. PS 2.0: Capped memory enables global optimizations. Safe for: all programs including relaxed atomics. |
| Section 12.26 (C11 Theorems) | DRF-SC theorem (valid), Release-acquire soundness (valid), Thin-air warning (critical). |
| When to Use Which Model (Updated for PS 2.0): | |
| Release/acquire only | C11 axiomatic is sufficient |
| Relaxed atomics present | Use PS 2.0 (not original PS) |
| Global optimizations needed | MUST use PS 2.0 |
| Register promotion analysis | MUST use PS 2.0 |
| ARMv8 compilation verification | MUST use PS 2.0 |

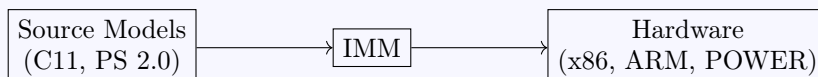
19.9 IMM: Intermediate Memory Model for Compilation Proofs

IMM—Intermediate Memory Model ([Podkopaev19])

Source: Podkopaev, Lahav, Vafeiadis. POPL 2019. “Bridging the Gap between Programming Languages and Hardware Weak Memory Models”

Coq Development: <http://plv.mpi-sws.org/imm/> (~33K lines)

Purpose: IMM serves as a **common target** for source language memory models and a **common source** for hardware models, reducing proof complexity:



Proof Burden WITHOUT IMM: $O(n \times m)$ —each source to each hardware
Proof Burden WITH IMM: $O(n + m)$ —source \rightarrow IMM + IMM \rightarrow hardware
Key Result: First **mechanized** compilation correctness proofs for models weaker than x86-TSO. Corrects error in Kang 2017 POWER proof.

IMM Execution Graph and Consistency

```
module BrrrMachine.IMM

type event_label =
  | LRead : loc:location → val:value → mode:read_mode → event_label
  | LWrite : loc:location → val:value → mode:write_mode → event_label
  | LFence : mode:fence_mode → event_label
  | LInit : loc:location → event_label

type execution_graph = {
  events : set event_id;
  lab : event_id → event_label;
  po : relation event_id;      (* Program order *)
  rf : relation event_id;      (* Reads-from *)
  mo : relation event_id;      (* Modification order *)
  data : relation event_id;    (* Data dependency *)
  addr : relation event_id;    (* Address dependency *)
  ctrl : relation event_id;    (* Control dependency *)
  rmw : relation event_id;     (* RMW pairing *)
}

(* The main acyclicity relation *)
val ar : execution_graph → relation event_id
let ar g = union_all [rfe g; bob g; ppo g; detour g; fre g; coe g]

val imm_consistent : execution_graph → bool
let imm_consistent g =
  wf_execution g &&
  complete_execution g &&
  coherent g &&
  atomic_rmw g &&
  acyclic (ar g) &&
  acyclic (union (ar g) g.rf)
```

Critical Discovery: ARMv8/RISC-V Exclusive Ops

ARMv8/RISC-V exclusive ops are **weaker** than POWER/ARMv7!
Solution: IMM distinguishes “strong” and “normal” RMW compilation.

| Hardware | RMW Strength | Notes |
|----------|---------------|------------------------------------|
| x86-TSO | Strong | LOCK prefix is strong |
| POWER | Strong | LWARX/STWCX is strong |
| ARMv7 | Strong | LDREX/STREX is strong |
| ARMv8 | Normal | LDXR/STXR is WEAKER—needs barrier! |
| RISC-V | Normal | LR/SC is WEAKER—needs barrier! |

Warning: Promising RMWs must compile to “strong” IMM RMWs with extra barrier on ARMv8/RISC-V.

| Source | x86-TSO | ARMv8 | POWER | RISC-V |
|-----------|---------|-------|------------------|---------------|
| rlx read | MOV | LDR | LD | LW |
| rlx write | MOV | STR | ST | SW |
| acq read | MOV | LDAR | LD;CMPW;BC;ISYNC | LW;FENCE R,RW |

| Source | x86-TSO | ARMv8 | POWER | RISC-V |
|--------------|--------------|---------------|-------------|---------------|
| rel write | MOV | STLR | LWSYNC;ST | FENCE RW,W;SW |
| sc read | MOV | LDAR;DMB ISH | SYNC;LD | ... |
| RMW (strong) | LOCK CMPXCHG | LDXR;STXR;DMB | LWARX;STWCX | LR;SC;FENCE |
| RMW (normal) | LOCK CMPXCHG | LDXR;STXR | LWARX;STWCX | LR;SC |

Theorem 19.24 (IMM Compilation Correctness). *1. $IMM \rightarrow Hardware$: For all hardware targets, IMM-consistent executions compile to hardware-consistent executions.*
2. $Promising \rightarrow IMM$: Promising behaviors have corresponding IMM-consistent graphs.
3. $C11 \rightarrow IMM$: C11-consistent executions are IMM-consistent.
4. End-to-End: $Promising \rightarrow Hardware$ via IMM.

20. Coeffect Systems: Context Requirements

Paper: [Petricek14]—Petricek, Orchard, Mycroft, ICFP 2014

Effect/Coeffect Duality—Two Sides of the Same Coin

| Effects (Section 6.1) | Coeffects (This Section) |
|---|---|
| What computation produces | What computation requires |
| Monadic structure ($T^\varepsilon A$) | Comonadic structure ($D^r A$) |
| Effect rows: $\langle E \mid \varepsilon \rangle$ | Coeffect annotations: $\Gamma @ r$ |
| Examples: Write, Throw[], IO | Examples: liveness, usage, capabilities |
| return : $A \rightarrow M^\varepsilon A$ | extract : $D^r A \rightarrow A$ |

Typing with Both:

$$\Gamma @ r \vdash e : \tau \ \& \ \varepsilon$$

“Under context Γ with requirements r , e has type τ and effects ε ”

Why This Matters for Brrr-Machine:

1. Effects tell us what code **does** (mutations, I/O, exceptions)
2. Coeffects tell us what code **needs** (liveness, resources, capabilities)
3. Complete analysis requires **both** for full specification

20.1 Flat vs Structural Coeffects

Definition 20.1 (Flat Coeffects—Whole-Context Properties). **Shape:** Single coeffect value for entire context.

Example: $\Gamma @ \{\text{network}, \text{gps}\} \vdash e : \tau$

Meaning: Expression e requires network AND GPS capabilities.

Use cases:

- Platform/capability requirements
- Resource access requirements (files, database)
- Runtime feature detection
- Cross-platform compatibility checking

Definition 20.2 (Structural Coeffects—Per-Variable Properties). **Shape:** Coeffect annotation per variable in context.

Example: $(x : \text{Int}@2, y : \text{Bool}@1) \vdash e : \tau$

Meaning: Expression e uses x twice and y once (bounded linear logic).

Use cases:

- Liveness analysis (is variable accessed?)

- Usage counting (how many times is variable used?)
- Dataflow dependencies (which past values needed?)
- Linear resource management

Definition 20.3 (Coefficient Algebraic Structure). Both shapes use:

- $+$ (addition) : Sequencing—using both contexts
- \times (multiplication) : Nesting—context for inner computation
- \sqcap (meet) : Branching—required by at least one branch
- 1 (unit) : Empty requirement
- 0 (zero) : Unused (for structural coefficients)

20.2 Liveness as a Structural Coeffect

Liveness Analysis via Coeffects

Liveness is naturally a **coeffect**: it describes which variables a computation **requires** from its context.

Definition 20.4 (Liveness Coeffect Algebra (Semiring Structure)). $L = \{\text{LDead}, \text{LLive}\}$

| | |
|---|---|
| $\text{LDead} + \text{LDead} = \text{LDead}$ | (neither uses \rightarrow not used) |
| $\text{LDead} + \text{LLive} = \text{LLive}$ | (one uses \rightarrow used) |
| $\text{LLive} + \text{LLive} = \text{LLive}$ | (both use \rightarrow used) |
| $\text{LDead} \times _ = \text{LDead}$ | (not used in outer \rightarrow dead) |
| $\text{LLive} \times l = l$ | (used in outer \rightarrow propagate inner) |
| $\text{LDead} \sqcap \text{LDead} = \text{LDead}$ | (dead in both branches \rightarrow dead) |
| $_ \sqcap _ = \text{LLive}$ | (live in either branch \rightarrow live) |

Connection to IFDS

Liveness is a GEN/KILL problem, hence IFDS-applicable. Coeffect formulation provides:

1. Compositionality via semiring structure
2. Typing guarantee: well-typed = correct liveness
3. Connection to linear logic via usage bounds

Definition 20.5 (Liveness Inference Rules).

- VAR: x is live at $(x : \tau @ \text{LLive})$
- APP: $f e$ has liveness $(\text{liveness}(f) + \text{liveness}(e))$
- LAMBDA: $\lambda x. e$ has liveness $(\text{liveness}(e) \setminus x)$
- IF: **if** c **then** t **else** e has liveness $(c + (t \sqcap e))$

20.3 Usage Coeffects and Linear Types

Definition 20.6 (Bounded Linear Types via Coeffects (Petricek 2014, Section 2.1)). Track how many times each variable is used:

- UZero : Never used
- UOne : Used exactly once
- $\text{UBounded } n$: Used at most n times
- UMany : Used arbitrarily many times

Definition 20.7 (Usage Coefficient Algebra).

$$\begin{aligned}
\text{UZero} + u &= u \\
\text{UOne} + \text{UOne} &= \text{UBounded } 2 \\
\text{UBounded } n + \text{UBounded } m &= \text{UBounded } (n + m) \\
\text{UMany} + _ &= \text{UMany} \\
\text{UZero} \times _ &= \text{UZero} \\
\text{UOne} \times u &= u \\
\text{UBounded } n \times \text{UBounded } m &= \text{UBounded } (n \times m)
\end{aligned}$$

Connection to Ownership

Rust-style ownership maps to usage coeffects:

| | |
|------------------|-------------------------------------|
| Move semantics | usage ≤ 1 (affine) |
| Borrow semantics | usage ≥ 0 (unrestricted reads) |
| Mut borrow | usage = 1 (linear) |

The semiring structure gives composition:

- x moved $+ x$ moved = error (usage > 1 violates affine)
- x borrowed $+ x$ borrowed = ok (unrestricted composes)

20.4 Platform Capability Coeffects

Definition 20.8 (Flat Coeffects for Cross-Platform Analysis). Track which platform capabilities code requires:

CapNetwork : Requires network access
 CapFileSystem : Requires filesystem access
 CapGPS : Requires location services
 CapDatabase : Requires database connection
 $\text{CapFFI}(lang)$: Requires FFI to specific language

Definition 20.9 (Capability Algebra). Capability algebra is simple set union:

| | |
|---------------------------------------|----------------------------------|
| $c_{\text{zero}} = \{\}$ | (no requirements) |
| $c_{\text{one}} = \{\}$ | (unit doesn't add requirements) |
| $c_{\text{plus}} = \text{Set.union}$ | (sequence needs both) |
| $c_{\text{times}} = \text{Set.union}$ | (nesting preserves requirements) |
| $c_{\text{meet}} = \text{Set.union}$ | (either branch might run) |

Applications

1. Cross-platform compatibility checking
2. Graceful degradation (fallback when capability missing)
3. Permission documentation generation
4. FFI boundary analysis (Section 6.4: C FFI limitations)

20.5 F* Specification: Coeffect System

The following F* specification formalizes the coeffect system. The key insight from Petricek et al. is that coeffects form an algebraic structure (a semiring) that enables compositional reasoning about context requirements. Each coeffect instance (liveness, usage, capabilities) instantiates this algebra with domain-specific operations:

- **coeffect_algebra**: The type class defining the semiring interface. The operations `c_plus` (sequential composition), `c_times` (nesting), and `c_meet` (branching) correspond directly to the algebraic operations from Section 20.1.
- **Liveness instance**: A simple two-element semiring tracking whether variables are live or dead. This connects directly to IFDS-based liveness analysis (Part IV).
- **Usage instance**: Tracks bounded usage counts, connecting to linear types and ownership (Part VII).
- **Capability instance**: Flat coeffects using set union, for platform/resource tracking.

Coeffect System Specification

```
module BrrrMachine.Coeffects

(* Coffect algebra - Operations for composing coeffects *)
class coeffect_algebra (c : Type) = {
  c_zero : c;                                (* No requirement *)
  c_one : c;                                  (* Basic requirement *)
  c_plus : c → c → c;                         (* Sequential: both needed *)
  c_times : c → c → c;                       (* Nesting: inner within outer *)
  c_meet : c → c → c;                        (* Branching: at least one needs it *)

  (* Semiring laws *)
  plus_comm : ∀ a b. c_plus a b = c_plus b a;
  plus_assoc : ∀ a b c. c_plus (c_plus a b) c = c_plus a (c_plus b c);
  times_assoc : ∀ a b c. c_times (c_times a b) c = c_times a (c_times b c);
  zero_plus : ∀ a. c_plus c_zero a = a;
  one_times : ∀ a. c_times c_one a = a;
}

```

Liveness Coffect Instance

```
type liveness = | LDead | LLive

instance liveness_algebra : coeffect_algebra liveness = {
  c_zero = LDead;
  c_one = LLive;
  c_plus = fun l1 l2 → if l1 = LDead && l2 = LDead then LDead else LLive;
  c_times = fun l1 l2 → if l1 = LDead then LDead else l2;
  c_meet = fun l1 l2 → if l1 = LDead && l2 = LDead then LDead else LLive;
  ...
}

type liveness_coeffect = map var_id liveness

```

Usage Coffect Instance

```
type usage_bound =
| UZero : usage_bound
| UOne : usage_bound
| UBounded : n:nat → usage_bound
| UMany : usage_bound

instance usage_algebra : coeffect_algebra usage_bound = {
  c_zero = UZero;
  c_one = UOne;
  c_plus = fun u1 u2 → match u1, u2 with
  | UZero, u → u
  | u, UZero → u
  | UOne, UOne → UBounded 2
  | UOne, UBounded n → UBounded (n + 1)
  | UBounded n, UOne → UBounded (n + 1)
  | UBounded n, UBounded m → UBounded (n + m)
  | UMany, _ | _, UMany → UMany;
  c_times = fun u1 u2 → match u1, u2 with
  | UZero, _ | _, UZero → UZero

```

```

    | UOne, u → u
    | u, UOne → u
    | UBounded n, UBounded m → UBounded (n * m)
    | UMany, _ | _, UMany → UMany;
c_meet = fun u1 u2 → max_usage u1 u2;
...
}

```

```

type usage_coeffect = map var_id usage_bound

```

Capability Coeffect Instance

```

type capability =
  | CapNetwork | CapFileSystem | CapGPS | CapCamera
  | CapDatabase : db_type:string → capability
  | CapFFI : language:string → capability
  | CapPlatform : platform:string → min_version:nat → capability

type capability_set = set capability

instance capability_algebra : coeffect_algebra capability_set = {
  c_zero = Set.empty;
  c_one = Set.empty;
  c_plus = Set.union;
  c_times = Set.union;
  c_meet = Set.union;  (* Either branch might run, need both capabilities *)
  ...
}

```

Combined Effect/Coeffect Typing

```

type full_type_judgment = {
  context : typing_context;
  coeffects : coeffect_annotation;  (* What it requires *)
  effects : effect_row;             (* What it produces - from Section 6.1 *)
  result_type : ir_type;
}

(* Well-typed expression with both effects and coeffects *)
val has_full_type :
  typing_context →
  coeffect_annotation →
  ir_expr →
  ir_type →
  effect_row →
  bool

(* Coeffect safety: no access beyond annotations *)
val coeffect_safety :
  γ:typing_context →
  r:coeffect_annotation →
  e:ir_expr →
  τ:ir_type →
  eff:effect_row →
  has_full_type γ r e τ eff →
  Lemma (accessed_variables e 'subset' live_in_coeffect r)

```

Part VII

Ownership and Resources

Historical Foundations: Ownership and Typestate Origins

Strom & Yemini 1986 — Original Typestate: The foundational paper introducing typestate as “a refinement of type.” While *type* determines operations ever permitted on an object, *typestate* determines the subset permitted in a particular context.

Key insights that persist to modern systems:

- State machine semantics: Objects have typestates forming lower semilattice
- Transitions via operations: Each op has pre and post typestate conditions
- Compile-time tracking: Typestate as static invariant at each point
- Greatest lower bound at merge: Conservative approximation at branches
- Finalization guarantee: All objects returned to bottom state at exit

Citation chain: Strom 1986 → DeLine/Fahndrich Vault 2004 → Bierhoff 2007 → Prusti pledges (Astrauskas 2022) → Modern Rust unsafe

Boyapati, Lee, Rinard 2003 — Ownership Types for Safe Programming: Introduced *owner-as-dominator* discipline: “All paths in the heap from root to object x must pass through x ’s owner.”

Key contributions:

- Ownership types for data race and deadlock prevention
- Parameterized classes by owners (precursor to Rust generics + lifetime)
- Lock level partial ordering for deadlock freedom
- Tree-based lock ordering for nested data structures
- Unique pointers combined with ownership (first system to do so)

Citation chain: Boyapati 2003 → Cyclone (Grossman 2006) → Rust ownership → RustBelt (Jung 2018) → RefinedRust (Gaher 2024)

Modern Synthesis: Rust’s ownership = Boyapati’s owner discipline + Girard’s linearity. Prusti pledges = Strom’s typestate + separation logic permissions. Iris cameras = Semantic model for Boyapati’s ownership hierarchy.

Two Ownership Models: Cameras vs Access Permissions

Cameras (Section 21) — From Iris/RustBelt:

- Use for: Rust-like ownership, move semantics, borrow checking
- Strengths: Expressive, handles complex aliasing patterns
- Complexity: Step-indexed, requires frame preservation proofs
- Best for: Systems languages (Rust, C++ with ownership annotations)

Access Permissions (Section 22.1) — From Bierhoff:

- Use for: Java-like typestate, resource protocols, API contracts
- Strengths: Decidable, practical for static analysis tools
- Complexity: 5 fixed permission kinds, simpler composition rules
- Best for: Managed languages (Java, C#, Go), protocol verification

Guidance: Start with Access Permissions for decidability. Upgrade to Cameras only when permission splitting is insufficient.

Ownership as Coeffects (Petricek 2014 Connection)

Ownership and linear types connect to usage coeffects (Section ??):

| Ownership Style | Usage Coefficient | Constraint |
|--------------------------|----------------------|------------------------|
| Rust move semantics | UBounded 1 (affine) | usage ≤ 1 |
| Rust borrow (&T) | UMany (unrestricted) | unlimited reads |
| Rust mut borrow (&mut T) | UOne (linear) | usage = 1 exactly |
| Linear [Girard87] | UOne | must use exactly once |
| Relevant | UAtLeast 1 | must use at least once |

Composition via Semiring:

- x moved + x moved = error (UBounded 1 + UBounded 1 = UBounded 2 > 1)
- x borrowed + x borrowed = ok (UMany + UMany = UMany)

Coefficients provide compositional reasoning; Coefficients provide semantic model.

Substructural Type Systems: Linear, Affine, and Relevant Types

Source: Girard 1987 (Linear Logic), Walker 2005 (Substructural Type Systems)

Substructural type systems restrict the *structural rules* of the lambda calculus:

- **Contraction:** Can a variable be used more than once?
- **Weakening:** Can a variable be ignored (not used)?

The Substructural Hierarchy:

| Type System | Contraction | Weakening | Property |
|--------------|-------------|-----------|--|
| Unrestricted | Yes | Yes | Normal types (can copy/discard) |
| Affine | No | Yes | Use <i>at most once</i> (Rust ownership) |
| Linear | No | No | Use <i>exactly once</i> (Girard) |
| Relevant | Yes | No | Use <i>at least once</i> |

Why This Matters for Resource Management:

- **Linear types** guarantee resources are neither leaked nor double-used. A file handle with linear type *must* be closed exactly once.
- **Affine types** allow discarding but prevent double-use. Rust’s ownership is affine: values can be dropped implicitly but cannot be used after move.
- **Relevant types** ensure resources are used at least once, preventing “acquire but never use” patterns.

Rust’s Borrow Checker as Affine + Borrowing:

1. Owned values are **affine**: use at most once (move semantics)
2. Shared borrows (&T) are **unrestricted**: can copy freely
3. Mutable borrows (&mut T) are **linear during lifetime**: exactly one active mutable borrow
4. The borrow checker enforces these constraints statically via lifetime analysis

Connection to Separation Logic: Linear types and separation logic share the “resources cannot be duplicated” principle:

- Linear $A \otimes B$ (tensor) \sim Separation $A * B$ (separating conjunction)
- Linear $A \multimap B$ (lollipop) \sim “consume A to produce B ”
- Both enforce disjointness of resource usage

Tension Resolution: Separation Logic vs GC Languages

Reynolds’ Warning [Reynolds02]: “GC interaction is problematic” for separation logic.

- Separation logic assumes explicit deallocation
- GC may collect memory mid-analysis (unpredictable timing)
- Cannot model “disconnected garbage” in standard separation logic

Resolution by Language Class:

GC Languages (Python, Go, Java, JavaScript):

- Use separation logic for *resources only* (files, connections, locks)
- Not for memory—grant MemSafe axiom from runtime
- Track ownership for API resources, not heap cells
- Frame Rule still applies to resource footprints

Manual Memory (C, C++, unsafe Rust):

- Full separation logic applies
- Frame rule essential for scalability
- Shape Analysis uses separating conjunction

21. Resource Algebras from Iris

Papers: [Girard87], [Reynolds02], [Boyapati03], [Jung18]

Historical Note: From Boyapati 2003 to Iris

Boyapati, Lee, Rinard (OOPSLA 2002) introduced ownership types for safe concurrent programming in Java. Their key insight: ownership encapsulation enables modular reasoning about locking.

Owner-as-Dominator: All heap paths to object x pass through x 's owner.

- Owner's lock protects all transitively owned objects
- Parameterized classes: different instances can have different owners
- `thisThread` owner: thread-local objects need no synchronization
- `self` owner: objects protected by their own lock

Evolution to modern systems:

| | | |
|----------------|---|-----------------------------------|
| Boyapati 2003: | <code>Owner<T></code> | → explicit owner parameter |
| Cyclone 2006: | <code>region<T></code> | → region-based memory management |
| Rust 2015: | <code>T, &'a T, &'a mut T</code> | → lifetime-annotated borrows |
| RustBelt 2018: | <code>own(t, v), shr(κ, t, l)</code> | → semantic model via Iris cameras |

Iris cameras provide the semantic foundation for ownership:

- Exclusive camera $\text{Ex}(T) \sim$ Boyapati's `thisThread/self` owner
- Fractional camera $\text{Frac} \sim$ Boyapati's shared read access
- Agreement camera $\text{Ag}(T) \sim$ Boyapati's readonly owner
- Authoritative camera $\text{Auth} \sim$ Boyapati's ownership transfer

The brrr-machine uses resource algebras (cameras) from Iris to track ownership. This provides a unified framework for reasoning about:

- Memory ownership (Rust-style)
- Reference counting
- Shared state with synchronization
- File handles and connections

21.1 The Camera Abstraction

Definition 21.1 (Camera (from Iris)). A *camera* is a step-indexed partial commutative monoid (PCM):

$$\text{Camera } M = (M, V : M \rightarrow \text{Prop}, |\cdot| : M \rightarrow M?, (\cdot) : M \times M \rightarrow M)$$

where:

- M is the carrier set (possible ownership states)

- V is the validity predicate (which states are coherent)
- $|\cdot|$ is the core (shareable part)
- (\cdot) is composition (combining ownerships)

Axioms:

$$\begin{aligned}
\text{ASSOC: } & (a \cdot b) \cdot c = a \cdot (b \cdot c) \\
\text{COMM: } & a \cdot b = b \cdot a \\
\text{CORE-ID: } & |a| \text{ defined} \implies |a| \cdot a = a \\
\text{VALID-OP: } & V(a \cdot b) \implies V(a)
\end{aligned}$$

Cameras capture different ownership disciplines:

Exclusive Camera $\text{Ex}(T)$: Elements are T or “invalid”. Composition always invalid (cannot share exclusive). Use: unique ownership, like `Box<T>` in Rust.

Agreement Camera $\text{Ag}(T)$: Elements are sets of T values. Composition is union (must agree on values). Valid if all elements in set are equal. Use: read-only shared state.

Fractional Camera Frac : Elements are rationals in $(0, 1]$. Composition is addition (must sum to ≤ 1). Use: fractional permissions, shared borrows.

Authoritative Camera $\text{Auth}(M)$: Elements are (full?, fragment) pairs. Full owns the “truth”, fragments are views. Use: ghost state, invariants.

Connection to Girard 1987 Exponentials: Cameras implement Girard’s exponential modality $(!A, ?A)$ via resource structure:

- Exclusive $\text{Ex}(T) \sim \text{bare } A$ (linear: exactly-once use, can free)
- Fractional $\text{Frac} \sim !A$ (of-course: shareable, copyable borrows)
- Agreement $\text{Ag}(T) \sim !A$ readonly (unlimited read-only copies)

The exponential $!A$ in linear logic makes A freely copyable and discardable. Cameras achieve this via the “core” operation: $|a|$ extracts the shareable part.

- $|\text{Ex}(v)| = \perp$ (exclusive has no shareable part)
- $|\text{Frac}(q)| = \text{Frac}(q)$ (fractions are their own core—duplicable)
- $|\text{Ag}(s)| = \text{Ag}(s)$ (agreement is duplicable)

21.1.1 Ghost State and Invariants

Ghost state enables strong specifications for concurrent programs without runtime cost:

Ghost State Rules:

$$\begin{aligned}
\text{ALLOCATION: } & \vdash \Rightarrow \exists \gamma. \text{own}(\gamma, a) \\
\text{INTERACTION: } & \text{own}(\gamma, a) * \text{own}(\gamma, b) \vdash \text{valid}(a \cdot b) \\
\text{UPDATE: } & \text{own}(\gamma, a) \vdash \Rightarrow \text{own}(\gamma, b) \quad [\text{frame-preserving}]
\end{aligned}$$

Invariant Protocol: $\text{inv } N \ P$ is duplicable—multiple threads can hold reference. But can only be opened during atomic operations. Must close before atomic operation completes. This enables safe sharing of logical resources.

21.1.2 View Shifts and Step-Indexing

Definition 21.2 (View Shift). The update modality $\Rightarrow P$ asserts ownership of resources that can be updated to satisfy P . Key primitive for ghost state manipulation.

Rules:

$$\begin{aligned}
& \Rightarrow \Rightarrow P \Leftrightarrow \Rightarrow P \quad (\text{idempotent}) \\
& P \vdash \Rightarrow P \quad (\text{identity}) \\
& (\Rightarrow P) * Q \vdash \Rightarrow (P * Q) \quad (\text{frame preserving}) \\
& \text{own}(\gamma, a) \vdash \Rightarrow \text{own}(\gamma, b) \quad (\text{when } a \rightsquigarrow b \text{ frame-preserving})
\end{aligned}$$

Step-Indexing (why cameras are step-indexed):

- **Problem:** Higher-order ghost state creates circularity. Propositions can contain ghost state, ghost state validates propositions.
- **Solution:** Stratify by “steps remaining” (approximation index n).
 - At step 0: all propositions equivalent (base case)
 - At step $n + 1$: can distinguish based on n -step behavior
 - Cameras are step-indexed: $\text{valid}_n(a)$ means valid for n more steps
- **Enables:** Impredicative invariants, recursive protocols, later modality (\triangleright) for guarded recursion

21.2 Separation Logic for Memory

Paper: [Reynolds02]

Definition 21.3 (Separating Conjunction). $P * Q$ means: P and Q hold on *disjoint* portions of memory.

Theorem 21.4 (Frame Rule).

$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}}$$

If C only needs P , then R is preserved. This enables local reasoning.

Definition 21.5 (Points-To Assertion). $x \mapsto v$ means: x points to location containing v , and we have *exclusive* access.

Example (swap):

$$\begin{aligned} &\{x \mapsto a * y \mapsto b\} \\ &t := *x; \\ &*x := *y; \\ &*y := t \\ &\{x \mapsto b * y \mapsto a\} \end{aligned}$$

The frame rule lets us add any R that does not mention x, y .

Why this matters for brrr-machine: We can express memory safety as separation logic formulas:

- No use-after-free: freed locations removed from valid heap
- No races: concurrent accesses require disjoint permissions
- Ownership transfer: moving $x \mapsto v$ from one context to another

GC Language Caveat: For GC languages (Python, Java, Go, JS), use separation logic for *resources* (files, connections) only—not for memory. Memory safety is granted axiomatically by the runtime.

21.3 F* Specification: Ownership System

The following F* specifications formalize the ownership concepts from Iris cameras in a form suitable for static analysis implementation. These type definitions provide the foundation for tracking resource ownership, permissions, and state transitions throughout the analysis.

Note: F* code using numeric multiplication requires `open FStar.Mul` at the module level. All code blocks assume a module preamble with appropriate imports.

21.3.1 Resource Algebra Typeclass

The `resource_algebra` typeclass captures the essential structure of Iris cameras: a partial commutative monoid with a validity predicate and a core operation for extracting shareable components. The four axioms ensure compositional reasoning about resource ownership.

Resource Algebra (Simplified Camera)

```

(* Option bind for composing optional values *)
let option_bind #a #b (x : option a) (f : a → option b) : option b =
  match x with
  | Some v → f v
  | None → None

(* Resource algebra operations as a record type *)
noeq type resource_algebra_ops (a : Type) = {
  (* Composition -- partial, may fail *)
  compose : a → a → option a;
  (* Validity predicate *)
  valid : a → bool;
  (* Core -- the shareable/duplicable part *)
  core : a → option a;
}

(* Axiom predicates -- instances must satisfy these *)
let ra_compose_comm #a (ops : resource_algebra_ops a) =
  ∀ (x y : a). ops.compose x y == ops.compose y x

let ra_compose_assoc #a (ops : resource_algebra_ops a) =
  ∀ (x y z : a).
    option_bind (ops.compose x y) (fun xy → ops.compose xy z) ==
    option_bind (ops.compose y z) (fun yz → ops.compose x yz)

let ra_valid_compose #a (ops : resource_algebra_ops a) =
  ∀ (x y r : a). ops.compose x y == Some r ==> ops.valid r ==> ops.valid x

let ra_core_id #a (ops : resource_algebra_ops a) =
  ∀ (x : a) (c : a). ops.core x == Some c ==> ops.compose c x == Some x

(* A valid resource algebra satisfies all axioms *)
type resource_algebra (a : Type) =
  ops:(resource_algebra_ops a){
    ra_compose_comm ops ∧ ra_compose_assoc ops ∧
    ra_valid_compose ops ∧ ra_core_id ops
  }

```

The `exclusive` type models unique ownership—the “Box” pattern from Rust. When you own something exclusively, composition with any other claim invalidates both. This captures Rust’s fundamental rule: a `Box<T>` cannot be aliased. The `core` returns `None` because exclusive resources have no shareable fragment.

Exclusive Ownership — Like Rust’s `Box<T>`

```

type exclusive (a : Type) =
  | ExOwned : value:a → exclusive a
  | ExInvalid : exclusive a

(* Operations for exclusive resources *)
let exclusive_ops (#a : Type) : resource_algebra_ops (exclusive a) = {
  compose = (fun x y →
    (* Exclusive resources cannot be composed *)
    Some ExInvalid);
  valid = (fun x →
    match x with
    | ExOwned _ → true
    | ExInvalid → false);
  core = (fun x → None); (* No shareable part *)
}

(* NOTE: To form a valid resource_algebra, one must prove the axioms hold *)

```

Fractional permissions enable shared borrows—Rust’s `&T` references. Instead of binary own-

ership, we track fractions: a full ownership is 1, and multiple shared references each hold a fraction summing to at most 1. Key properties:

- Fractions can be *composed* (combined) as long as the sum stays ≤ 1
- Unlike exclusive, fractions have a **core**—they are duplicable (within limits)
- When all fractions reassemble to 1, we can recover full ownership

This directly models Rust’s rule: multiple `&T` or one `&mut T`, never both.

Fractional Permissions — For Shared Borrows

```
type fraction = { num : nat; denom : pos } (* num/denom in (0,1] *)

let frac_add (f1 f2 : fraction) : option fraction =
  let num = f1.num * f2.denom + f2.num * f1.denom in
  let denom = f1.denom * f2.denom in
  if num ≤ denom then Some { num; denom }
  else None (* Sum > 1, invalid *)

type fractional (a : Type) =
  | FracOwn : frac:fraction → value:a → fractional a
  | FracInvalid : fractional a

(* Operations for fractional resources *)
let fractional_ops (#a : Type) : resource_algebra_ops (fractional a) = {
  compose = (fun x y →
    match x, y with
    | FracOwn f1 v1, FracOwn f2 v2 →
      if v1 = v2 then
        match frac_add f1 f2 with
        | Some f → Some (FracOwn f v1)
        | None → Some FracInvalid
      else Some FracInvalid (* Disagreement on value *)
    | _, _ → Some FracInvalid);
  valid = (fun x →
    match x with
    | FracOwn f v → f.num > 0 && f.num ≤ f.denom
    | FracInvalid → false);
  core = (fun x →
    match x with
    | FracOwn f v → Some (FracOwn f v) (* Fractions are duplicable *)
    | _ → None);
}
```

(* NOTE: To form a valid resource_algebra, one must prove the axioms hold *)

The `ownership_state` type below models the lifecycle of a memory location through the lens of Rust-style ownership semantics. Each location transitions through states from allocation to deallocation, with borrowing as intermediate states. The `valid_transition` function encodes which state changes are legal, directly corresponding to Rust’s borrow checker rules.

Key type signatures:

- `OsOwned owner`: Location owned by scope `owner`, can be moved or freed
- `OsBorrowedShared borrowers`: Shared borrows active, read-only access by multiple scopes
- `OsBorrowedMut borrower`: Exclusive mutable borrow, single scope has write access
- `OsMoved`: Value moved out, location is “poisoned” (use-after-move error if accessed)

Ownership State Machine — Per-Location Tracking

```
type ownership_state =
  | OsUnallocated : ownership_state
  | OsOwned : owner:scope_id → ownership_state
  | OsBorrowedShared : borrowers:set scope_id → ownership_state
  | OsBorrowedMut : borrower:scope_id → ownership_state
  | OsMoved : ownership_state
```

```

| OsFreed : ownership_state

(* Valid transitions *)
let valid_transition (from to_ : ownership_state) : bool =
  match from, to_ with
  | OsUnallocated, OsOwned _ → true          (* Allocation *)
  | OsOwned _, OsOwned _ → true              (* Transfer *)
  | OsOwned o, OsBorrowedShared bs →
    Set.for_all (fun b → b ≠ o) bs          (* Borrow *)
  | OsOwned o, OsBorrowedMut b →
    b ≠ o                                   (* Mut borrow *)
  | OsBorrowedShared bs, OsOwned _ →
    Set.is_empty bs                         (* Borrows ended *)
  | OsBorrowedMut _, OsOwned _ → true        (* Mut borrow ended *)
  | OsOwned _, OsMoved → true                (* Move *)
  | OsOwned _, OsFreed → true                (* Deallocation *)
  | _, _ → false

```

21.4 Capability Multiplicities

Paper: Crary, Walker, Morrisett 1999 (Typed Memory Management)

Critical Distinction: Lifecycle vs Aliasing

The `ownership_state` type above tracks **lifecycle**:

Unallocated → Owned → Borrowed → Moved → Freed

But this misses **aliasing information**: “Is this pointer unique, or might other pointers exist?”

Crary’s insight: Only *unique* capabilities can safely free memory. An “owned” but aliased pointer *cannot* safely free—dangling refs remain.

Definition 21.6 (Multiplicity (from Crary 1999)).

$\text{MUnique}(\{r^1\})$ Provably no aliases exist. Can free.

$\text{MDup}(\{r^+\})$ May have aliases. Can only access, not free.

This is **stronger** than “owned”:

- Owned + Aliased \Rightarrow Cannot free safely
- Unique \Rightarrow Can free (no dangling pointers possible)

Capability Algebra:

Join rules: $\{r^1\} + \{r^1\} = \text{Invalid}$ (can’t have two uniques!)

$\{r^1\} + \{r^+\} = \{r^+\}$ (unique + dup = dup)

$\{r^+\} + \{r^+\} = \{r^+\}$ (dup + dup = dup)

Strip (lending): $\text{strip}(\{r^1\}) = \{r^+\}$ (lend as duplicatable)

$\text{strip}(\{r^+\}) = \{r^+\}$ (already dup)

Bounded Quantification for Capability Recovery:

- **Problem:** How to get uniqueness back after lending?
- **Solution:** $\forall[\varepsilon \leq \{r^+\}]. (\varepsilon, \dots, (\varepsilon, \dots) \rightarrow 0) \rightarrow 0$
- Caller instantiates $\varepsilon = \{r^1\}$ (passes unique)
- Callee sees only $\varepsilon \leq \{r^+\}$ (can’t free, can only access)
- Continuation gets $\{r^1\}$ back!
- This models Rust’s borrow-and-return pattern.

Theorem 21.7 (Complete Collection (Crary 1999)). *Well-typed terminating programs return all memory to the system.*

If $\vdash (M, e)$ well-typed, then either: $\begin{cases} (M, e) \text{ diverges,} & \text{or} \\ (M, e) \rightarrow^* (\emptyset, \text{halt } i) & \text{heap is empty at termination} \end{cases}$

Proof sketch:

1. Typing rule for **halt** requires $\text{capability} = \emptyset$ (empty)
2. Empty capability means no regions owned
3. Subject reduction preserves typing
4. Progress ensures termination reaches **halt**
5. Therefore: terminating programs have empty heap

This is leak-freedom by construction.

22. Access Permissions and Typestate

22.1 Access Permissions (Bierhoff)

Paper: Bierhoff 2007 (Modular Typestate Checking)

Historical Foundation: Strom & Yemini 1986 (Original Typestate)

Historical Note: From Strom 1986 to Modern Typestate

Strom & Yemini 1986 — Original Typestate Concept: “Typestate is a refinement of the concept of type. Whereas the type of a data object determines the set of operations ever permitted on the object, typestate determines the subset of these operations which is permitted in a particular context.”

Core Principles (still valid today):

1. State machine semantics: Each type has associated typestate lattice
2. Preconditions: Each operation requires specific typestate
3. Postconditions: Operations produce new typestate (possibly per outcome)
4. Static tracking: Typestate is compile-time invariant at each point
5. Merge rule: At branches, take greatest lower bound of typestates
6. Finalization: All objects must reach bottom typestate at exit

Key Evolution:

| | |
|----------------|---|
| Strom 1986: | Typestate as static invariant, no aliasing handling |
| Vault 2004: | Adoption (permanent) and Focus (temporary) for aliasing |
| Bierhoff 2007: | 5 permission kinds integrate aliasing with typestate |

Modern typestate = Strom’s state machines + Bierhoff’s permission aliasing

Access Permissions: Decidable Alternative to Iris Cameras

Iris cameras (Section 21) are **general** but **undecidable**. Bierhoff’s 5 permission kinds are **specific** but **decidable**.

Five Permission Kinds:

| Permission | Description |
|-----------------------------|--|
| $\text{unique}(r, n, g)$ | Full ownership, can free, no aliases exist |
| $\text{full}(r, n, g)$ | Full access, may have pure aliases |
| $\text{share}(r, n, g, k)$ | Shared mutable, fraction k |
| $\text{immutable}(r, n, g)$ | Read-only, unlimited copies |
| $\text{pure}(r, n, g)$ | Read-only, guarantees no modification |

Key Insight: Permissions bound to state guarantees via root nodes. A permission can be confined to a state subtree, guaranteeing the object stays in that region of the typestate.

The Five Permission Kinds

```
type access_permission =
  | APUnique   : root:root_node → node:state_node → guarantee:state_node
                → access_permission
    (* Full ownership, can free, guaranteed no aliases *)
  | APFull    : root:root_node → node:state_node → guarantee:state_node
                → access_permission
    (* Full access, may have pure aliases observing *)
  | APShare   : root:root_node → node:state_node → guarantee:state_node
                → frac:fraction → access_permission
    (* Shared mutable access with fraction *)
  | APImmutable : root:root_node → node:state_node → guarantee:state_node
                → access_permission
    (* Read-only, can duplicate freely *)
  | APPure    : root:root_node → node:state_node → guarantee:state_node
                → access_permission
    (* Read-only, guarantees object won't be modified *)
```

Permission splitting and joining form the core algebra of access control. These operations implement Rust's borrow semantics at the permission level:

- `split_unique`: Creates a mutable reference from an owned value (like `&mut` from `Box`)
- `split_full`: Creates multiple shared references from full access
- `join_shares`: Recombines all shared references back to full access
- `join_full_pure`: Restores unique ownership when all borrows end

These operations are *linear*—permissions cannot be duplicated or discarded, only split and rejoined.

Permission Splitting and Joining

```
(* Split unique into full + pure *)
val split_unique :
  APUnique r n g → (APFull r n g * APPure r n g)

(* Split full into share + share *)
val split_full :
  APFull r n g → (APShare r n g (half_frac) * APShare r n g (half_frac))

(* Join shares back to full *)
val join_shares :
  APShare r n g f1 → APShare r n g f2 →
  requires (add_fractions f1 f2 = full_frac) →
  APFull r n g

(* Join full + pure back to unique *)
val join_full_pure :
  APFull r n g → APPure r n g → APUnique r n g
```

22.1.1 Per-Node Fraction Functions (Bierhoff 2007 Extended)

Fraction functions generalize simple fractions to complex objects with heterogeneous access patterns. Consider a database connection object: you might have full access to the metadata but only partial (shared) access to the result cursor. The fraction function maps each state node to its permission fraction, enabling fine-grained access control over object substructure. This extends Bierhoff's typestate checking to handle partially-locked composite objects.

Fraction Functions

```
(* Complex objects may have different access levels at different states.
  A "partially open" file has full access to metadata, partial to content. *)
```

```

type fraction_function = state_node → fraction

val frac_fn_split : fraction_function → (fraction_function * fraction_function)
let frac_fn_split ff =
  (fun n → half_of (ff n), fun n → half_of (ff n))

val frac_fn_join : fraction_function → fraction_function → option fraction_function
let frac_fn_join ff1 ff2 =
  (* Sum must not exceed 1 at any node *)
  let can_join = ∀ (fun n → can_add (ff1 n) (ff2 n)) in
  if can_join then Some (fun n → add_frac (ff1 n) (ff2 n))
  else None (* Sum exceeds 1 at some node *)

```

22.1.2 Temporary State in Weak Permissions

When strong permission (unique/full) splits into weak (pure/immutable), the weak permission carries **temporary state**—the state observed at split. This is **forgotten** when permissions rejoin, ensuring:

- Observers saw consistent state at observation time
- State may have changed by rejoin time (that’s OK)

Temporary State Capture and Forget

```

(* Split unique into full + pure WITH state capture *)
val split_unique_with_state :
  p:access_permission_v2{APUnique_v2? p} →
  (access_permission_v2 * access_permission_v2)
let split_unique_with_state (APUnique_v2 r n g) =
  let current_state = n in (* Capture state at split time *)
  (APFull_v2 r n g,
   APPure_v2 r n g (Some current_state)) (* Pure carries observed state *)

(* When rejoining, temporary state is FORGOTTEN *)
val join_full_pure_forget :
  full:access_permission_v2{APFull_v2? full} →
  pure:access_permission_v2{APPure_v2? pure} →
  option access_permission_v2
let join_full_pure_forget (APFull_v2 r n g) (APPure_v2 r' n' g' obs) =
  if r = r' && g = g' then
    (* Observed state (obs) is DISCARDED *)
    Some (APUnique_v2 r n g) (* Back to unique, no memory of observation *)
  else None

```

22.2 MALL Specifications for Method Contracts

Source: [Girard87] (Linear Logic), Bierhoff 2007 (Modular Typestate)

MALL: Linear Logic for Permission Contracts

Multiplicative-Additive Linear Logic ([Girard87]) provides precise permission contracts. MALL is the fragment without Par (\wp) or exponentials.

Girard’s Two Conjunctions (multiplicative vs additive):

$P \otimes Q$ = “have both P and Q ” (tensor/times) — both consumed

$P \& Q$ = “can use as P or as Q ” (with) — choice by consumer

Girard’s Two Disjunctions:

$P \oplus Q$ = “provides P or provides Q ” (plus) — choice by provider

$P \wp Q$ = Par (excluded from MALL)

Linear Implication:

$P \multimap Q$ = “consume P to produce Q ”

Effect Composition Mapping ([Girard87] Section 1.1.4):

- Sequential composition ($e_1; e_2$) maps to tensor ($E_1 \otimes E_2$)
- Choice (if-then-else) maps to plus ($E_1 \oplus E_2$)

The `mall_formula` type encodes MALL (Multiplicative-Additive Linear Logic) formulas for specifying method contracts. Each constructor corresponds to a linear logic connective:

- **MAtom**: A single permission (the atomic propositions of our logic)
- **MTensor** (\otimes): “Have both”—consumes both resources
- **MLolli** (\multimap): “Consume to produce”—linear implication
- **MWith** ($\&$): “Choose one”—internal choice by consumer
- **MPlus** (\oplus): “Provides one”—external choice by provider

MALL Formula Type

```
type mall_formula =  
  | MAtom : perm:access_permission_v2 → mall_formula  
  | MTensor : mall_formula → mall_formula → mall_formula      (* P tensor Q: have both *)  
  | MLolli : mall_formula → mall_formula → mall_formula      (* P -o Q: consume to produce *)  
  | MWith : mall_formula → mall_formula → mall_formula      (* P & Q: choose one *)  
  | MPlus : mall_formula → mall_formula → mall_formula      (* P + Q: provides one *)  
  | MOne : mall_formula      (* Unit for tensor *)  
  | MTop : mall_formula      (* Unit for with *)  
  | MZero : mall_formula      (* Unit for plus *)  
  | MBot : mall_formula      (* Unit for par *)
```

Method contracts specify resource protocols using MALL. A contract has:

- **Precondition**: Permissions the caller must provide (these are *consumed*)
- **Postcondition**: Permissions the method returns (these are *produced*)

The contract itself is a linear implication: `pre \multimap post`. Calling a method *consumes* the precondition and *produces* the postcondition—this is the essence of substructural reasoning for APIs. The examples below show file operations where state transitions are tracked through permission transformations.

Method Contracts in MALL

```
type method_contract = {  
  method_name : string;  
  precondition : mall_formula;      (* Required permissions - CONSUMED *)  
  postcondition : mall_formula;      (* Produced permissions - RETURNED *)  
}  
  
(* File.open(): unique(closed) -o unique(open) *)  
let file_open_contract : method_contract = {  
  method_name = "open";  
  precondition = MAtom (APUnique_v2 "file" "closed" "closed");  
  postcondition = MAtom (APUnique_v2 "file" "open" "open");  
}  
  
(* File.read(): can use full OR immutable - caller chooses *)  
let file_read_contract : method_contract = {  
  method_name = "read";  
  precondition = MWith  
    (MAtom (APFull_v2 "file" "open" "open"))  
    (MAtom (APIImmutable_v2 "file" "open" "open"));  
  postcondition = MWith  
    (MAtom (APFull_v2 "file" "open" "open"))  
    (MAtom (APIImmutable_v2 "file" "open" "open"));  
}
```

22.3 Vault Adoption and Focus Mechanisms

Papers: DeLine & Fahndrich 2001, 2002, 2004 (Vault, Fugue)

Vault: Adoption and Focus for Aliasing Management

Bierhoff (Section 22.1) provides 5 permission kinds for `tyestate`. Vault **complements** this with two critical operations for aliasing:

Adoption: Permanently transition from unique to shared-frozen state.

$$\text{unique}(\text{obj}) \xrightarrow{\text{adopt}} \text{pure}(\text{obj}) \quad [\text{Permanent, tyestate frozen}]$$

Use case: Publishing object to shared memory, `builder.build()`

Focus: Temporarily upgrade shared to unique within a scope.

$$\text{pure}(\text{obj}) \xrightarrow{\text{focus}} \{\text{unique}(\text{obj})\} \xrightarrow{\text{unfocus}} \text{pure}(\text{obj}) \quad [\text{Temporary}]$$

Use case: Critical section, exclusive state change on shared object

Key Difference from Rust:

- Rust borrows are **temporary** and return ownership
- Vault adoption is **permanent**—once adopted, never unique again
- Use adoption for “publish once” patterns (builders, caches)
- Use Rust-style borrowing (Section 21) for temporary access

The `adopt` function below demonstrates the key operation that transitions from unique ownership to permanent sharing. Once an object is adopted, its `tyestate` becomes frozen—it cannot be modified further. This is useful for “publish once” patterns like builders that produce immutable results.

Adoption Operation

```
(* Adopt operation: unique → pure (frozen) *)
val adopt :
  perm:access_permission_v2{APUnique_v2? perm} →
  adoption_result
let adopt (APUnique_v2 r n g) =
  (* Tyestate at adoption time becomes the permanent frozen state *)
  AdoptionSuccess (APPure_v2 r n g (Some n))

(* Adoption is idempotent on already-adopted (pure) permissions *)
val adopt_idempotent :
  perm:access_permission_v2 →
  Lemma (APPure_v2? perm ==> adopt_safe perm = perm)
```

The `focus` operation provides the inverse capability: temporarily upgrading a shared (pure) permission back to unique within a protected scope. This requires lock acquisition to ensure no concurrent access occurs during the focus period. The key insight is that `focus` is *scoped*—upon exiting the focus block, the permission reverts to pure with the updated `tyestate` visible to all aliases.

Focus Operation

```
(* Enter focus: pure → unique (temporarily) *)
val enter_focus :
  perm:access_permission_v2{APPure_v2? perm} →
  acquired_lock:bool →
  focus_result
let enter_focus (APPure_v2 r n g obs) acquired_lock =
  let token = {
    target_root = r; target_node = n; target_guarantee = g;
    original_mode = AMMayBeAliased;
    scope_id = fresh_scope_id ();
    requires_lock = true; (* Concurrent access requires lock *)
  } in
  if acquired_lock || not token.requires_lock then
    FocusSuccess token (APUnique_v2 r n g)
  else
    FocusFailure "Focus requires lock acquisition for concurrent access"
```

```

(* Exit focus: unique → pure (restore aliased state) *)
val exit_focus :
  token:focus_token →
  current_perm:access_permission_v2{APUnique_v2? current_perm} →
  access_permission_v2
let exit_focus token (APUnique_v2 r n g) =
  (* New typestate (n) becomes the frozen state for all aliases *)
  APPure_v2 r n g (Some n)

```

22.4 Frame-Based OOP Inheritance

Source: DeLine & Fahndrich 2004 (Vault, Fugue)

Frame Typestate for Object-Oriented Inheritance

When typestate meets inheritance, a challenge arises: subclasses may have their own state machines, and virtual method calls must handle all possible subclass states.

Key Insight: Decompose object typestate into *per-class frames*. Each class in the inheritance hierarchy maintains its own frame typestate. Virtual calls use “sliding” signatures that abstract over unknown subclass frames.

Frame Typestate Structure:

- Each class has a frame with its own state machine
- Object typestate = composition of all frame typestates
- Abstract references include a “rest” component for unknown subclasses
- Virtual dispatch slides all frames including rest

The following F* code defines the frame typestate structure. The `frame_typestate` type captures per-class state, while `object_typestate` composes frames for the full inheritance chain. The distinction between `OTSConcrete` and `OTSAbstract` is critical: concrete typestates know all frames (no virtual dispatch needed), while abstract typestates include a “rest” state for unknown subclass behavior.

Frame Typestate for OOP

```

(* Per-class frame typestate *)
type frame_typestate = {
  class_name : string;
  state_name : state_node;
  fields : option (list (string * symbolic_name)); (* None = packed *)
}

(* Object typestate: collection of frame typestates *)
type object_typestate =
  | OTSConcrete : frames:list frame_typestate → object_typestate
    (* Known concrete type - all frames specified *)
  | OTSAbstract : frames:list frame_typestate → rest:state_node → object_typestate
    (* Abstract reference - rest represents unknown subclass frames *)

(* Get frame for a specific class *)
val get_frame : object_typestate → string → option frame_typestate
let get_frame ots cls =
  let frames = match ots with
  | OTSConcrete fs → fs
  | OTSAbstract fs _ → fs
  in
  List.find (fun f → f.class_name = cls) frames

(* Upcast: convert concrete to abstract (lose subclass info) *)
val upcast : object_typestate → string → object_typestate
let upcast ots target_class =
  match ots with

```

```

| OTSConcrete frames →
  let (above, below) = List.partition (fun f →
    is_superclass_or_equal f.class_name target_class
  ) frames in
  let rest_state = match below with
    | [] → "Unknown"
    | f :: _ → f.state_name
  in
  OTSAbstract above rest_state
| OTSAbstract _ _ → ots  (* Already abstract *)

```

Virtual methods that change typestate require two signatures: an *implementation signature* (implSig) that specifies what the implementing class does, and a *virtual signature* (virtSig) that specifies behavior for all possible dynamic types. The key difference is that virtSig changes the “rest” state, affecting all unknown subclass frames.

Sliding Method Signatures

```

type method_signature = {
  method_name : string;
  declaring_class : string;
  this_pre : object_tystate;
  this_post : object_tystate;
}

(* Implementation signature: changes frames up to implementing class *)
val impl_sig : string → string → state_node → state_node → method_signature
let impl_sig decl impl pre_s post_s =
  let pre_ts = OTSAbstract
    [{ class_name = decl; state_name = pre_s; fields = None }]
    pre_s  (* rest unchanged *)
  in
  let post_ts = OTSAbstract
    [{ class_name = decl; state_name = post_s; fields = None };
     { class_name = impl; state_name = post_s; fields = None }]
    pre_s  (* rest UNCHANGED in impl sig *)
  in
  { method_name = ""; declaring_class = decl;
    this_pre = pre_ts; this_post = post_ts }

(* Virtual signature: changes all frames INCLUDING rest *)
val virt_sig : string → state_node → state_node → method_signature
let virt_sig decl pre_s post_s =
  let pre_ts = OTSAbstract
    [{ class_name = decl; state_name = pre_s; fields = None }]
    pre_s
  in
  let post_ts = OTSAbstract
    [{ class_name = decl; state_name = post_s; fields = None }]
    post_s  (* rest = post_state - ALL frames change! *)
  in
  { method_name = ""; declaring_class = decl;
    this_pre = pre_ts; this_post = post_ts }

```

22.4.1 API Pattern Examples with Adoption and Focus

The following examples demonstrate how Vault’s adoption and focus mechanisms apply to common design patterns. These patterns show the practical application of typestate and permission tracking.

Builder Pattern with Adoption

```

(* Builder Pattern: unique → unique → ... → adopt → pure + result
   Builder starts unique, accumulates state, adopts at build(). *)

type builder_state = | BEmpty | BHasName | BHasAge | BComplete

```

```

(* Builder.new() → unique(Empty) *)
let builder_new_post = APUnique_v2 "builder" "Empty" "alive"

(* Builder.withName() : unique(Empty) → unique(HasName) *)
let builder_withName_contract : method_contract = {
  method_name = "withName";
  precondition = MAtom (APUnique_v2 "builder" "Empty" "alive");
  postcondition = MAtom (APUnique_v2 "builder" "HasName" "alive");
}

(* Builder.build() : unique(HasAge) → ADOPT → pure(Complete) + unique(Person) *)
let builder_build_contract : method_contract = {
  method_name = "build";
  precondition = MAtom (APUnique_v2 "builder" "HasAge" "alive");
  postcondition = MTensor
    (MAtom (APPure_v2 "builder" "Complete" "Complete" (Some "Complete")))
    (MAtom (APUnique_v2 "person" "Valid" "alive"));
  (* Builder is ADOPTED - frozen at Complete, cannot be reused *)
}

```

Socket State Machine with Focus

```

(* Socket lifecycle demonstrates state transitions with focus for closing
   a shared socket. Multiple readers can share, but close requires focus. *)

type socket_state = | SRaw | SBound | SConnected | SClosed

let socket_bind_contract : method_contract = {
  method_name = "bind";
  precondition = MAtom (APUnique_v2 "socket" "Raw" "alive");
  postcondition = MAtom (APUnique_v2 "socket" "Bound" "alive");
}

(* receive() : pure(Connected) → pure(Connected) - read-only, aliased OK *)
let socket_receive_contract : method_contract = {
  method_name = "receive";
  precondition = MAtom (APPure_v2 "socket" "Connected" "alive" None);
  postcondition = MAtom (APPure_v2 "socket" "Connected" "alive" None);
}

(* close() requires focus if socket was shared:
   focus socket with lock {      -- temporarily unique(Connected)
     socket.close();             -- unique(Closed)
   }                             -- back to pure(Closed) *)

```

Vault-Bierhoff Permission Integration Summary

Bierhoff Provides (Section 22.1):

- 5 permission kinds: unique, full, share, immutable, pure
- Permission splitting and joining
- State guarantees via root nodes
- Per-node fraction functions

Vault Adds (This Section):

- Adoption: permanent sharing with frozen typestate
- Focus: temporary unique upgrade for aliased objects
- Frame typestate decomposition for OOP
- Sliding methods for virtual dispatch

Relationship:

| | | |
|--------------------|---|--|
| Vault NotAliased | = | Bierhoff unique |
| Vault MayBeAliased | = | Bierhoff pure with frozen state |
| Adoption | : | unique \rightarrow pure (permanent) |
| Focus | : | pure \rightarrow unique \rightarrow pure (temporary, scoped) |

vs Rust Borrowing:

- Rust borrow: temporary, lifetime-tracked, ownership returns
- Vault adoption: permanent, no restoration
- Vault focus: temporary exclusive access with explicit lock

Use Rust-style borrowing (Section 21) for most cases. Use Vault adoption for “publish once” patterns (builders, caches).

22.5 Rust Verification: RefinedRust and Prusti

Papers: [Gaher24] (RefinedRust); Astrauskas, Poli, Mueller, Summers, Leymann 2022 (Prusti)

Rust Verification: Beyond Basic Ownership

The `ownership_state` in Section 21.3 tracks lifecycle (owned/borrowed/moved) but lacks:

1. **Functional correctness** (`Vec::push` adds element correctly)
2. **Place structure** (nested field access `x.f.g.h`)
3. **Unsafe code verification** (preconditions for raw pointers)
4. **Temporal reasoning** (what holds when a borrow expires)

RefinedRust (Foundational, Coq-verified):

- Refinement types: $T @ spec$ where *spec* is mathematical
- Place types: `place(T)`, `blockeda(T)` for borrow tracking
- Unsafe verification with explicit preconditions
- Layout-generic (correct for any valid layout algorithm)

Prusti (Practical, automated via Viper):

- Pledges: `assert_on_expiry` for reborrowing contracts
- Automatic core proof from type information
- Pure functions for specification use
- Type-conditional specifications (`refine_spec`)

Selection Guidance:

| | |
|-----------------------------|---|
| Safe Rust, simple lifetimes | \rightarrow Prusti (faster, less annotation) |
| Unsafe Rust, libraries | \rightarrow RefinedRust (foundational proofs) |
| Functional correctness | \rightarrow Either (both support refinements) |
| Machine-checked proofs | \rightarrow RefinedRust (Coq output) |

22.5.1 Place Types and Blocked Types (RefinedRust)

RefinedRust introduces *place types* to make Rust’s lvalue structure first-class in the type system. The key insight is that a “place” (an lvalue like `x.field.subfield`) has two components:

1. The **refinement**: the mathematical/specification value for verification
2. The **place type**: the ownership structure (owned, blocked, uninitialized, dead)

The `PlaceBlocked` constructor is critical: when a place is borrowed, it becomes “blocked” until the lifetime ends. The inner type records what will be *restored* when the borrow expires. This enables precise tracking of what ownership state returns after borrowing.

Place Types for Rust Borrow Reasoning

```

type place_type =
| PlaceOwned : inner:rust_type → place_type
  (* place(T) - full ownership of a place containing value of type T *)
| PlaceBlocked : lifetime:lifetime_id → inner:rust_type → place_type
  (* blocked_'a(T) - place is borrowed until lifetime 'a ends
    The inner type T records what will be restored when borrow ends *)
| PlaceUninit : size:nat → place_type
  (* Uninitialized memory of given size in bytes *)
| PlaceDead : place_type
  (* Place has been moved out of - cannot access *)

(* Refined type = refinement paired with place type *)
type refined_type = {
  ref_val : refinement;      (* Mathematical/specification value *)
  place_ty : place_type;     (* Ownership structure *)
}

```

22.5.2 Pledge Specifications for Reborrowing (Prusti)

Prusti introduces *pledges* to handle a challenging verification problem: when a function returns a mutable reference (reborrowing), what happens when that reference expires? The key insight is that reborrowing creates a *temporal contract*:

- **Caller obligation:** ensure certain conditions hold *before* the reference expires
- **Callee guarantee:** certain conditions will hold *after* the reference expires

This is expressed via `assert_on_expiry` annotations that specify both the pre-expiry constraint and the post-expiry guarantee. The pledge below demonstrates this for a BST's `get_root_value` method, which returns a mutable reference to the root value.

Pledges for Reborrowing Contracts

```

(* MOTIVATION: When a function returns a mutable reference (reborrowing),
we need to specify:
1. What the CALLER must ensure before the reference expires
2. What the CALLEE guarantees after the reference expires *)

type pledge = {
  (* The reborrowed reference this pledge applies to *)
  reborrowed_ref : string;

  (* Constraint that MUST hold before expiry (caller obligation) *)
  pre_expiry_constraint : temporal_expr → prop;

  (* Guarantee that WILL hold after expiry (callee guarantee) *)
  post_expiry_guarantee : temporal_expr → temporal_expr → prop;
}

(* Example: BST get_root_value specification *)
let bst_get_root_pledge : pledge = {
  reborrowed_ref = "result";
  pre_expiry_constraint = fun te →
    (* All elements in left subtree less than result *)
    ∀_in_tree (fun elem → elem < eval_at_before_expiry te) left_subtree;
  post_expiry_guarantee = fun old_self before_result →
    (* The root value equals what was returned *)
    match old_self with
    | Node (root_val, _, _) → root_val = before_result
    | _ → false;
}

```

23. Linearizability for Concurrent Objects

Paper: Herlihy & Wing 1990

What Linearizability Provides

The synthesis has: race detection, ownership, happens-before. The synthesis **lacks**: concurrent data structure *correctness*.

Linearizability answers: “Does this concurrent queue behave like a queue?” (Not just “are there races?” but “is the behavior correct?”)

Key Theorem (Locality): H is linearizable $\iff \forall x. H|_x$ is linearizable.

This enables **modular verification**—verify each object independently.

Definition 23.1 (Linearizability). H is linearizable w.r.t. spec S iff:

1. H can be extended to H' (completing pending operations)
2. $\text{complete}(H')$ is equivalent to some legal sequential history S
3. Precedence in H is preserved in S

The formalization below captures concurrent execution histories as sequences of events. Each event is either an **Invocation** (a process starting an operation) or a **Response** (the operation completing). The **sequential** predicate checks if a history represents serial execution (invocations immediately followed by responses), while **precedes** captures the real-time ordering that linearizability must preserve.

Histories and Events

```

type event =
  | Invocation : obj:string → op:string → args:list value → proc:nat → event
  | Response : obj:string → result:value → proc:nat → event

type history = list event

(* A history is sequential if invocations immediately followed by responses *)
let rec sequential (h : history) : bool =
  match h with
  | [] → true
  | Invocation _ _ _ _ :: Response _ _ _ _ :: rest → sequential rest
  | _ → false

(* Precedence: response r precedes invocation i if r appears before i *)
let precedes (e1 e2 : event) (h : history) : bool =
  match e1, e2 with
  | Response _ _ _, Invocation _ _ _ _ → index_of e1 h < index_of e2 h
  | _ → false

```

Why Set Abstraction for Concurrent Objects

Sequential Objects: $\alpha : \text{rep} \rightarrow \text{abs}$ (Single abstract value)

Concurrent Objects: $\alpha : \text{rep} \rightarrow \mathcal{P}(\text{abs})$ (Set of abstract values)

Why a set? Concurrent execution introduces nondeterminism. Multiple threads may interleave operations, and multiple linearizations of the same history may all be valid. Each linearization produces a (potentially different) abstract state.

Example: Concurrent queue with `enqueue(1)`, `enqueue(2)` overlapping:

- Linearization 1: `enqueue(1)` then `enqueue(2)` $\rightarrow [1, 2]$
- Linearization 2: `enqueue(2)` then `enqueue(1)` $\rightarrow [2, 1]$
- $\alpha(\text{rep}) = \{[1, 2], [2, 1]\}$ — both are valid abstract states

23.1 Progress Properties for Concurrent Objects

Source: Herlihy & Shavit 2008 — “The Art of Multiprocessor Programming”

Linearizability is a *safety* property (“nothing bad happens”). Progress properties ensure *liveness* (“something good eventually happens”).

Definition 23.2 (Progress Guarantees (Hierarchy from strongest to weakest)). **Wait-Free:** Every operation completes in bounded steps, regardless of other threads. No thread can be starved.

Lock-Free: System-wide progress: some operation completes in finite time. Individual threads may starve, but system as a whole progresses.

Obstruction-Free: Progress guaranteed if thread runs in isolation (no contention). May live-lock under contention.

Blocking: May use locks; susceptible to deadlock, priority inversion, and indefinite blocking if lock holder fails.

Progress Guarantee Types

```
type progress_guarantee =
  | WaitFree : bound:nat → progress_guarantee
  | LockFree : progress_guarantee
  | ObstructionFree : progress_guarantee
  | Blocking : progress_guarantee

(* Well-known classifications *)
let michael_scott_queue : progress_guarantee = LockFree
let harris_linked_list : progress_guarantee = LockFree
let herlihy_snapshot : progress_guarantee = WaitFree 2 (* 2-scan algorithm *)
let treiber_stack : progress_guarantee = LockFree
let mutex_protected : progress_guarantee = Blocking
```

23.2 Temporal Logic for Liveness

Source: Pnueli 1977 — “The Temporal Logic of Programs”

Temporal Logic for Program Verification

Progress properties describe *what* guarantees a concurrent object provides. Temporal logic provides the formal *language* to specify and verify these properties.

Key Insight (Pnueli): Program properties split into two classes:

- **Safety:** “Nothing bad ever happens” (invariance)
- **Liveness:** “Something good eventually happens” (eventuality)

The temporal operators \Box (always) and \Diamond (eventually) provide:

- Precise specification of concurrent behavior
- Proof principles for verification (P1 for safety, P2 for liveness)
- Foundation for fairness assumptions in concurrent systems

Temporal Operators

```
type temporal_formula =
  (* State formula lifted to path *)
  | TFState : state_formula → temporal_formula

  (* Next: holds in the next state *)
  | TFNext : temporal_formula → temporal_formula (* X phi *)

  (* Globally/Always: holds in all future states *)
  | TFAlways : temporal_formula → temporal_formula (* G phi or Box phi *)

  (* Finally/Eventually: holds in some future state *)
  | TFEventually : temporal_formula → temporal_formula (* F phi or Diamond phi *)

  (* Until: phi holds until psi becomes true *)
  | TFUntil : temporal_formula → temporal_formula → temporal_formula (* phi U psi *)

  (* Leads-to: if phi holds, psi eventually holds *)
  | TFLeadsTo : temporal_formula → temporal_formula → temporal_formula (* phi -> psi *)
```

Theorem 23.3 (Invariance Principle (P1) — Pnueli 1977). *To prove $\Box P$ (P always holds):*

1. *Show P holds initially*
2. *Show every transition preserves P*

*This is the foundation for verifying **safety** properties.*

Theorem 23.4 (Eventuality Principle (P2) — Pnueli 1977). *To prove $P \rightsquigarrow Q$ (if P holds, Q eventually holds): Find a ranking function $r : \text{state} \rightarrow \mathbb{N}$ such that:*

- *r decreases on each step while Q does not hold*
- *When r reaches 0, Q must hold*

*This is the foundation for verifying **liveness** properties.*

23.3 CTL Model Checking

Source: Clarke, Emerson, Sistla 1986 — “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic”

CTL Model Checking: Branching-Time Temporal Logic

Pnueli’s LTL (Section 23.2) uses **linear time**—formulas describe properties of single execution paths. CTL uses **branching time**—formulas can distinguish between “all paths” and “some path”.

Key Distinction:

- LTL: $\Box(\text{request} \Rightarrow \Diamond \text{response})$ — on this path, responses follow requests
- CTL: $\text{AG}(\text{request} \Rightarrow \text{AF response})$ — on all paths from all states
- CTL: $\text{AG}(\text{request} \Rightarrow \text{EF response})$ — on some path from all states

Why CTL Matters for Security:

- Use-after-free: $\text{AG}(\text{freed}(p) \Rightarrow \text{AX AG}(\neg \text{deref}(p)))$
- Resource leak: $\text{AG}(\text{acquire}(r) \Rightarrow \text{AF release}(r))$
- Deadlock-free: $\text{AG}(\text{EF progress})$

Complexity: $O(|f| \cdot (|S| + |R|))$ — **linear** in formula and state graph. This is much better than LTL model checking (PSPACE-complete)!

CTL Formula Type

```
type ctl_formula =
  | CTLAtom : pred:(node_id → bool) → ctl_formula
  | CTLNot : ctl_formula → ctl_formula
  | CTLAnd : ctl_formula → ctl_formula → ctl_formula

  (* Path quantifier + Next *)
  | CTLAX : ctl_formula → ctl_formula (* On ALL paths, phi holds NEXT *)
  | CTLEX : ctl_formula → ctl_formula (* On SOME path, phi holds NEXT *)

  (* Path quantifier + Until *)
  | CTLAU : ctl_formula → ctl_formula → ctl_formula (* A[phi U psi] *)
  | CTLEU : ctl_formula → ctl_formula → ctl_formula (* E[phi U psi] *)

  (* Derived operators *)
  let ctl_af phi = CTLAU ctl_true phi (* AF: inevitable *)
  let ctl_ef phi = CTLEU ctl_true phi (* EF: reachable *)
  let ctl_ag phi = CTLNot (ctl_ef (CTLNot phi)) (* AG: invariant *)
  let ctl_eg phi = CTLNot (ctl_af (CTLNot phi)) (* EG: persistent *)
```

Security Properties as CTL Formulas

```
(* Use-after-free prevention:
   AG(freed(p) → AX AG(not deref(p))) *)
val use_after_free_safe : (node_id → bool) → (node_id → bool) → ctl_formula
let use_after_free_safe is_free is_deref =
```

```

let freed = CTLAtom is_free in
let deref = CTLAtom is_deref in
ctl_ag (CTLImplies freed (CTLAX (ctl_ag (CTLNot deref))))

(* Resource leak detection:
   AG(acquire(r) → AF release(r)) *)
val resource_must_release : (node_id → bool) → (node_id → bool) → ctl_formula
let resource_must_release is_acquire is_release =
  let acquired = CTLAtom is_acquire in
  let released = CTLAtom is_release in
  ctl_ag (CTLImplies acquired (ctl_af released))

(* Deadlock freedom:
   AG(EF progress) *)
val deadlock_free : (node_id → bool) → ctl_formula
let deadlock_free is_progress =
  ctl_ag (ctl_ef (CTLAtom is_progress))

```

24. Frame Rule and Footprint Computation

Source: [Reynolds02] — “Separation Logic: A Logic for Shared Mutable Data Structures”

24.1 The Frame Rule

Theorem 24.1 (Frame Rule — The Most Important Theorem for Compositional Analysis).

$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}}$$

Provided:

- *R* shares no modified variables with *C*
- *R* does not mention locations freed by *C*

Meaning: If *C* is correct with precondition *P* and postcondition *Q*, then *C* is **also** correct when extra resources *R* exist.

Why This Matters:

- Analyze functions in **isolation**
- Caller’s resources automatically preserved
- Enables true **compositional** verification

24.2 Footprint Computation

Memory Footprint Definition

```

type footprint = {
  reads : set loc;      (* Locations read *)
  writes : set loc;     (* Locations written *)
  allocates : set loc;  (* Locations allocated *)
  frees : set loc;      (* Locations freed *)
}

val compute_footprint : ir_stmt → footprint
let rec compute_footprint stmt = match stmt with
| SRead dst ptr →
  { empty_footprint with reads = singleton (resolve_loc ptr) }
| SWrite ptr val_ →
  { empty_footprint with writes = singleton (resolve_loc ptr) }
| SAlloc dst size →
  { empty_footprint with allocates = fresh_abstract_loc () }
| SFree ptr →
  { empty_footprint with frees = singleton (resolve_loc ptr) }
| SSeq s1 s2 →

```

```

merge_footprint (compute_footprint s1) (compute_footprint s2)
| Sif cond s1 s2 →
  merge_footprint (compute_footprint s1) (compute_footprint s2)
| SWhile cond body →
  compute_footprint body
| SCall dst func args →
  lookup_footprint_summary func
| SPure _ → empty_footprint

```

24.3 Compositional Analysis Algorithm

Algorithm: Analyze function with frame rule

1. **Compute Footprint:** $\text{footprint}(f) = \bigcup \{\text{footprint}(\text{stmt}) \mid \text{stmt} \in f.\text{body}\}$
2. **Generate Specification:**
 - Pre: resources needed by footprint
 - Post: resources produced/modified
3. **At Call Sites:**
 - Caller has resources: P_{caller}
 - Callee needs: P_{callee} (from spec)
 - Frame: $R = P_{\text{caller}} - P_{\text{callee}}$ (what's left over)
 - Apply frame rule: $\{P_{\text{callee}} * R\} \text{ call } \{Q_{\text{callee}} * R\}$
4. **Verify Disjointness:**
 - Callee footprint $\cap R = \emptyset$
 - No aliasing between callee access and frame

Benefit: Analyze callee **once**, reuse at all call sites!

24.4 Magic Wands for Partial Data Structures

Source: Muller et al. 2016 (Viper), [Reynolds02]

Definition 24.2 (Magic Wand (Separating Implication)). Magic wand $A \multimap B$ means: “providing A yields B ”

Semantics: $(A \multimap B)$ holds in state s if:

$$\forall s'. s' \text{ disjoint from } s \implies A \text{ holds in } s' \implies B \text{ holds in } (s * s')$$

Use Cases:

- Partial data structure traversal: $\text{lseg}(\text{ptr}, \text{null}) \multimap \text{lseg}(\text{hd}, \text{null})$
- Reborrowing specifications (Prusti)
- Modular verification of iterative algorithms

Viper Syntax:

```
acc(lseg(ptr, null)) --* acc(lseg(hd, null))
```

Operations:

```

package (A --* B)    -- Create wand
apply (A --* B)      -- Use wand: exchange A for B

```

Complexity Warning: Full magic wand reasoning is PSPACE-complete (Reynolds). Use for *specification* only. Prefer frame rule for implementation.

24.5 Quantified Permissions for Arrays

Source: Muller et al. 2016 (Viper)

Quantified Permissions

```
(* Enables pointwise permission specification for arrays and graphs:
    $\forall i: \text{Int} :: 0 \leq i \ \&\& \ i < \text{len} \Rightarrow \text{acc}(\text{loc}(\text{arr}, i).\text{val}) *$  *)

type quantified_permission = {
  index_var : string;
  index_type : ir_type;
  range_constraint : ir_expr;      (* e.g.,  $0 \leq i \ \&\& \ i < \text{len} *$  *)
  permission_body : sl_assertion; (* e.g.,  $\text{acc}(\text{loc}(\text{arr}, i).\text{val}) *$  *)
}

(* Array permission: all elements accessible *)
val array_perm : arr:var_id → len:nat → quantified_permission
let array_perm arr len = {
  index_var = "i";
  index_type = TInt;
  range_constraint = EBinOp OpAnd
    (EBinOp OpLeq (EInt 0) (EVar "i"))
    (EBinOp OpLt (EVar "i") (EInt len));
  permission_body = SLPointsTo (EArrayAccess (EVar arr) (EVar "i")) PermFull;
}
```

Advantages over Recursive Predicates:

- Direct random access (no fold/unfold chains)
- Works for cyclic structures
- Efficient SMT encoding

24.6 Automated Proof Search for Separation Logic

Source: Mulder et al. 2022 (Diaframe), Calcagno et al. 2009 (Bi-Abduction)

Diaframe: Goal-Directed Separation Logic Proof Search

Key Insight: Treat separation logic connectives as proof search instructions. Inspired by linear logic programming (Hodas & Miller).

Automation Strategy:

1. Interpret connectives as search instructions:
 - $*$ (star) \rightarrow ISplit: split goal into two
 - \multimap (wand) \rightarrow IIntro: introduce hypothesis
 - $\exists x \rightarrow$ IExists: postpone instantiation
 - \Rightarrow (update) \rightarrow IUpdate: perform ghost state update
2. Use bi-abduction for resource matching
3. Apply domain-specific hints when stuck

Diaframe Achieves:

- Foundational soundness (proofs checked by Coq kernel)
- Practical automation (10x less manual proof than raw Iris)
- Extensibility via user-defined hints

25. Representation Predicates

Source: VeriFFI (Wang et al. 2025) — “A Verified Foreign Function Interface between Coq and C”

Representation Predicates — VeriFFI

Representation predicates connect **high-level types** to **low-level memory**. This is critical for:

- FFI boundary verification

- Data structure correctness
- Garbage collection safety

Key Insight: Without representation predicates, we cannot prove that foreign function calls preserve type invariants across language boundaries.

Definition 25.1 (Representation Predicate (InGraph)). $\text{InGraph}(g, x, p, t)$ means:

- x is a value of type t
- p points to x 's representation in memory graph g
- The representation is **valid** according to t 's layout

Representation Predicate Type Class

```
class rep_predicate (t : Type) = {
  (* The predicate itself: graph → value → address → proposition *)
  in_graph : memory_graph → t → address → bool;

  (* MONOTONICITY: Adding nodes doesn't invalidate existing predicates.
     Critical for GC correctness. *)
  monotone : squash (∀ g g' x p.
    extends g g' →
    in_graph g x p →
    in_graph g' x p);

  (* INJECTIVITY: Same address, same type ⇒ same value.
     A representation is deterministic. *)
  injective : squash (∀ g x y p.
    in_graph g x p →
    in_graph g y p →
    x == y);
}
```

Theorem 25.2 (GC Preservation (VeriFFI Lemma 3.2)). *Representation predicates are preserved under GC isomorphism. If x is represented at p in old_graph , and GC maps p to p' , then x is represented at p' in new_graph .*

26. GC-Aware Ownership Analysis

Papers: VeriFFI (Wang et al. 2025), RustBelt ([Jung18])

GC-Isomorphism — VeriFFI

For languages with garbage collection (Java, Python, Go, OCaml), ownership analysis must account for:

- GC can **move** objects (address changes)
- GC can **collect** unreachable objects (lifetime shorter than scope)
- GC provides **implicit** memory safety

The key property: **representation preserved under GC**.

26.1 GC-Isomorphism Definition

GC State and Isomorphism

```
type gc_state = {
  heap : memory_graph;
  roots : set address;
  generation : nat; (* GC generation/epoch *)
}

type gc_isomorphism = {
  source : gc_state;
```

```

target : gc_state;
address_map : map address address;  (* Old addr → new addr *)
}

(* Validity: bijection on reachable, preserves structure and values *)
val is_valid_gc_iso : gc_isomorphism → bool
let is_valid_gc_iso iso =
  is_bijection iso.address_map (reachable iso.source) (reachable iso.target) &&
  (* Preserves pointer structure *)
  (∀ a b. points_to iso.source.heap a b =>
    points_to iso.target.heap (Map.find a iso.address_map)
    (Map.find b iso.address_map)) &&
  (* Preserves values *)
  (∀ a. iso.source.heap.values a =
    iso.target.heap.values (Map.find a iso.address_map))

```

26.2 Ownership in GC Languages

GC-Aware Ownership States

```

(* For GC languages, we DON'T track deallocation (GC handles it).
   Instead, we track reachability, mutation, and finalization. *)

type gc_ownership_state =
| GCRooted          (* Definitely reachable from a GC root *)
| GCReachable       (* Reachable from some rooted object *)
| GCMaybeCollected (* May have been collected *)
| GCFinalized       (* Finalizer has run - object is "dead" *)

```

26.3 GC at Language Boundaries

GC Requirements at FFI Boundaries

```

(* CRITICAL: When GC and non-GC languages interact via FFI,
   we must ensure GC doesn't collect objects still referenced
   by the non-GC side. *)

type boundary_gc_requirement =
| PinDuringCall      (* Object must not move during FFI call *)
| RegisterAsRoot     (* Register pointer as GC root *)
| CopyOut            (* Copy data out of GC heap *)
| Opaque             (* Non-GC side cannot access GC heap *)

val determine_gc_requirement :
  source_lang : language_id{has_gc source_lang} →
  target_lang : language_id →
  value_type : ir_type →
  boundary_gc_requirement
let determine_gc_requirement src tgt ty =
  if has_gc tgt then
    if same_gc_runtime src tgt then Opaque
    else RegisterAsRoot
  else
    if is_primitive ty then CopyOut
    else if is_small_struct ty then CopyOut
    else PinDuringCall

```

27. WebAssembly Memory Safety Analysis

Papers: MS-Wasm (Disselkoen et al. 2019), WASM Security Survey (Perrone & Romano 2024)

WebAssembly Memory Model and Safety Extensions

WebAssembly provides **isolation** from host (sandbox) but **not** memory safety *within* the sandbox. C/C++ vulnerabilities carry over when compiled to Wasm.

MS-Wasm extends Wasm with explicit memory safety semantics:

- **Segments:** bounded memory regions with lifetime tracking
- **Handles:** typed pointers that encapsulate bounds information
- **Progressive enforcement:** backends choose safety/performance tradeoff

Key insight: Capture C/C++ pointer semantics at *compile time* so backends can leverage *hardware features* (ARM MTE, PAC, CHERI) at runtime.

27.1 WebAssembly Linear Memory Model

WebAssembly uses **linear memory**: a contiguous, mutable array of raw bytes.

- Loads and stores operate on untyped byte addresses (i32)
- No bounds metadata preserved from source language
- Stack-smashing mitigated (separate stack), but in-sandbox exploits remain

Attack primitives that still work in WebAssembly:

- Stack-based buffer overflow (into Wasm’s linear memory stack area)
- Heap metadata corruption
- Overwriting “constant” data in linear memory
- Redirecting indirect calls via corrupted function table indices

27.2 MS-Wasm Segment Memory Model

MS-Wasm Handle

```
(* HANDLE: A typed pointer with bounds information
   4-tuple: (base, offset, bound, isCorrupted) *)

type ms_wasm_handle = {
  base : address;      (* Start of segment in segment memory *)
  offset : int;         (* Current offset within segment *)
  bound : nat;          (* Size of accessible region *)
  is_corrupted : bool; (* Set if handle integrity violated *)
}

(* Spatial safety check *)
val is_spatially_safe : ms_wasm_handle → bool
let is_spatially_safe h =
  not (h = handle_null) &&
  h.offset ≥ 0 &&
  h.offset < h.bound &&
  not h.is_corrupted

(* Handle slicing for intra-object safety *)
val segment_slice :
  parent:ms_wasm_handle →
  new_base:nat →
  new_bound:nat{new_base + new_bound ≤ parent.bound} →
  ms_wasm_handle
let segment_slice parent new_base new_bound =
  { base = parent.base + new_base;
    offset = 0;
    bound = new_bound;
    is_corrupted = parent.is_corrupted }
```

27.3 MS-Wasm Memory Safety Properties

Three Memory Safety Properties (de Amorim et al. 2017):

1. **Spatial Safety:** no out-of-bounds reads/writes
2. **Temporal Safety:** no use-after-free
3. **Pointer Integrity:** no pointer forgery or corruption

27.4 Hardware-Accelerated Memory Safety

Hardware Mechanisms for MS-Wasm Backends

```
type hw_memory_safety_mechanism =  
  | HWSoftwareOnly      (* Pure software checks,  $\neg 2x$  overhead *)  
  | HWMemoryTagging     (* ARM MTE,  $\neg 0\%$  overhead, probabilistic *)  
  | HWPointerAuth       (* ARM PAC,  $\neg 20\%$  overhead for all pointers *)  
  | HWCcapabilities     (* CHERI,  $<5\%$  overhead, full spatial safety *)  
  
type safety_enforcement_level =  
  | EnforceFull         (* All three properties enforced *)  
  | EnforceSpatialOnly  (* Bounds checking only *)  
  | EnforceProbabilistic (* MTE-style detection *)  
  | EnforceNone         (* Performance mode, equivalent to plain Wasm *)
```

27.5 WebAssembly Analysis Landscape (2024 Survey)

Source: Perrone & Romano 2024 (121 papers surveyed)

Analysis Categories:

- **Static Analysis:** CFG analysis, context-sensitive data flow, CPG, symbolic semantic graphs
- **Dynamic Analysis:** Fuzzing, symbolic execution (EUNOMIA), concolic execution (SWAM), runtime instrumentation (Wasabi)

Vulnerability Types Detected:

- Smart Contracts: Fake transfer, reentrancy, rollback, missing auth
- Memory Safety: Buffer overflow, UAF, double-free, format string
- Side Channels: Cache timing, port contention, transient execution

Security Enhancements:

- CT-Wasm (Watt 2019): Type-driven constant-time for crypto
- MS-Wasm (Michael 2023): Memory safety via segment memory
- Formal verification: WasmCert-Isabelle, WasmCert-Coq
- Software Fault Isolation (SFI): VeriWasm verifier

27.6 Integration with Brrr-Machine Analysis

Integration Points:

1. Part V (Pointer Analysis):

- Wasm linear memory as single abstract object
- Function table indices as pointer-like values
- MS-Wasm handles map directly to ownership analysis

2. Part VIII (Security Analysis):

- Taint analysis for Wasm imports (JS inputs are taint sources)
- XSS/injection via Wasm-to-JS string passing
- Cryptojacking detection (characteristic Wasm patterns)

3. Section 26 (GC-Aware Analysis):

- JS-Wasm boundary: JS side is GC'd, Wasm side is manual
- ArrayBuffer references must survive GC cycles

- Similar to VeriFFI constraints
4. **Part IX (Multi-Language):**
- Wasm-JS boundary follows Matthews-Findler semantics
 - Type coercions at boundary (JS dynamic \rightarrow Wasm static)
 - Gradual typing applicable to Wasm-JS interface
- Cross-References:**
- See Section 26 for GC-aware ownership when Wasm interoperates with JS/Python
 - See Part VIII for taint analysis applicable to Wasm imports
 - See Part IX for boundary semantics (JS-Wasm follows Matthews-Findler)

Part VIII

Security Analysis — Information Flow and Taint

Effect Absence Enables Zero False-Positive Detection

Source: [Leijen14] (Koka), Theorems 2–4 — See Section 12.27

Critical Insight: Effect inference provides *mathematically proven* safety. If an effect is **absent** from the inferred effect row, the corresponding bug class is **impossible**—not just unlikely, but provably cannot occur.

Effect-Based Bug Classification:

| Condition | Guarantee |
|--|--|
| $\text{exn} \notin \text{effects}$ | Proven exception-safe (Theorem 12.27.1) No unhandled exceptions possible—0% false positive |
| $\text{div} \notin \text{effects}$ | Proven terminating (Theorem 12.27.2) Guaranteed to halt—no infinite loops possible |
| $\text{st}\langle h \rangle \notin \text{effects}$ | Proven heap-isolated (Theorem 12.27.3) Cannot modify/observe heap h —state encapsulated |
| $\text{IO} \notin \text{effects}$ | Proven no external I/O Taint cannot exfiltrate via I/O sinks |

Application to Taint Analysis: Effect absence *strengthens* taint findings by eliminating false positives:

- Exception sinks unreachable if $\text{exn} \notin \text{effects}$
- Heap-based taint flows impossible if $\text{st}\langle h \rangle \notin \text{effects}$
- External data exfiltration impossible if $\text{IO} \notin \text{effects}$

Integration: Use effect inference *before* taint analysis to prune impossible flows. Remaining flows have higher true-positive rate.

Cross-references: Section 12.27 for effect absence theorems and proofs; Section 12.34 for robust declassification theorems; Section 6.1.3 for effect row definitions.

28. Information Flow and Taint Analysis

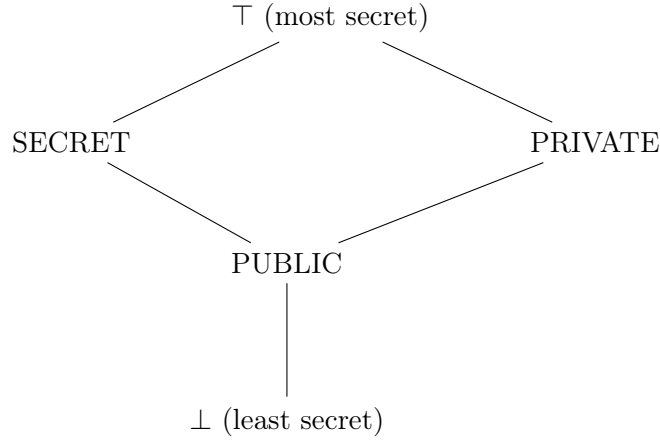
Foundational Papers

[Denning77], [Livshits05], [Tripp09]

Security vulnerabilities are fundamentally about improper information flow: untrusted data reaching sensitive operations without proper validation.

28.1 Denning’s Lattice Model

Definition 28.1 (Security Lattice [Denning77]). Security levels form a *lattice*:



Definition 28.2 (Information Flow Rule). Information can flow from level L_1 to level L_2 *only* if $L_1 \sqsubseteq L_2$ (L_1 is below or equal to L_2):

- ✓ PUBLIC \rightarrow SECRET (upgrading is safe)
- × SECRET \rightarrow PUBLIC (leaking is dangerous)

28.1.1 Taint Mapping

For taint analysis, we assign:

UNTAINTED = safe = low security

TAINTED = dangerous = high security

Improper flow: TAINTED \rightarrow *sensitive_operation*

28.1.2 The 4-Point Lattice with Integrity

Denning's original model tracks only **confidentiality** (secrecy). For robust declassification (Section 28.4.3, Section 12.34), we need **integrity** too.

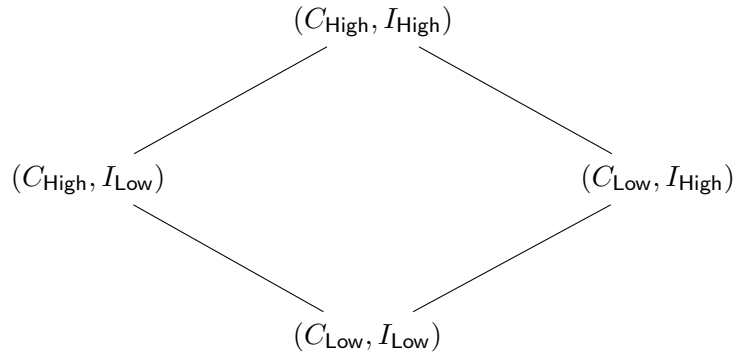
Definition 28.3 (Confidentiality and Integrity). • **Confidentiality**: Can the adversary *observe* this data?

- C_{Low} = Public (adversary can observe)
- C_{High} = Secret (adversary cannot observe)

• **Integrity**: Can the adversary *influence* this data?

- I_{Low} = Untrusted (adversary can control)
- I_{High} = Trusted (adversary cannot control)

Definition 28.4 (4-Point Product Lattice).



Where:

- $(C_{\text{High}}, I_{\text{High}})$: Secret and Trusted (e.g., system password)

- $(C_{\text{High}}, I_{\text{Low}})$: Secret but Untrusted
- $(C_{\text{Low}}, I_{\text{High}})$: Public and Trusted
- $(C_{\text{Low}}, I_{\text{Low}})$: Public and Untrusted (adversary-controlled input)

Ordering: $(c_1, i_1) \sqsubseteq (c_2, i_2)$ iff $c_1 \sqsubseteq_C c_2$ and $i_1 \sqsupseteq_I i_2$

Note: Integrity ordering is *inverted* (higher integrity = lower risk).

Why Integrity Matters

For **robust declassification** (Section 28.4.3), we must ensure:

- Adversary cannot control *which* secret gets declassified
- Adversary cannot influence the *value* being declassified

This requires tracking integrity of control flow and data dependencies.

Mapping Taint to 4-Point Lattice:

- Taint source (user input) $\Rightarrow \{conf = C_{\text{Low}}; integ = I_{\text{Low}}\}$
- Secret source (password) $\Rightarrow \{conf = C_{\text{High}}; integ = I_{\text{High}}\}$
- Public constant $\Rightarrow \{conf = C_{\text{Low}}; integ = I_{\text{High}}\}$

Cross-reference: Section 12.34 for complete formalization and theorems.

28.2 The Taint Analysis Framework

The taint analysis framework provides the core infrastructure for tracking untrusted data as it flows through a program. The following F* code defines the fundamental types used throughout the analysis:

- **Taint sources** represent entry points where untrusted data enters the system (HTTP parameters, environment variables, file reads, etc.)
- **Taint sinks** represent sensitive operations where untrusted data could cause harm (SQL queries, shell commands, HTML output, etc.)
- **Sanitizers** represent functions that transform untrusted data into safe values for specific contexts

The types are designed to be language-agnostic, supporting analysis across Python, JavaScript, Go, Rust, and other languages in the brrr-machine framework.

Taint Analysis Core Types

```
(* =====
   Taint ANALYSIS
   Sources: Denning 1977, Livshits 2005, Tripp 2009
   ===== *)
module BrrrMachine.Security.Taint

(* -----
   Taint SOURCES --- Where untrusted data enters
   ----- *)
type taint_source =
  (* User input *)
  | SrcHttpParam : param:string → taint_source
  | SrcHttpHeader : header:string → taint_source
  | SrcHttpBody : taint_source
  | SrcCookie : name:string → taint_source
  | SrcUrlPath : taint_source
  (* Environment *)
  | SrcEnvVar : var:string → taint_source
  | SrcCommandArg : index:nat → taint_source
  | SrcStdin : taint_source
  (* External data *)
  | SrcFileRead : taint_source
  | SrcNetworkRead : taint_source
```

```

| SrcDatabaseRead : taint_source
| SrcDeserialize : format:string → taint_source
(* Language-specific *)
| SrcEval : taint_source (* JS eval, Python exec *)
| SrcReflection : taint_source

```

Taint Sinks — Where Tainted Data is Dangerous

```

type taint_sink =
  (* Injection vulnerabilities *)
  | SinkSqlQuery : taint_sink
  | SinkSqlParam : taint_sink
  | SinkShellCommand : taint_sink
  | SinkShellArg : taint_sink
  | SinkLdapQuery : taint_sink
  | SinkXPathQuery : taint_sink
  | SinkRegexPattern : taint_sink
  (* XSS *)
  | SinkHtmlOutput : taint_sink
  | SinkHtmlAttribute : taint_sink
  | SinkJavaScript : taint_sink
  | SinkCssValue : taint_sink
  (* Path traversal *)
  | SinkFilePath : taint_sink
  | SinkFileOpen : taint_sink
  (* Deserialization *)
  | SinkDeserialize : format:string → taint_sink
  | SinkYamlLoad : taint_sink
  | SinkPickle : taint_sink
  (* Code execution *)
  | SinkEval : taint_sink
  | SinkExec : taint_sink
  | SinkReflection : taint_sink
  (* Sensitive data exposure *)
  | SinkLog : taint_sink
  | SinkHttpResponse : taint_sink
  | SinkErrorMessage : taint_sink
  (* SSRF *)
  | SinkHttpRequest : taint_sink
  | SinkUrlFetch : taint_sink

```

Sanitizers — Functions that Make Tainted Data Safe

```

(* CRITICAL DISTINCTION (see Section 8.1.4.3, 12.34.9):
- Sanitizer: Changes VALUE, preserves LABEL (e.g., HTML escape)
- Declassification: Preserves VALUE, changes LABEL (policy decision)

```

```

Both may be needed: sanitize to prevent injection, declassify to
allow controlled observation of secrets. *)

```

```

type sanitizer =
  | SanHtmlEscape : sanitizer
  | SanUrlEncode : sanitizer
  | SanSqlEscape : sanitizer
  | SanSqlParameterize : sanitizer
  | SanShellEscape : sanitizer
  | SanPathCanonicalize : sanitizer
  | SanInputValidation : pattern:string → sanitizer
  | SanTypeCheck : expected:string → sanitizer
  | SanLengthCheck : max_len:nat → sanitizer
  | SanWhitelist : allowed:list string → sanitizer
  | SanCustom : name:string → sanitizer

(* Which sinks does a sanitizer protect? *)
let sanitizer_protects (san : sanitizer) (sink : taint_sink) : bool =

```

```

match san, sink with
| SanHtmlEscape, (SinkHtmlOutput | SinkHtmlAttribute) → true
| SanSqlEscape, (SinkSqlQuery | SinkSqlParameter) → true
| SanSqlParameterize, (SinkSqlQuery | SinkSqlParameter) → true
| SanShellEscape, (SinkShellCommand | SinkShellArg) → true
| SanUrlEncode, SinkHttpRequest → true
| SanPathCanonicalize, (SinkFilePath | SinkFileOpen) → true
| SanInputValidation _, _ → true  (* Depends on pattern *)
| _, _ → false

```

Vulnerability Types and Analysis Results

```

type vulnerability_type =
| VulnSqlInjection
| VulnCommandInjection
| VulnXss
| VulnPathTraversal
| VulnSsrif
| VulnDeserializationRce
| VulnLdapInjection
| VulnXPathInjection
| VulnLogInjection
| VulnReDoS
| VulnOpenRedirect
| VulnHeaderInjection
| VulnTemplateInjection

let sink_to_vuln (sink : taint_sink) : vulnerability_type =
match sink with
| SinkSqlQuery | SinkSqlParameter → VulnSqlInjection
| SinkShellCommand | SinkShellArg → VulnCommandInjection
| SinkHtmlOutput | SinkHtmlAttribute | SinkJavaScript → VulnXss
| SinkFilePath | SinkFileOpen → VulnPathTraversal
| SinkHttpRequest | SinkUrlFetch → VulnSsrif
| SinkDeserialize _ | SinkYamlLoad | SinkPickle → VulnDeserializationRce
| SinkLdapQuery → VulnLdapInjection
| SinkXPathQuery → VulnXPathInjection
| SinkLog → VulnLogInjection
| SinkRegexPattern → VulnReDoS
| SinkEval | SinkExec → VulnCommandInjection
| _ → VulnXss  (* Default *)

type taint_flow = {
  source : taint_source;
  source_location : node_id;
  sink : taint_sink;
  sink_location : node_id;
  path : list node_id;  (* Nodes on the taint path *)
  vuln_type : vulnerability_type;
  confidence : float;  (* 0.0 - 1.0, LEGACY - see Section 12.3 for Manifest/Latent *)
}

type taint_analysis_result = {
  flows : list taint_flow;
  source_count : nat;
  sink_count : nat;
  sanitizer_count : nat;
}

```

The taint analysis is formulated as an IFDS problem (see Section 4.1 for the IFDS framework). The key insight is that taint facts form a finite set (at most one fact per variable/access path), making the analysis tractable. The flow function propagates taint through assignments, kills taint at sanitizers, and generates taint at sources.

Taint Analysis via IFDS

```

val run_taint_analysis :
  cpg →
  sources:(node_id → option taint_source) →
  sinks:(node_id → option taint_sink) →
  sanitizers:(node_id → option sanitizer) →
  taint_analysis_result

let run_taint_analysis cpg sources sinks sanitizers =
  (* Build IFDS problem *)
  let problem = {
    supergraph = cpg;
    domain = (* all taint facts *);
    zero = TaintZero;
    flow_function = fun edge d →
      let n = edge.source in
      (* Check for source *)
      begin match sources n with
      | Some src →
          let var = get_assigned_var cpg n in
          Set.add (TaintedVar var src) (Set.singleton d)
      | None → Set.singleton d
      end
      |>
      (* Check for sanitizer *)
      begin match sanitizers n with
      | Some san →
          Set.filter (fun t → not (sanitizes san t))
      | None → identity
      end
      |>
      (* Propagate through assignments *)
      propagate_taint cpg n;
      (* ... call/return flows ... *)
  } in
  (* Solve IFDS *)
  let solution = solve problem in
  (* Check for tainted values at sinks *)
  let flows = ref [] in
  Map.iter (fun node facts →
    match sinks node with
    | Some sink →
        Set.iter (fun fact →
          match fact with
          | TaintedVar var src →
              flows := {
                source = src;
                source_location = 0; (* TODO: track *)
                sink = sink;
                sink_location = node;
                path = []; (* TODO: reconstruct *)
                vuln_type = sink_to_vuln sink;
                confidence = 1.0;
              } :: !flows
          | _ → ()
        ) facts
    | None → ()
  ) solution;
  { flows = !flows;
    source_count = count_sources cpg sources;
    sink_count = count_sinks cpg sinks;
    sanitizer_count = count_sanitizers cpg sanitizers }

```

28.3 Hybrid Thin Slicing

TAJ Insight [Tripp09]

Full taint analysis is expensive. Most paths are irrelevant. **Thin slicing**: Only follow dependencies that are *relevant* to the sink.

Example:

```
x = user_input()      # Source
y = sanitize(x)       # Sanitizer
z = y + " suffix"     # Relevant
w = unrelated()       # NOT relevant
log(w)                # NOT relevant
query(z)              # Sink
```

Full backward slice from query(z):

- query(z) ← z ← y ← sanitize ← x ← user_input ✓ Relevant
- query(z) ← z ← " suffix" Partially relevant
- log(w) ← w ← unrelated() × Not relevant!

Thin slice: Only follow z backward, ignoring w.

Implementation:

1. Start from sink argument
2. Track which variables are “relevant”
3. Only follow data dependencies for relevant variables
4. Stop at sanitizers for the sink type

This dramatically reduces analysis scope.

Thin Slicing Implementation

```
val thin_slice_taint :
  cpg →
  sink_node:node_id →
  sink_arg:string →
  sanitizers:(node_id → option sanitizer) →
  set node_id  (* Nodes in the thin slice *)

let thin_slice_taint cpg sink_node sink_arg sanitizers =
  let rec slice visited frontier relevant_vars =
    if Set.is_empty frontier then visited
    else
      (* Get predecessors via data dependence *)
      let edges = get_data_dep_edges cpg frontier in
      (* Filter to relevant variables only *)
      let relevant_edges = List.filter (fun e →
        let defined_var = get_defined_var cpg e.source in
        Set.mem defined_var relevant_vars
      ) edges in
      (* Stop at sanitizers *)
      let unsanitized = List.filter (fun e →
        match sanitizers e.source with
        | Some san → not (sanitizer_protects san (get_sink_type sink_node))
        | None → true
      ) relevant_edges in
      let new_nodes = Set.of_list (List.map (fun e → e.source) unsanitized) in
      let new_nodes' = Set.diff new_nodes visited in
      (* Update relevant variables *)
      let new_vars = Set.concat_map (fun n →
        get_used_vars cpg n
      ) new_nodes' in
```

```

    slice (Set.union visited new_nodes')
      new_nodes'
      (Set.union relevant_vars new_vars)
  in
  slice (Set.singleton sink_node)
    (Set.singleton sink_node)
    (Set.singleton sink_arg)

```

Priority-Driven Taint Propagation (TAJ 2009)

```

(* Key insight: Most taint flows are LOCAL - tainted values are typically
   used close to where they were introduced. Processing near-source nodes
   first finds vulnerabilities faster and enables early termination.
   This is the "locality-of-taint" principle: prioritize analysis of nodes
   that are closer (in the PDG) to known taint sources.
   Cross-reference: Uses IFDS foundation from Section 4.1 (Reps 1995) *)

module BrrrMachine.Security.PriorityTaint

type taint_priority_queue = {
  queue : priority_queue (node_id * taint_fact);
  distance_from_source : map node_id nat;
  source_nodes : set node_id;
}

(* Initialize with sources at priority 0 *)
val init_priority_queue : cpg → (node_id → option taint_source) → taint_priority_queue
let init_priority_queue cpg sources =
  let source_set = Set.filter (fun n → Option.is_some (sources n)) (all_nodes cpg) in
  let initial_dist = Set.fold (fun acc n → Map.add n 0 acc) Map.empty source_set in
  {
    queue = PQueue.of_list (List.map (fun n → (n, initial_taint n, 0))
      (Set.to_list source_set));
    distance_from_source = initial_dist;
    source_nodes = source_set;
  }

(* Priority = distance from nearest source (lower = higher priority) *)
val priority_of_node : taint_priority_queue → node_id → nat
let priority_of_node pq node =
  match Map.find node pq.distance_from_source with
  | Some d → d
  | None → max_int (* Unknown nodes have lowest priority *)

(* Process taint in priority order - finds vulnerabilities faster *)
val process_priority_order :
  cpg →
  taint_priority_queue →
  sinks:(node_id → option taint_sink) →
  max_findings:nat →
  list taint_flow

let rec process_priority_order cpg pq sinks max_findings =
  if max_findings = 0 then []
  else match PQueue.pop pq.queue with
  | None → [] (* Queue exhausted *)
  | Some ((node, fact), queue') →
    let pq' = { pq with queue = queue' } in
    (* Check if we reached a sink *)
    match sinks node with
    | Some sink →
      let flow = { source = fact.source; sink = sink; path = [];
        confidence = 1.0 } in
      flow :: process_priority_order cpg pq' sinks (max_findings - 1)

```

```

| None →
  (* Propagate to successors with updated priorities *)
  let successors = get_taint_successors cpg node fact in
  let pq'' = List.fold_left (fun acc (succ_node, succ_fact) →
    let acc' = update_distance acc node succ_node in
    let priority = priority_of_node acc' succ_node in
    { acc' with queue = PQueue.push (succ_node, succ_fact) priority acc'.queue }
  ) pq' successors in
  process_priority_order cpg pq'' sinks max_findings

```

Empirical Result (TAJ 2009)

Priority-driven processing finds first vulnerability **10–100x faster** than breadth-first or depth-first orderings on large codebases.

28.3.1 Taint Finding Classification

Cross-reference: Section 12.3 (Manifest/Latent Framework)

After detecting taint flow via IFDS (Section 4.1) or TAJ priority-driven analysis, classify each finding using the Manifest/Latent framework:

1. Build ISL triple from taint path:

$$[\text{emp}] \text{ source-to-sink-path } [tainted_at_sink; E_r]$$

2. Check manifest conditions ([Le22] Definition 3.3):

- Empty presumption (any input triggers)?
- Satisfiable result (path reachable)?
- All heap locs existential?
- Pure constraints universally satisfiable?

3. Report based on classification:

- **Manifest taint** → CRITICAL (0% FP guaranteed by theorem)
- **Latent taint** → Report with triggering context
- **Refuted** → Filter out (was false positive from IFDS)

This eliminates the heuristic “confidence score” approach. A manifest taint vulnerability is *proven* to be exploitable.

28.4 Extended Information Flow Control

Beyond Simple Taint: Complete IFC for Real-World Security

8.1.4.1 Implicit Flow (PC Label): Flows through control structure

if (h) { l := 1 } — leaks h via branch taken

8.1.4.2 Multi-Principal (DLM): Multiple independent flow policies

{alice: alice,hospital; bob: bob,insurance}

8.1.4.3 Declassification: Controlled release of secrets

declassify(secret, newLabel) with robust declassification

8.1.4.4 Covert Channels: Timing, termination, side channels

while(secret){} leaks via termination; timing attacks on crypto

8.1.4.5 Concurrent IFC: Thread interleavings as channels

Scheduler-dependent flows, internal timing

28.4.1 Implicit Flow Analysis via PC Label

Foundational Paper

[Sabelfeld03] (Sabelfeld & Myers 2003)

Critical Insight

Explicit-only taint analysis is **insufficient** for security. Implicit flows through control structure leak secrets equally well.

Example — Both are insecure:

```
l := h;                                // EXPLICIT flow (synthesis tracks this)

if (h == secret) {                      // IMPLICIT flow (synthesis MISSES this!)
  l := 1;
}
```

In the second case, the *value* of *h* is revealed by whether *l* becomes 1. The assignment itself uses only low data, but the *branch* depends on high data.

Definition 28.5 (PC Label). The *program counter* (pc) label tracks the security level of the current control context. It is elevated when entering high branches.

Rule: An assignment is secure only if:

$$\text{level}(\text{rhs}) \sqcup \text{pc} \sqsubseteq \text{level}(\text{lhs})$$

This rejects `l := 1` when *pc* is high and *l* is low.

Implicit Flow Analysis

```
(* =====
  IMPLICIT FLOW ANALYSIS
  Source: Sabelfeld & Myers 2003
  ===== *)
module BrrrMachine.Security.ImplicitFlow

(* Extended taint state: includes PC label *)
type implicit_flow_state = {
  var_label : map string security_level;
  pc : security_level;          (* Program counter security level *)
  pc_sources : list node_id;    (* Where PC elevation came from *)
}

(* Security level operations *)
let level_join l1 l2 = match l1, l2 with
| SHigh, _ | _, SHigh → SHigh
| SLow, SLow → SLow

let level_leq l1 l2 = match l1, l2 with
| SLow, _ → true
| SHigh, SHigh → true
| _, _ → false

(* Transfer function with implicit flow detection *)
type implicit_finding =
| ExplicitFlow : var:string → source:node_id → implicit_finding
| ImplicitFlow : var:string → pc_source:node_id → implicit_finding
| TerminationChannel : loop_loc:node_id → implicit_finding

val transfer_implicit : cpq_node → implicit_flow_state →
  (implicit_flow_state * list implicit_finding)

let transfer_implicit node state =
  match node.kind with
  (* Assignment: check BOTH explicit AND implicit flow *)
  | NAssign var →
    let rhs_level = compute_expr_level state (get_rhs node) in
```

```

(* CRITICAL: Include PC in effective level *)
let effective_level = level_join rhs_level state.pc in
let var_declared = get_declared_label var in
if not (level_leq effective_level var_declared) then
  let finding =
    if state.pc = SHigh && rhs_level = SLow then
      ImplicitFlow var (List.hd state.pc_sources) (* Pure implicit *)
    else
      ExplicitFlow var (get_source_node state) (* Explicit *)
  in
  (state, [finding])
else
  ({ state with var_label = Map.add var effective_level state.var_label }, [])

(* Conditional: ELEVATE PC *)
| NIf →
  let cond_level = compute_expr_level state (get_condition node) in
  let new_pc = level_join state.pc cond_level in
  ({ state with pc = new_pc;
    pc_sources = if cond_level = SHigh
                  then node.id :: state.pc_sources
                  else state.pc_sources }, [])

(* Loop: ELEVATE PC + warn about termination channel *)
| NWhile →
  let cond_level = compute_expr_level state (get_condition node) in
  let findings = if cond_level = SHigh
                  then [TerminationChannel node.id]
                  else [] in
  ({ state with pc = level_join state.pc cond_level;
    pc_sources = node.id :: state.pc_sources }, findings)

(* Scope exit: RESTORE PC (static analysis advantage!) *)
| NJoin →
  ({ state with pc = get_enclosing_pc node;
    pc_sources = get_enclosing_sources node }, [])
| _ → (state, [])

(* Noninterference soundness theorem --- See Section 12.2 for full statement
implicit_analysis_sound: If analysis reports no leaks, noninterference holds. *)

```

28.4.2 Multi-Principal Labels (DLM)

Foundational Paper

[Myers97] (Decentralized Label Model)

Why DLM, Not Simple Taint

Simple taint lattice: $T_{\perp} < T_{\text{Untainted}} < T_{\text{Maybe}} < T_{\text{Tainted}}$

Problem: Cannot express “Alice’s data readable by Alice+Hospital, Bob’s data readable by Bob+Insurance” — two independent policies.

DLM Solution: Labels have *multiple owners*, each with their own reader sets.

Example: {alice: alice,hospital; bob: bob,insurance} — two independent flow policies in the same label.

DLM *subsumes* simple taint:

- $T_{\text{Tainted}} = \{\text{source} : \text{source}\}$ (only source can read)
- $T_{\text{Untainted}} = \{\}$ (empty label, anyone can read)

Decentralized Label Model (DLM)

```
(* =====
   DECENTRALIZED LABEL MODEL (DLM)
   Source: Myers 1997
   ===== *)
module BrrrMachine.Security.DLM

type principal = string

(* DLM LABEL: Multiple owners, each with their own reader set
   Label = {o1: r1,r2; o2: r3,r4}
   means: owner o1 allows readers r1,r2
          owner o2 allows readers r3,r4 *)
type dlm_label = {
  owners : set principal;
  readers : principal → set principal;  (* readers for each owner *)
}

(* Empty label: anyone can read (public) *)
let dlm_public : dlm_label = { owners = Set.empty; readers = fun _ → Set.all }

(* Tainted label: only source can read *)
let dlm_tainted (source : principal) : dlm_label = {
  owners = Set.singleton source;
  readers = fun o → if o = source then Set.singleton source else Set.all;
}

(* Effective reader set: intersection of all owner's reader sets *)
let effective_readers (l : dlm_label) : set principal =
  if Set.is_empty l.owners then Set.all
  else Set.fold (fun acc owner →
    Set.inter acc (l.readers owner)
  ) Set.all l.owners

(* DLM LATTICE OPERATIONS *)

(* Join: union owners, intersect readers --- more restrictive *)
let dlm_join (l1 l2 : dlm_label) : dlm_label = {
  owners = Set.union l1.owners l2.owners;
  readers = fun o →
    let r1 = if Set.mem o l1.owners then l1.readers o else Set.all in
    let r2 = if Set.mem o l2.owners then l2.readers o else Set.all in
    Set.inter r1 r2;
}

(* Restriction check: l1 ≤ l2 iff l1 allows at least what l2 allows *)
let dlm_leq (l1 l2 : dlm_label) : bool =
  (* l2 has subset of owners *)
  Set.subset l2.owners l1.owners &&
  (* For each owner in l2, l2's readers subset of l1's readers *)
  Set.for_all (fun o →
    Set.subset (l2.readers o) (l1.readers o)
  ) l2.owners
```

Principal Hierarchy and Declassification

```
(* Principal hierarchy (acts-for relation)
   Enables groups, roles, and delegation.
   If p acts-for q, then p can do anything q can do. *)
type principal_hierarchy = {
  principals : set principal;
  acts_for : principal → principal → bool;
  (* Reflexive and transitive *)
  acts_for_refl : squash (∀ p. acts_for p p);
```

```

acts_for_trans : squash (∀ p q r.
  acts_for p q && acts_for q r => acts_for p r);
}

(* Can process acting as 'actor' declassify data owned by 'owner'? *)
let can_declassify (ph : principal_hierarchy) (actor owner : principal) : bool =
  ph.acts_for actor owner

(* DECLASSIFICATION: Authorized release of information
  Rule 1 (Restriction): Anyone can make data MORE restrictive
  Rule 2 (Declassification): Only owner (or delegate) can make LESS restrictive *)
type relabel_result = RelabelOK | RelabelDenied of string

let check_relabel
  (ph : principal_hierarchy)
  (actor : principal)
  (from_label to_label : dlm_label)
  : relabel_result =
  if dlm_leq from_label to_label then
    (* Restriction: always allowed (making more restrictive) *)
    RelabelOK
  else
    (* Declassification: need authority *)
    let owners_losing_restriction =
      Set.filter (fun o →
        not (Set.subset (to_label.readers o) (from_label.readers o))
      ) from_label.owners in
    if Set.for_all (can_declassify ph actor) owners_losing_restriction then
      RelabelOK
    else
      RelabelDenied ("Actor " ^ actor ^
        " cannot declassify for all affected owners")

```

28.4.3 Declassification and Endorsement

Foundational Papers

[Zdancewic01], [Chong04] (Security Policies for Downgrading), [Sabelfeld09]

Declassification: Secure Downgrading under Adversarial Conditions

Key Insight: Declassification is not just “allow this flow” — it requires precise specification of **WHAT**, **WHERE**, **WHO**, and **WHEN** information is released. Without robustness, attackers can manipulate what gets declassified.

Cross-reference: Section 12.34 for complete F* formalization.

The Fundamental Problem

Strict noninterference is **too restrictive** for real systems. Real systems *must* release some secrets:

- Password comparison results (boolean)
- Encrypted versions of data
- Statistical aggregates over private data
- Authentication tokens derived from secrets

But **uncontrolled** declassification creates “laundering” vulnerabilities where attackers can influence *what* gets declassified.

Semantic Security under Declassification

Traditional noninterference:

$$M_1 \sim_L M_2 \implies P(M_1) \sim_L P(M_2)$$

“Low-equivalent inputs produce low-equivalent outputs”

Security under declassification [Chong04]:

$$M_1 \sim_L M_2 \implies \text{visible}(P(M_1)) \sim_L \text{visible}(P(M_2))$$

where $\text{visible}(\text{trace})$ extracts *only* the declassified values.

The key change: we don’t require *all* outputs to be equivalent, only the *observable* outputs (public + explicitly declassified).

Delimited Release: WHAT Can Be Released

`declassify(expression, target_label)`

The **expression** defines an “escape hatch” — only the information computed by that expression may be released.

- **Good:** `declassify(password == guess, public)`
Only releases **boolean** (one bit), not the password.
- **Bad:** `declassify(password, public)`
Releases entire password — too much!

Robust Declassification: Integrity-Based Security

The Attack Model: Attacker controls LOW inputs (low integrity) but should **not** be able to:

1. Influence *which* secret gets declassified
2. Influence *how much* of a secret gets declassified

Non-Robust (Vulnerable):

```
if (attacker_controlled) {
    declassify(secret1, public)
} else {
    declassify(secret2, public)
}
// Attacker controls WHICH secret is released!
```

Robust Declassification Condition: For expression e in `declassify(e, L)`:

- The PC (program counter) must have **high integrity** at declassification
- Expression e must **not** depend on low integrity inputs
- The **path** to declassification must be attacker-independent

Critical Insight

Robustness requires tracking **integrity**, not just confidentiality. The synthesis upgrades to 4-point security labels.

The Four Dimensions of Declassification [Sabelfeld09]

1. **WHAT** — What information is released (delimited release)
Escape hatches define the *function* of secrets that is released.
2. **WHERE** — At what program points declassification is allowed
Only specific code locations may perform declassification.
E.g., only the authentication module may release auth results.
3. **WHO** — By whose authority
Principals must have ownership of the secret to authorize release.
Integrates with DLM (Section 28.4.2): acts-for hierarchy.
4. **WHEN** — Under what conditions
State-dependent policies: “release only after encryption”
Temporal constraints: “release only during session”

5. **SPECULATION-AWARE** (SPECTECTOR extension, Section 28.4.6): Declassification must **not** occur on **speculative** paths!

Endorsement: The Integrity Dual

Endorsement is the dual of declassification for **integrity**:

- **Declassification:** $high_confidentiality \rightarrow low_confidentiality$
- **Endorsement:** $low_integrity \rightarrow high_integrity$

```
endorse(untrusted_input, high_integrity)
```

Robust Endorsement Condition (dual of robust declassification): High-*confidentiality* code cannot influence *what* gets endorsed. This prevents secrets from leaking via the choice of trusted values.

Sanitizers vs Declassification: Critical Distinction

These are DIFFERENT operations — do not conflate them!

Sanitizer: Changes **value**, keeps **label**

```
sanitize(user_input) -> safe_value
```

The value is *transformed* (e.g., HTML escaping). The security label remains unchanged (still “from user input”).

Declassification: Keeps **value**, changes **label**

```
declassify(secret, public) -> secret
```

The value is *unchanged*. The security label is downgraded (high \rightarrow low confidentiality).

Both are needed for complete security analysis:

- Sanitizers prevent injection attacks (transform dangerous values)
- Declassification allows controlled information release (change policy)

Declassification and Endorsement Analysis

```
(* =====
  DECLASSIFICATION AND ENDORSEMENT ANALYSIS
  Source: Chong & Myers 2004 "Security Policies for Downgrading"

  KEY INSIGHT: Security under declassification requires tracking BOTH
  confidentiality AND integrity. Robustness is an INTEGRITY property.

  Cross-reference: Section 12.34 for complete theorems and proofs.
  ===== *)
module BrrrMachine.Security.Declassification

(* SECURITY LABELS WITH BOTH CONFIDENTIALITY AND INTEGRITY
   This is the critical upgrade from simple taint: 4-point lattice. *)
type confidentiality_level = CLow | CHigh
type integrity_level = ILow | IHigh

type security_label = {
  conf : confidentiality_level;    (* Can information be observed? *)
  integ : integrity_level;         (* Can information be trusted? *)
}

(* Label ordering: (c1, i1) ≤ (c2, i2) iff c1 ≤ c2 AND i1 ≥ i2
   Higher confidentiality = more secret = higher in conf lattice
   Higher integrity = more trusted = LOWER risk = inverted ordering *)
val label_leq : security_label → security_label → bool
let label_leq l1 l2 =
```

```

    (l1.conf = CLow || l2.conf = CHigh) &&    (* conf: Low ≤ High *)
    (l1.integ = IHigh || l2.integ = ILow)      (* integ: High ≤ Low (inverted!) *)

val label_join : security_label → security_label → security_label
let label_join l1 l2 = {
  conf = if l1.conf = CHigh || l2.conf = CHigh then CHigh else CLow;
  integ = if l1.integ = ILow || l2.integ = ILow then ILow else IHigh;
}

```

Declassification Policy and Violations

```

(* DECLASSIFICATION POLICY: Full Four-Dimensional Specification *)
type declassification_policy = {
  what : list escape_hatch;          (* WHAT can be released *)
  where : set node_id;               (* WHERE declassification allowed *)
  who : set principal;               (* WHO can authorize *)
  when_conditions : list ir_expr;    (* WHEN conditions required *)
  robust_required : bool;            (* Require integrity-based robustness *)
}

(* VIOLATION TYPES: Comprehensive Error Reporting *)
type declassify_violation =
  (* Robustness violations - attacker influence detected *)
  | LowIntegrityControl : declass_node:node_id → influencing_node:node_id →
    declassify_violation
    (* Low-integrity code path can reach declassification *)
  | LowIntegrityData : var:string → declass_node:node_id →
    declassify_violation
    (* Low-integrity data flows into declassified expression *)

  (* Policy violations *)
  | ExcessiveRelease : released:ir_expr → allowed:ir_expr →
    declassify_violation
    (* Expression releases more than escape hatch allows *)
  | Unauthorized : required:principal → actual:principal →
    declassify_violation
    (* Principal lacks authority to declassify *)
  | WrongLocation : actual:node_id → allowed:set node_id →
    declassify_violation
    (* Declassification at unauthorized code location *)
  | ConditionNotMet : required:ir_expr → actual_state:ir_state →
    declassify_violation
    (* When-condition not satisfied *)

```

Robustness Check: Integrity-Based Verification

```

(* This is the key contribution of Chong & Myers 2004. *)
val check_robustness :
  cpg:cpg →
  declass_node:node_id →
  hatch:escape_hatch →
  option declassify_violation

let check_robustness cpg declass_node hatch =
  (* CHECK 1: PC integrity must be HIGH at declassification point
   If low-integrity code can reach this point, attacker controls
   WHETHER declassification occurs. *)
  let pc_integ = get_pc_integrity cpg declass_node in
  if pc_integ = ILow then
    let influencing = get_pc_integrity_source cpg declass_node in
    Some (LowIntegrityControl declass_node influencing)
  else
    (* CHECK 2: No low-integrity data dependencies
   If the declassified expression depends on attacker-controlled data,
   attacker influences WHAT is declassified. *)

```

```

let expr = get_declassified_expr cpg declass_node in
let deps = backward_slice cpg expr in
let low_integ_deps = Set.filter (fun n →
  (get_label cpg n).integ = ILow
) deps in
if not (Set.is_empty low_integ_deps) then
  let bad_var = get_var_name (Set.choose low_integ_deps) in
  Some (LowIntegrityData bad_var declass_node)
else
  (* CHECK 3: Expression matches escape hatch semantically *)
  if not (expr_semantically_matches expr hatch.expr) then
    Some (ExcessiveRelease expr hatch.expr)
  else
    (* CHECK 4: Authority check *)
    let current_principal = get_current_principal cpg declass_node in
    if not (has_authority current_principal hatch.authority) then
      Some (Unauthorized hatch.authority current_principal)
    else
      None

```

28.4.4 Covert Channels

Foundational Papers

[Agat00] (Timing), [Sabelfeld00] (Probabilistic)

Covert Channels: Information leaks through *side effects* of computation.

1. **Termination Channel** (already in Section 28.4.1):

```
while (secret) { loop_forever(); }
```

Leaks secret via whether program terminates.

2. **Timing Channel:**

```
if (secret) { expensive_operation(); }
else { cheap_operation(); }
```

Leaks secret via execution time difference.

3. **Cache Timing:**

```
array[secret * 256] // Cache side channel (Spectre-style)
```

Memory access patterns leak secret.

See Section 28.4.6 for speculative execution attacks (Spectre).

See Section 28.4.7 for constant-time verification (CT-Verif).

4. **Storage Channel:**

Allocation patterns, file sizes, etc.

Definition 28.6 (Timing-Sensitive Noninterference [Agat00]). Programs must take *same time* regardless of secrets.

Requirement: All branches on secrets must be “balanced”:

```
if (secret) { A } else { B }
```

requires: $time(A) = time(B)$

Cross-copying transformation:

```
if (secret) { A; skip_B } else { skip_A; B }
```

where `skip_X` takes same time as `X` but does nothing.

Timing Security Layers (complete coverage):

1. Instruction timing ([Agat00]) — balance branch execution
2. Memory access patterns (CT-Verif, Section 28.4.7) — constant-time verification
3. Speculative execution (SPECTECTOR, Section 28.4.6) — microarchitectural security
4. Countermeasure verification — LFENCE, SLH placement

Timing Channel Analysis

```

type timing_violation =
| UnbalancedBranch : branch:node_id → diff:time_estimate → timing_violation
| SecretDependentLoop : loop:node_id → timing_violation
| VariableTimeOp : op:node_id → op_name:string → timing_violation

type time_estimate =
| Constant of nat
| Linear of string      (* Linear in variable *)
| Unknown

val analyze_timing : cpg → list timing_violation
let analyze_timing cpg =
  let violations = ref [] in
  (* Check all branches on secret conditions *)
  iter_nodes cpg (fun node →
    match node.kind with
    | NIf when condition_is_secret cpg node →
      let then_time = estimate_time cpg (get_then_branch node) in
      let else_time = estimate_time cpg (get_else_branch node) in
      if not (times_equal then_time else_time) then
        violations := UnbalancedBranch node (time_diff then_time else_time)
        :: !violations
    | NWhile when condition_is_secret cpg node →
      violations := SecretDependentLoop node :: !violations
    | NCall when is_variable_time_op cpg node →
      if any_arg_is_secret cpg node then
        violations := VariableTimeOp node (get_func_name node) :: !violations
    | _ → ()
  );
  !violations

(* Variable-time operations to flag *)
let variable_time_ops = [
  "strcmp"; "memcmp";      (* String comparison - timing varies *)
  "division"; "modulo";    (* Some CPUs have variable-time div *)
  "branch_on_secret";      (* Any conditional on secret *)
]

```

28.4.5 Concurrent Information Flow

Foundational Papers

[Smith98] (Smith & Volpano 1998), [Russo06] (Russo & Sabelfeld 2006)

Concurrent IFC Challenges:

1. Internal Timing:

```

Thread 1: if (secret) { sleep(1000); } l1 := 1;
Thread 2: sleep(500); l2 := l1;

```

Thread 2 observes *whether* Thread 1 took the branch via timing.

2. Scheduler Channels:

Thread scheduling decisions may depend on secrets. Especially problematic with priority inheritance.

3. Shared Memory Races:

Thread 1: `if (secret) { x := 1; }`
 Thread 2: `l := x;`

Value of `l` depends on interleaving *and* `secret`.

Solutions:

1. **Observational Determinism** ([McLean92]): Low-equivalent inputs \Rightarrow same low observations regardless of scheduler
2. **Strong Security** (Sabelfeld & Sands): Require timing-insensitivity *plus* scheduler-independence
3. **Practical Approach:**
 - Ban races on shared variables with different security levels
 - Require synchronized access to shared secrets
 - Flag any shared variable accessed by threads with different clearances

Concurrent IFC Analysis

```
type concurrent_ifc_violation =
  | SharedSecretRace : var:string → threads:list thread_id →
    concurrent_ifc_violation
  | InternalTimingLeak : high_thread:thread_id → observer:thread_id →
    concurrent_ifc_violation
  | UnsyncedCrossLevelAccess : var:string → writer_level:security_level →
    reader_level:security_level → concurrent_ifc_violation

val analyze_concurrent_ifc : cpg → list concurrent_ifc_violation
let analyze_concurrent_ifc cpg =
  let violations = ref [] in
  let shared_vars = find_shared_variables cpg in
  Set.iter (fun var →
    let accessors = get_accessing_threads cpg var in
    let levels = List.map (fun t → get_thread_clearance t) accessors in
    (* Check 1: Race on secret *)
    if get_var_level var = SHigh && has_race cpg var then
      violations := SharedSecretRace var accessors :: !violations;
    (* Check 2: Cross-level access without sync *)
    let writers = get_writers cpg var in
    let readers = get_readers cpg var in
    List.iter (fun w →
      List.iter (fun r →
        if get_level w ≠ get_level r && not (synced_access cpg w r) then
          violations := UnsyncedCrossLevelAccess var (get_level w)
            (get_level r) :: !violations
      ) readers
    ) writers;
  ) shared_vars;
  !violations
```

28.4.6 Speculative Execution Security

Foundational Paper

[Guarnieri20] (SPECTECTOR — Guarnieri, Kopf, Morales, Reineke, Sanchez 2020)

Speculative execution creates covert channels through *microarchitectural* state. Even with perfect constant-time code, speculative execution can leak secrets through cache side channels (Spectre family attacks).

The Problem: Modern CPUs speculatively execute past branches before they resolve. On misprediction, **architectural** state rolls back. But **microarchitectural** state (caches) is *not* rolled back. This creates observable side channels.

Spectre Gadget Example:

```
if (x < array1_size) {           // Mispredicted as true when false
  y = array1[x];                 // Speculatively loads secret
  z = array2[y * 256];           // Leaks secret via cache
}                                // Rollback - but cache state persists!
```

Definition 28.7 (Speculative Non-Interference (SNI)). Two low-equivalent initial memories produce distinguishable speculative observations *only if* they produce distinguishable non-speculative observations.

Formally:

$$\forall M_1, M_2. \text{low_equiv}(M_1, M_2) \implies (\text{obs_eq}(\text{run_spec}(P, M_1), \text{run_spec}(P, M_2)) \implies \text{obs_eq}(\text{run_std}(P, M_1), \text{run_std}(P, M_2)))$$

Key Insight: Worst-Case Branch Predictor Abstraction

Instead of modeling specific CPUs, use a predictor that **always mispredicts**. Sound: secure against worst-case \Rightarrow secure against any real predictor.

This is a Galois connection:

$$\begin{array}{c} \text{Concrete predictors } \{1\text{-bit, } 2\text{-bit, neural, ...}\} \\ \downarrow (\text{abstraction}) \\ \text{Abstract predictor: always-mispredict (worst case)} \end{array}$$

Observer Model (compatible with Section 28.4.4):

1. Memory access addresses (cache timing)
2. Jump targets (branch prediction)

Countermeasure Verification: SPECTECTOR can verify that mitigations are correctly placed:

- **LFENCE:** Serializing instruction forces rollback
- **SLH** (Speculative Load Hardening): Masks speculative values
- **Retpoline:** Return trampoline for indirect branches

Speculative Execution Security (SPECTECTOR)

```
module BrrrMachine.Security.Speculative

(* Observation types for cache timing attacks *)
type observation =
  | ObsMemAccess : addr:nat → observation
  | ObsJumpTarget : target:nat → observation

type obs_trace = list observation

(* Speculation state *)
type spec_state = {
  pc : nat;
  regs : map nat int;
  mem : map nat int;
  spec_depth : nat; (* Current speculation depth *)
  mispredict_stack : list (nat * map nat int); (* Recovery points *)
  trace : obs_trace; (* Accumulated observations *)
}

(* Speculation window bound (typical for Intel CPUs) *)
let max_spec_window : nat = 200

(* Speculative step: fork on BOTH branch outcomes *)
```

```

val spec_branch : spec_state → nat → nat → list spec_state
let spec_branch st true_target false_target =
  if st.spec_depth < max_spec_window then
    let cond_val = eval_condition st in
    let (correct, wrong) = if cond_val then (true_target, false_target)
                           else (false_target, true_target) in
    [
      (* Non-speculative: take correct path *)
      record_jump { st with pc = correct } correct;
      (* Speculative: take WRONG path (misprediction) *)
      record_jump {
        st with
        pc = wrong;
        spec_depth = st.spec_depth + 1;
        mispredict_stack = (correct, st.regs) :: st.mispredict_stack
      } wrong
    ]
  else
    (* Speculation window exhausted - rollback *)
    rollback st

```

Spectre Gadget Detection and Countermeasure Verification

```

type spectre_gadget = {
  secret_load : node_id;      (* Load that accesses secret *)
  dependent_access : node_id; (* Memory access dependent on secret *)
  speculation_path : list node_id; (* Path through speculative execution *)
}

val detect_spectre_gadgets :
  cpg → (nat → security_level) → nat → list spectre_gadget
let detect_spectre_gadgets cpg policy window =
  (* Symbolic execution under speculative semantics *)
  let spec_states = symbolic_spec_execute cpg window in
  List.filter_map (fun state →
    let spec_loads = get_speculative_secret_loads state policy in
    let dependent = get_dependent_memory_accesses state spec_loads in
    if List.length dependent > 0 then
      Some {
        secret_load = List.hd spec_loads;
        dependent_access = List.hd dependent;
        speculation_path = state.path
      }
    else None
  ) spec_states

type countermeasure =
  | LFENCE : loc:node_id → countermeasure
  | SLH : loc:node_id → masked_reg:nat → countermeasure
  | Retpoline : loc:node_id → countermeasure

val countermeasure_blocks_gadget :
  cpg → spectre_gadget → countermeasure → bool
let countermeasure_blocks_gadget cpg gadget cm =
  match cm with
  | LFENCE loc →
    (* LFENCE must dominate the vulnerable load *)
    dominates cpg loc gadget.secret_load &&
    (* No speculative path bypasses the fence *)
    not (∃_speculative_bypass cpg loc gadget.speculation_path)
  | SLH loc masked_reg →
    (* SLH must mask the value used in dependent access *)
    slh_masks_dependency cpg loc gadget.dependent_access masked_reg
  | Retpoline loc →

```

```

(* Retpoline must replace indirect branch *)
is_indirect_branch cpg loc &&
loc 'List.mem' gadget.speculation_path

(* Theorem: Correctly placed countermeasure establishes SNI *)
val countermeasure_soundness :
  cpg:cpg →
  gadget:spectre_gadget →
  cm:countermeasure →
  policy:(nat → security_level) →
  window:nat →
  Lemma (requires countermeasure_blocks_gadget cpg gadget cm)
    (ensures sni (apply_countermeasure (cpg_to_ir cpg) cm) policy window)

```

28.4.7 Constant-Time Verification via Product Programs

Foundational Paper

[Almeida16] (CT-Verif — Almeida, Barbosa, Barthe, Dupressoir, Emmi 2016)

Constant-Time Security: Execution time and memory access patterns must be independent of secret data.

The Problem:

1. Execution time variations leak via branch timing
2. Memory access patterns leak via cache timing
3. Variable-time operations (div, modulo) leak via operand-dependent timing
4. Compiler optimizations may introduce violations

Constant-Time is a 2-Safety Hyperproperty: Relates *two* executions of the same program. Cannot be verified by analyzing single executions.

Definition 28.8 (CT-Verif Insight: Product Program Reduction). Given program P , construct product program Q that:

- Maintains two shadow copies of state (original + shadow)
- Executes both copies in lockstep along identical control paths
- Asserts observations match at each step

P is constant-time $\iff Q$ is assertion-safe

Reduction verified in Coq.

Leakage Models (parameterized observer power):

1. **PC Model:** Only control flow visible (branch-prediction attacks)
2. **Memory Model:** PC + memory addresses (cache-timing attacks)
3. **Operand Model:** PC + addresses + operand values (variable-time instructions)

Output-Sensitive Security: Allow intentional leaks bounded by public outputs. Distinguishes benign leaks from true vulnerabilities.

Architecture-Aware Variable-Time Operations:

| Operation | x86_64 | ARM_v8 | RISC-V |
|-----------|--------|--------|--------|
| div | VAR | VAR | CONST |
| mod | VAR | VAR | CONST |
| fdiv | VAR | VAR | VAR |
| strcmp | VAR | VAR | VAR |
| memcmp | VAR | VAR | VAR |

Constant-Time Verification via Product Programs

```

module BrrrMachine.Security.ConstantTime

(* LEAKAGE MODELS *)
type leakage_model =
| LeakPC      (* Control flow only - branch prediction attacks *)
| LeakMemory  (* PC + memory addresses - cache timing attacks *)
| LeakOperand (* Full operand values - variable-time instructions *)

type ct_observation =
| ObsPC : pc:nat → ct_observation
| ObsMem : pc:nat → addr:nat → ct_observation
| ObsOp : pc:nat → addr:nat → operands:list nat → ct_observation

(* CONSTANT-TIME SECURITY DEFINITION *)
type program_state = map string int
type trace = list ct_observation

(* Two states are i-equivalent if they agree on public inputs *)
val i_equivalent : program_state → program_state → set string → bool
let i_equivalent s1 s2 public_vars =
  Set.for_all (fun v → s1 v = s2 v) public_vars

(* DEFINITION: Constant-time security *)
val is_constant_time :
  ir_program →
  leakage_model →
  secrets:set string →
  outputs:list public_output →
  bool
let is_constant_time prog model secrets outputs =
  let public = Set.complement secrets in
  ∀ s1 s2.
    i_equivalent s1 s2 public ==>
    (let (t1, s1') = execute_with_trace prog model s1 in
     let (t2, s2') = execute_with_trace prog model s2 in
     o_equivalent s1' s2' outputs ==> t1 = t2)

```

Product Program Construction

```

type product_state = {
  orig : program_state;
  shadow : program_state;
}

(* Rename variables to shadow namespace *)
val rename_to_shadow : ir_instr → ir_instr
let rename_to_shadow instr =
  map_vars (fun v → "shadow_" ^ v) instr

(* Construct observation assertion for leakage model *)
val observation_assertion : ir_instr → leakage_model → ir_instr
let observation_assertion instr model =
  match model with
  | LeakPC →
    IAssert (EBinOp Eq (EVar "pc") (EVar "shadow_pc"))
  | LeakMemory →
    let addr = get_memory_addr instr in
    let shadow_addr = get_shadow_memory_addr instr in
    IAssert (EBinOp And
      (EBinOp Eq (EVar "pc") (EVar "shadow_pc"))
      (EBinOp Eq addr shadow_addr))
  | LeakOperand →
    let ops = get_operands instr in

```

```

    let shadow_ops = get_shadow_operands instr in
    IAssert (all_equal ops shadow_ops)

(* Product program transformation *)
val construct_product_instr : ir_instr → leakage_model → list ir_instr
let construct_product_instr instr model =
[
  instr;                                (* Original instruction *)
  rename_to_shadow instr;               (* Shadow instruction *)
  observation_assertion instr model     (* Assert observations match *)
]

(* THEOREM: Product construction is sound and complete *)
val product_reduction_correct :
  prog:ir_program →
  model:leakage_model →
  secrets:set string →
  outputs:list public_output →
  Lemma (
    is_constant_time prog model secrets outputs ≤⇒
    is_assertion_safe (construct_product prog model)
  )

```

Constant-Time Violations and Analysis

```

type ct_violation =
| SecretBranch : condition:ir_expr → secret_deps:set string →
  location:node_id → ct_violation
| SecretMemoryAccess : address:ir_expr → secret_deps:set string →
  location:node_id → ct_violation
| VariableTimeOpOnSecret : op:string → operand:ir_expr →
  secret_deps:set string → location:node_id →
  ct_violation

val analyze_constant_time :
  cpg →
  model:leakage_model →
  secrets:set string →
  outputs:list public_output →
  list ct_violation
let analyze_constant_time cpg model secrets outputs =
  let prog = cpg_to_ir cpg in
  let product = construct_product prog model in
  let failed = find_failing_assertions product in
  List.map (fun (loc, _) →
    let instr = get_instr_at cpg loc in
    match categorize_violation instr secrets with
    | VBranch cond deps → SecretBranch cond deps loc
    | VMemAccess addr deps → SecretMemoryAccess addr deps loc
    | VVariableOp op operand deps → VariableTimeOpOnSecret op operand deps loc
  ) failed

(* Architecture-aware variable-time operation check *)
type architecture = X86_64 | ARM_v8 | RISC_V | PowerPC

val is_variable_time : string → architecture → bool
let is_variable_time op arch =
  match (op, arch) with
  | ("div", X86_64) | ("div", ARM_v8) → true
  | ("mod", X86_64) | ("mod", ARM_v8) → true
  | ("fdiv", _) → true
  | ("strcmp", _) | ("memcmp", _) → true
  | _ → false

```

28.5 Report Optimization and Taint Carriers (TAJ 2009)

Foundational Paper

[Tripp09] (Tripp, Pistoia, Fink, Sridharan, Weisman 2009)

Problem: Raw Taint Analysis Produces Too Many Reports

A large codebase may have thousands of potential taint flows. Developers cannot review them all — they need **actionable** findings.

TAJ Solutions:

- 8.1.5.1 Library Call Point (LCP) Grouping — reduce report volume
- 8.1.5.2 Taint Carrier Detection — find nested/indirect taint

Cross-references:

- Findings feed into Manifest/Latent classification (Section 12.3)
- Taint on object state relates to typestate tracking (Section 7.2.1)

28.5.1 Library Call Point (LCP) Grouping

The Insight: Multiple taint flows often share the *same root cause* — a single API call where tainted data enters the system.

Example:

```
line 100: data = request.get_param("user")    # LCP: taint enters here
line 110: query1 = "SELECT * FROM users WHERE name = '" + data + "'"
line 120: query2 = "SELECT * FROM orders WHERE user = '" + data + "'"
line 130: query3 = "DELETE FROM sessions WHERE user = '" + data + "'"
```

Raw analysis reports 3 SQL injection vulnerabilities. But the **fix** is **one** location: sanitize at line 100.

LCP Grouping:

- Group all findings by the Library Call Point where taint enters
- Report **one** representative finding per LCP group
- Developer fixes **one** location, addresses **all** variants

Benefit:

- 10–100x reduction in report volume
- Each finding is actionable (one fix location)
- Prioritize by LCP with most flows (highest impact fixes)

Library Call Point Grouping

```
module BrrrMachine.Security.LCPGrouping

type library_call_point = {
  api_function : string;           (* e.g., "request.get_param" *)
  call_site : node_id;            (* Location in source code *)
  sink_types : set taint_sink;    (* Types of sinks this LCP reaches *)
}

type lcp_grouped_report = {
  lcp : library_call_point;
  representative_flow : taint_flow; (* Shortest/clearest example *)
  all_flows : list taint_flow;      (* All flows from this LCP *)
  impact_score : nat;               (* Number of sinks affected *)
}

(* Extract LCP from a taint flow - the source is the LCP *)
val extract_lcp : taint_flow → library_call_point
```

```

let extract_lcp flow =
{
  api_function = get_source_api flow.source;
  call_site = flow.source_location;
  sink_types = Set.singleton (get_sink_type flow.sink);
}

(* Create deduplicated report - one entry per LCP *)
val deduplicate_reports : list taint_flow → list lcp_grouped_report
let deduplicate_reports flows =
  let groups = group_by_lcp flows in
  Map.fold (fun lcp group acc →
    (* Pick shortest path as representative - clearest to understand *)
    let representative = List.min_by (fun f → List.length f.path) group in
    (* Compute impact: number of distinct sinks *)
    let all_sinks = List.fold_left (fun s f →
      Set.add (get_sink_type f.sink) s
    ) Set.empty group in
    {
      lcp = { lcp with sink_types = all_sinks };
      representative_flow = representative;
      all_flows = group;
      impact_score = List.length group;
    } :: acc
  ) groups []

(* Sort by impact - highest impact LCPs first *)
val prioritize_reports : list lcp_grouped_report → list lcp_grouped_report
let prioritize_reports reports =
  List.sort (fun a b → compare b.impact_score a.impact_score) reports

```

28.5.2 Taint Carrier Detection

The Problem: Taint stored in **heap objects** may reach sinks indirectly. Standard taint tracking may miss these “carrier” objects.

Example:

```

line 10: user_data = request.get_param("data")    # Taint source
line 20: obj.field = user_data                    # Store taint in carrier
line 30: process(obj)                             # Pass carrier
line 40: def process(o):
line 50:     query(o.field)                        # Taint reaches sink!

```

The taint flows through the **carrier object** `obj`. Without heap graph analysis, we might miss that `obj.field` is tainted when it reaches the sink.

Solution: Build a heap graph from pointer analysis. Track which objects have tainted fields. At sinks, check reachability from tainted fields via heap graph.

Cross-reference: This relates to tpestate tracking (Section 7.2.1, [Bierhoff07]) where object state (including tainted fields) must be tracked across calls.

Taint Carrier Detection

```

module BrrrMachine.Security.TaintCarriers

type taint_carrier = {
  object_var : var_id;          (* Variable holding carrier object *)
  tainted_field : field_id;     (* Field that holds tainted data *)
  store_site : node_id;        (* Where taint was stored *)
  source : taint_source;       (* Original taint source *)
}

type heap_edge =
  | FieldEdge : base:abstract_loc → field:field_id → target:abstract_loc

```

```

    → heap_edge
  | ArrayEdge : base:abstract_loc → target:abstract_loc → heap_edge

type heap_graph = {
  nodes : set abstract_loc;
  edges : set heap_edge;
  tainted_locs : map abstract_loc (set (field_id * taint_source));
}

(* Build heap graph from pointer analysis + taint stores *)
val build_heap_graph : cpg → pts_solution → heap_graph
let build_heap_graph cpg pts =
  let nodes = collect_all_locs pts in
  let edges = collect_heap_edges cpg pts in
  let tainted = find_taint_stores cpg pts in
  { nodes; edges; tainted_locs = tainted }

(* Check if a sink argument can receive tainted data via carrier *)
val check_carrier_taint :
  heap_graph →
  sink_arg:abstract_loc →
  option (taint_carrier * list heap_edge) (* Carrier + path to sink *)

let rec check_carrier_taint hg sink_arg =
  (* Direct check: is sink_arg a tainted location? *)
  match Map.find_opt sink_arg hg.tainted_locs with
  | Some taints when not (Set.is_empty taints) →
    let (field, source) = Set.choose taints in
    Some ({ object_var = loc_to_var sink_arg;
            tainted_field = field;
            store_site = 0; (* TODO: track *)
            source = source }, [])
  | _ →
    (* Transitive check: can we reach a tainted location via heap edges? *)
    bfs_find_taint hg sink_arg

```

Integration: TAJ Report Pipeline

1. Run standard IFDS taint analysis (Section 28.2)
2. Detect taint carriers via heap graph (Section 8.1.5.2)
3. Merge direct + carrier taint flows
4. Group by LCP (Section 8.1.5.1)
5. Prioritize by impact score
6. Feed into Manifest/Latent classification (Section 12.3)

Result: Actionable, prioritized, deduplicated vulnerability reports.

28.6 Lifecycle-Aware Taint Analysis (FlowDroid)

Foundational Paper

[Arzt14] (Arzt, Rasthofer, Fritz, Bodden, Bartel, Klein, Le Traon, Octeau, McDaniel 2014)

Critical Insight: Framework Applications are NOT Standalone Programs

Android apps, Django views, Express handlers, etc. are **plugins** into a framework that invokes callbacks based on system events.

Without lifecycle modeling: Analysis misses many real leaks

With lifecycle modeling: 93% recall vs 25–76% for tools without it

The Challenge:

- Framework invokes app callbacks in arbitrary order
- Callbacks share state via instance fields
- Taint may flow: callback1 → field → callback2 → sink
- Without modeling *all* callbacks, we miss these flows

Example (Android):

```
class LeakyActivity extends Activity {
    String secret;

    void onCreate(Bundle b) {
        secret = getDeviceId(); // Source: IMEI
    }
    void onResume() {
        sendToServer(secret);    // Sink: network
    }
}
```

Standard analysis: Sees only `onCreate` *or* `onResume`, misses the flow.

FlowDroid: Models that `onResume` follows `onCreate`, finds the leak.

Solution: Dummy Main Method

Generate a synthetic entry point that models all possible callback orderings:

```
void dummyMain() {
    Activity a = new LeakyActivity();
    while (true) { // Non-deterministic loop = all orderings
        switch (nondet()) {
            case 0: a.onCreate(null); break;
            case 1: a.onStart(); break;
            case 2: a.onResume(); break;
            case 3: a.onPause(); break;
            case 4: a.onStop(); break;
            case 5: a.onDestroy(); break;
        }
    }
}
```

The loop structure allows analysis to consider all orderings without explicit enumeration (2^n combinations).

28.6.1 Activation Statements for Flow-Sensitive Heap Taint

Problem: Standard IFDS Loses Flow-Sensitivity for Heap-Stored Taint

```
p1.f = source();    // Line 1: taint stored in heap
sink(p2.f);         // Line 2: should NOT report (p1 != p2 yet)
p1 = p2;            // Line 3: now aliased!
sink(p2.f);         // Line 4: SHOULD report (p1 == p2)
```

Without flow-sensitivity: BOTH sinks flagged (false positive at line 2)

With activation statements: Only line 4 flagged (precise)

Solution: Activation Statements

When backward alias analysis finds a potential alias, record *where* it was found. Mark the taint as **inactive** until forward analysis passes that point.

1. Forward finds heap write: `p1.f = tainted`
→ Spawn backward analysis to find aliases of `p1.f`
2. Backward finds: at line 3, `p2` becomes aliased to `p1`
→ Create fact: (`p2.f`, `INACTIVE`, `activation=line3`)
3. Forward propagates inactive taint past line 3
→ Fact becomes: (`p2.f`, `ACTIVE`, `activation=line3`)
4. At sink: only report **active** taints
→ Line 2: `p2.f` is `INACTIVE` → no report
→ Line 4: `p2.f` is `ACTIVE` → **report leak**

The F* formalization below introduces **access paths** — a representation that tracks not just variables, but entire field access chains (e.g., `x.f.g.h`). Access paths enable field-sensitive taint tracking:

- **base**: The root variable
- **fields**: A list of field identifiers forming the access chain
- **array_wildcard**: When true, matches any array index (e.g., `x.f[*]`)

The `max_access_path_length` constant (typically 5) bounds path length to ensure termination. Longer paths are approximated using array wildcards.

Activation-Aware Taint Analysis

```
(* Source: Arzt 2014 (FlowDroid Section 4.3) *)
module BrrrMachine.Security.ActivationTaint

(* Access path = variable + field chain for precise heap tracking *)
type access_path = {
  base : var_id;
  fields : list field_id;    (* x.f.g.h represented as [f; g; h] *)
  array_wildcard : bool;    (* x.f[*] matches any array index *)
}

(* Maximum path length to ensure termination *)
let max_access_path_length : nat = 5

(* Taint state with activation tracking *)
type taint_activation_state =
| TaintActive                (* Can trigger leak at sink *)
| TaintInactive : activation:node_id → (* Waiting to pass activation *)
    taint_activation_state

(* Extended taint fact with activation *)
type activation_taint_fact = {
  path : access_path;
  state : taint_activation_state;
  source : taint_source;
  source_location : node_id;
}

(* At sink: only report if taint is ACTIVE *)
val check_sink_with_activation :
  node:node_id →
  facts:set activation_taint_fact →
  sink:taint_sink →
  list taint_flow

let check_sink_with_activation node facts sink =
  (* CRITICAL: Filter to ACTIVE taints only *)
  let active_facts = Set.filter (fun f →
    match f.state with
    | TaintActive → true
```

```

    | TaintInactive _ → false
  ) facts in
Set.to_list active_facts |> List.map (fun fact →
  { source = fact.source;
    source_location = fact.source_location;
    sink = sink;
    sink_location = node;
    path = [];
    vuln_type = sink_to_vuln sink;
    confidence = 1.0 }
)

```

Bidirectional IFDS Extension

FlowDroid extends standard IFDS with **bidirectional analysis**. The forward phase propagates taint from sources, while the backward phase finds heap aliases to tainted locations. These two directions *spawn* each other:

- **Forward** → finds heap write → **spawns Backward**
- **Backward** → finds alias → **spawns Forward** (with INACTIVE taint)

Note: Not Pure IFDS

This is *not* pure IFDS because may-alias is non-distributive. The backward phase uses on-demand alias analysis. A cleaner formalization would use IDE (the non-distributive extension of IFDS) for the backward phase.

Cross-reference: Section 4.1 for standard IFDS (Reps 1995).

Bidirectional IFDS Extension

```

(* Bidirectional analysis direction *)
type analysis_direction =
  | Forward : analysis_direction
  | Backward : analysis_direction

(* Worklist for bidirectional analysis *)
noeq type bidirectional_worklist = {
  forward : list (int * activation_taint_fact);      (* node * fact *)
  backward : list (int * access_path * int);         (* node * path * activation_stmt *)
}

(* Propagate taint through statement, handling activation *)
val propagate_with_activation :
  cpg:cpg →
  node:node_id →
  fact:activation_taint_fact →
  set activation_taint_fact

let propagate_with_activation cpg node fact =
  (* Check if we're passing the activation point *)
  let fact' = match fact.state with
    | TaintInactive activation when node = activation →
      { fact with state = TaintActive } (* ACTIVATE! *)
    | _ → fact
  in
  (* Standard taint propagation *)
  propagate_taint_fact cpg node fact'

```

The bidirectional worklist maintains separate queues for forward and backward work items. When forward analysis encounters a heap write of tainted data, it spawns backward analysis to find all aliases. When backward analysis discovers an alias, it spawns forward analysis with an inactive taint fact that will be activated when passing the alias point.

28.6.2 Framework Lifecycle Modeling

The FlowDroid lifecycle modeling creates a synthetic “dummy main” that captures all possible orderings of framework callbacks. The following F* code generalizes this beyond Android to support Django, Express, and other frameworks. The key insight is that lifecycle callbacks can be modeled as non-deterministic choices within a loop, ensuring the analysis considers all possible callback orderings without explicitly enumerating 2^n combinations.

Framework Lifecycle Modeling

```
(* Generalized beyond Android to support multiple frameworks. *)
module BrrrMachine.Analysis.Lifecycle

(* Generic component type (framework-agnostic) *)
type component_kind =
| CompActivity      (* Android Activity, Django View *)
| CompService       (* Android Service, background worker *)
| CompReceiver       (* Android BroadcastReceiver, event handler *)
| CompProvider       (* Android ContentProvider, data source *)
| CompMiddleware     (* Express/Django middleware *)
| CompHandler        (* HTTP handler, route handler *)

(* Lifecycle callback specification *)
type lifecycle_callback = {
  component_kind : component_kind;
  method_pattern : string;      (* e.g., "on*", "handle*" *)
  phase : lifecycle_phase;
  can_access_state : bool;      (* Can read/write instance fields *)
}

type lifecycle_phase =
| PhaseInit         (* Constructor, onCreate *)
| PhaseStart        (* onStart, request received *)
| PhaseActive        (* onResume, handling request *)
| PhasePause         (* onPause, request complete *)
| PhaseStop          (* onStop, cleanup *)
| PhaseDestroy       (* onDestroy, teardown *)

(* Generate dummy main from detected components *)
val generate_dummy_main :
  components:list (component_kind * string * list method_id) →
  lifecycle:lifecycle_graph →
  ir_func

let generate_dummy_main components lifecycle =
  (* 1. Instantiate all components *)
  let instantiations = List.map (fun (kind, class_name, _) →
    let var = fresh_var kind in
    SLet var (TRef (TStruct class_name)) (ENew class_name [])
  ) components in
  (* 2. Build non-deterministic callback invocation *)
  let callback_calls = List.concat_map (fun (kind, class_name, methods) →
    List.map (fun method_id →
      SCall None (EMethodCall (EVar (var_for kind)) method_id) []
    ) methods
  ) components in
  (* 3. Wrap in while(true) for all orderings *)
  let body = SWhile (EVal (VBool true)) (SNondet callback_calls) in
  { id = "dummyMain"; name = "dummyMain";
    params = []; return_type = TUnit;
    body = SSeq (SSeq_many instantiations) body;
    effect_sig = effect_any; is_public = true;
    source_lang = "java" }
```

28.6.3 Taint Wrappers for Library Methods

Real-world applications depend heavily on library methods whose source code may be unavailable or too expensive to analyze. FlowDroid's **taint wrappers** provide explicit specifications of how taint propagates through these APIs. The wrapper rules specify:

- **PassThrough**: Taint flows from one parameter to another
- **Kill**: Taint is sanitized/removed at a parameter
- **Source**: The method introduces new taint
- **Sink**: The method is a security-sensitive operation

The F* code below models these wrapper rules. The `param_spec` type allows precise specification of taint flow through the receiver (`this`), arguments, return values, and fields.

Taint Wrappers for Library Methods

```
(* Library methods (whose code is unavailable) need explicit taint
   propagation rules. Wrappers specify how taint flows through APIs. *)
module BrrrMachine.Security.TaintWrapper

type taint_wrapper_rule =
  | WrapperPassThrough : from:param_spec → to:param_spec → taint_wrapper_rule
    (* Taint flows from 'from' to 'to' *)
  | WrapperKill : param:param_spec → taint_wrapper_rule
    (* Taint is removed at this parameter *)
  | WrapperSource : to:param_spec → source:taint_source → taint_wrapper_rule
    (* Method introduces taint *)
  | WrapperSink : from:param_spec → sink:taint_sink → taint_wrapper_rule
    (* Method is a sensitive sink *)

type param_spec =
  | ParamThis (* The 'this' pointer *)
  | ParamArg : index:nat → param_spec (* Argument by index *)
  | ParamReturn (* Return value *)
  | ParamFieldOfThis : field:string → param_spec
  | ParamFieldOfArg : index:nat → field:string → param_spec

type taint_wrapper = {
  method_signature : string;
  rules : list taint_wrapper_rule;
}

(* Standard library wrappers *)
val string_wrappers : list taint_wrapper
let string_wrappers = [
  (* String.concat: if either arg tainted, result tainted *)
  { method_signature = "java.lang.String.concat";
    rules = [WrapperPassThrough (ParamArg 0) ParamReturn;
              WrapperPassThrough ParamThis ParamReturn] };
  (* StringBuilder.append: arg taints this and return *)
  { method_signature = "java.lang.StringBuilder.append";
    rules = [WrapperPassThrough (ParamArg 0) ParamThis;
              WrapperPassThrough ParamThis ParamReturn] };
]

val collection_wrappers : list taint_wrapper
let collection_wrappers = [
  (* List.add: if arg tainted, list becomes tainted *)
  { method_signature = "java.util.List.add";
    rules = [WrapperPassThrough (ParamArg 0) ParamThis] };
  (* List.get: if list tainted, return tainted *)
  { method_signature = "java.util.List.get";
    rules = [WrapperPassThrough ParamThis ParamReturn] };
  (* Map.put: both key and value taint the map *)
]
```

```

{ method_signature = "java.util.Map.put";
  rules = [WrapperPassThrough (ParamArg 0) ParamThis;
           WrapperPassThrough (ParamArg 1) ParamThis] };
1

```

Tension: TAJ Taint Carriers vs FlowDroid Activation

TAJ: Uses heap graph BFS reachability — **fast** but flow-insensitive

FlowDroid: Uses activation statements — **precise** but more expensive

Resolution: Configurable strategy based on analysis budget

- For quick scans: TAJ carrier detection (Section 8.1.5.2)
- For precision: FlowDroid activation statements (Section 8.1.6.1)
- For hybrid: Use TAJ for initial candidates, FlowDroid to verify

Heap Taint Strategy Selection

```

type heap_taint_strategy =
| StrategyTAJCarriers      (* Fast heap graph BFS - Section 8.1.5.2 *)
| StrategyActivation      (* Precise activation statements - Section 8.1.6.1 *)
| StrategyHybrid          (* TAJ for candidates, activation to verify *)

```

28.7 Incremental Taint Analysis

Cross-reference: Section 10.1.3 (DRedL), Section 10.1.4 (Infer Deployment)

Incremental Taint Analysis for IDE/CI Integration:

When code changes, we can incrementally update taint analysis results using the techniques from [Szabo18] (DRedL) and [Distefano19] (Infer).

Strategy:

1. File change detected
 - Adaption marks affected CPG thunks dirty (Section 10.1.1)
2. Re-parse changed file via tree-sitter (fast, < 50ms typically)
3. Convert CPG changes to Datalog tuple changes:
 - New taint sources/sinks → insertions
 - Removed sources/sinks → deletions
 - Modified flow edges → increasing/decreasing replacements
4. Use DRedL (Section 10.1.3) to update taint relations:
 - Taint lattice: UNTAINTED < TAINTED (simple 2-point)
 - Adding taint source → increasing replacement (monotonic)
 - Removing taint source → anti-monotonic, needs re-derivation
5. Only report flows involving **changed** code (diff-time principle)

Expected Performance (based on IncA benchmarks [Szabo18]):

- Median update time: 2–5ms
- Speedup vs from-scratch: 65x–243x
- Target: sub-second for IDE integration

Integration with IFDS: Taint analysis is often formulated as IFDS (Section 4.1). Encode IFDS path edges as Datalog relations:

```

PathEdge(d1, n1, d2, n2) :-
  FlowFunction(n1, n2, d1, d2).
PathEdge(d1, n1, d3, n3) :-
  PathEdge(d1, n1, d2, n2),
  PathEdge(d2, n2, d3, n3).

```

Then use DRedL to incrementally maintain `PathEdge` relations.

Zoncolan Experience [Distefano19]:

Facebook's security taint analyzer for 100M LOC Hack codebase:

- Compositional summary-based taint propagation
- Diff-time deployment in CI pipeline
- Outperforms other security detection methods
- Thousands of security fixes deployed

Incremental Taint via DRedL

```
val incremental_taint_update :  
  prev_solution:ifds_solution →  
  cpq_changes:list cpq_change →  
  (ifds_solution * list taint_flow)  
  
let incremental_taint_update prev cpq_changes =  
  (* Convert CPG changes to Datalog tuple changes *)  
  let datalog_changes = List.concat_map cpq_change_to_tuples cpq_changes in  
  (* Use DRedL to maintain IFDS relations incrementally *)  
  let new_solution = maintain_incrementally  
    ifds_program prev.relations prev.support datalog_changes in  
  (* Extract new taint flows from changed relations *)  
  let new_flows = extract_taint_flows_from_solution new_solution in  
  (new_solution, new_flows)
```

Part IX

Multi-Language Analysis

Tension Resolution: Type-Directed vs Arbitrary Conversion

Matthews & Findler 2007: See Appendix D.10.6 for full analysis.

THIS SECTION assumes type-directed conversion: $\text{type_map} : \text{Type}_1 \rightarrow \text{Type}_2$

M&F 2007 shows: Real FFIs have **ARBITRARY** conventions.

Real-World FFI Conventions:

- C returns -1 for error, 0 for success (zero-for-error)
- Python None \leftrightarrow C NULL (null-for-none)
- Sentinel values for special cases
- Custom serialization for complex types

Resolution:

- Section 12.20 adds `conversion_strategy` type
- Section 9.3 integrates guards with conversion strategies
- `CSTypeDirected`, `CSZeroForError`, `CSNullForNone`, `CSCustom` variants

Guard Generation (Section 9.3) handles polarity FLIP at function arguments!

Theoretical Foundation: The multi-language analysis is grounded in **Institution Theory** [Goguen92]. See **Section 12.25** for the formal foundation including:

- The satisfaction condition (truth invariance under translation)
- Theory colimits (combining language theories)
- Institution morphisms (cross-system theorem proving)
- Constraint institutions (freeness/initiality requirements)
- Duplex institutions (combining decidable and expressive logics)

29. Crossing Language Boundaries

Papers: [Matthews07], [Goguen92]

When code crosses from one language to another, safety invariants may be violated. The brrr-machine detects these risks.

29.1 Matthews' Boundary Semantics

Matthews' Key Insight (2007)

Multi-language programs have **EXPLICIT BOUNDARIES**. At each boundary, values must be converted (marshaled).

The Boundary Term:

$$\mathcal{L}_2^{\mathcal{L}_1}(e) \text{ means: "embed } L_1 \text{ expression } e \text{ into } L_2"$$

Evaluation:

- If e evaluates to v in L_1
- Then $\mathcal{L}_2^{\mathcal{L}_1}(e)$ evaluates to $\text{convert}(v, L_1, L_2)$

Conversion Strategies:

1. **Natural Embedding:** Values with direct equivalents are converted naturally.
 - `int` (Python) \rightarrow `i64` (Rust): natural
 - `str` (Python) \rightarrow `String` (Rust): natural
2. **Lump Embedding:** Values without equivalents are "lumped" as opaque.
 - `object` (Python) \rightarrow `PyObject*` (C): lump
3. **Guarded Embedding:** Runtime checks enforce target language invariants.
 - `Option` (Python None) \rightarrow Non-null (Java): requires check

What Can Go Wrong:

- **Python** → **C**: Python has GC, C doesn't. If Python frees object while C holds pointer → use-after-free
- **JavaScript** → **Rust**: JavaScript has null, Rust has Option. If null passed where non-null expected → crash
- **Go** → **C**: Go has goroutines, C has raw threads. If Go object escapes to C → race condition

29.2 Boundary Risk Analysis

The following F* code defines the core data structures for cross-language boundary analysis. The `axiom` type enumerates the safety guarantees that different languages provide, while `language_config` captures the complete characterization of a language's safety properties. When code crosses from one language to another, the difference in axiom sets determines what properties might be violated at the boundary.

Boundary Analysis — Source: Matthews 2007

```
module BrrrMachine.Boundary

(* Language Axioms --- What each language guarantees *)
type axiom =
| AxMemSafe      (* No use-after-free, double-free, buffer overflow *)
| AxNullSafe     (* No null pointer dereference *)
| AxTypeSafe     (* No type errors at runtime *)
| AxRaceFree     (* No data races *)
| AxLeakFree     (* No resource leaks *)
| AxDetDrop      (* Deterministic resource destruction *)
| AxInitSafe     (* No uninitialized reads *)

type language_config = {
  name : string;
  axioms : set axiom;
  memory_mode : memory_mode;
  type_mode : type_mode;
  null_mode : null_mode;
  eval_mode : eval_mode; (* Evaluation strategy - affects VC soundness *)
}

type memory_mode = ModeGC | ModeRC | ModeOwned | ModeManual
type type_mode = TStatic | TDynamic | TGradual
type null_mode = NNullable | NOptional | NNonNull
```

Evaluation Mode — Critical for Verification Condition Soundness (Vazou 2014)

```
(* CRITICAL INSIGHT: VCs that are SOUND under eager evaluation may be
UNSOUND under lazy evaluation! Under laziness, a binding like:
  let n = diverge 1 in ...
means the VC can include assumptions about n that are NEVER realized
because n may never be evaluated.

The "false" refinement in diverge's output type contaminates the VC:
  false ∧ y = 0 ⇒ v = 0 ⇒ v > 0
This is VALID (contradiction in antecedent) but UNSOUND under laziness!

See Section 2.1.5b for stratified types that restore soundness. *)

type eval_mode =
| EvalStrict     (* Call-by-value: all bindings are values, VCs are sound *)
| EvalLazy       (* Call-by-need: bindings may be thunks, need stratification *)
| EvalHybrid     (* Mixed: strict by default with lazy constructs *)
```

Python Configuration with JARVIS-Style Analysis (Huang 2023)

```
(* Python requires specialized call graph construction via Function Type
Graph (FTG) rather than Qilin-style object-sensitive analysis. *)
```

```

type python_mro_algorithm =
| C3Linearization    (* Standard Python MRO - REQUIRED *)
| SimpleDFS          (* Fallback for pathological hierarchies *)

type python_magic_handling =
| MagicFull          (* Handle __getattr__, __call__, descriptors *)
| MagicPartial       (* Only __init__ and __call__ *)
| MagicNone          (* Ignore magic methods - fast but imprecise *)

type python_decorator_mode =
| DecoratorExpand    (* Expand decorator chains for precise call graph *)
| DecoratorIdentity  (* Treat decorators as identity - fast but imprecise *)

type python_import_resolution =
| ImportFullPackage  (* Full package hierarchy with __init__.py *)
| ImportFlat         (* Flat module resolution *)

type python_type_graph_mode =
| FTGFlowSensitive   (* JARVIS-style with strong updates - RECOMMENDED *)
| FTGFlowInsensitive (* PyCG-style - faster but less precise *)

type python_analysis_mode =
| AnalysisApplicationCentered (* Only reachable from entry points *)
| AnalysisWholeProgramme     (* Analyze everything - expensive *)
| AnalysisDemandDriven       (* Only what's queried *)

type python_options = {
  mro_algorithm : python_mro_algorithm;
  magic_handling : python_magic_handling;
  decorator_mode : python_decorator_mode;
  import_resolution : python_import_resolution;
  type_graph_mode : python_type_graph_mode;
  analysis_mode : python_analysis_mode;
}

let default_python_options : python_options = {
  mro_algorithm = C3Linearization;
  magic_handling = MagicFull;
  decorator_mode = DecoratorExpand;
  import_resolution = ImportFullPackage;
  type_graph_mode = FTGFlowSensitive;
  analysis_mode = AnalysisApplicationCentered;
}

```

Concrete Language Configurations

```

let python_config : language_config = {
  name = "Python";
  axioms = set_of [AxMemSafe; AxLeakFree];
  memory_mode = ModeGC;
  type_mode = TDynamic;
  null_mode = NNullable;
  eval_mode = EvalHybrid; (* Strict by default, generators are lazy *)
}

let rust_config : language_config = {
  name = "Rust";
  axioms = set_of [AxMemSafe; AxNullSafe; AxTypeSafe; AxRaceFree;
                  AxLeakFree; AxDetDrop; AxInitSafe];
  memory_mode = ModeOwned;
  type_mode = TStatic;
  null_mode = NOptional;
  eval_mode = EvalStrict;
}

```

```

}

let c_config : language_config = {
  name = "C";
  axioms = set_of []; (* C guarantees nothing! *)
  memory_mode = ModeManual;
  type_mode = TStatic;
  null_mode = NNullable;
  eval_mode = EvalStrict;
}

let go_config : language_config = {
  name = "Go";
  axioms = set_of [AxMemSafe; AxTypeSafe; AxLeakFree];
  memory_mode = ModeGC;
  type_mode = TStatic;
  null_mode = NNullable;
  eval_mode = EvalStrict;
}

let javascript_config : language_config = {
  name = "JavaScript";
  axioms = set_of [AxMemSafe; AxLeakFree; AxRaceFree]; (* Single-threaded *)
  memory_mode = ModeGC;
  type_mode = TDynamic;
  null_mode = NNullable;
  eval_mode = EvalHybrid;
}

let haskell_config : language_config = {
  name = "Haskell";
  axioms = set_of [AxMemSafe; AxTypeSafe; AxLeakFree];
  memory_mode = ModeGC;
  type_mode = TStatic;
  null_mode = NOptional;
  eval_mode = EvalLazy; (* CRITICAL: Lazy evaluation *)
  (* NOTE: EvalLazy requires stratified type analysis (Div/Wnf/Fin).)
  (* Standard VC translation is UNSOUND. *)
}

```

The following F* code defines the **boundary mechanism** taxonomy, which captures the different ways code can cross language boundaries. Each mechanism type has distinct security implications: FFI calls may expose memory safety issues, IPC/RPC boundaries introduce serialization concerns, and language-specific mechanisms like `ctypes` or `JNI` have their own quirks. The boundary record type pairs the source and target language configurations with the specific mechanism and call site location.

Boundary Definition

```

type boundary_mechanism =
| BoundaryFFI           (* Foreign function interface *)
| BoundaryIPC           (* Inter-process communication *)
| BoundaryRPC           (* Remote procedure call *)
| BoundaryFile          (* File-based communication *)
| BoundarySocket        (* Network socket *)
| BoundarySerialization (* JSON, protobuf, etc. *)
(* Python-C specific FFI mechanisms (PolyCruise, Li 2022) *)
| BoundaryFFI_Ctypes    (* Python ctypes.CDLL *)
| BoundaryFFI_Extension (* Python C extension module *)
| BoundaryFFI_Callback  (* C calling back to Python *)
| BoundaryFFI_CFFI      (* Python CFFI *)
(* Node.js specific *)
| BoundaryFFI_NAPI      (* Node.js N-API for native addons *)
| BoundaryFFI_WASM      (* WebAssembly boundary *)

```

```

(* JNI/JNA *)
| BoundaryFFI_JNI      (* Java Native Interface *)
| BoundaryFFI_JNA      (* Java Native Access *)

type boundary = {
  source_lang : language_config;
  target_lang : language_config;
  mechanism : boundary_mechanism;
  call_site : node_id;
}

```

Risk Calculation

```

(* Axioms at risk = axioms source has but target lacks *)
val boundary_risks : boundary → set axiom
let boundary_risks b =
  Set.diff b.source_lang.axioms b.target_lang.axioms

(* Risk severity *)
type risk_level = RiskCritical | RiskHigh | RiskMedium | RiskLow | RiskNone

let axiom_risk_level (ax : axiom) : risk_level =
  match ax with
  | AxMemSafe → RiskCritical    (* Memory corruption = RCE *)
  | AxTypeSafe → RiskHigh      (* Type confusion = potential RCE *)
  | AxNullSafe → RiskMedium    (* Null deref = crash *)
  | AxRaceFree → RiskHigh      (* Races = unpredictable behavior *)
  | AxLeakFree → RiskLow       (* Leaks = DoS over time *)
  | AxDetDrop → RiskLow        (* Non-determinism = subtle bugs *)
  | AxInitSafe → RiskMedium    (* Uninit = info leak or crash *)

let max_risk_level (risks : set axiom) : risk_level =
  Set.fold (fun ax level →
    max level (axiom_risk_level ax)
  ) risks RiskNone

```

The following F* code implements **type consistency** for gradual typing, a key concept for handling type checking at language boundaries. Unlike traditional subtyping, type consistency is reflexive and symmetric but **NOT transitive**—this non-transitivity is essential for soundness. The `gradual_type` algebraic data type represents types that may contain the dynamic type GTDynamic (written as `?` in the literature), which is consistent with any other type.

Type Consistency for Gradual Typing (Siek & Taha 2006, Garcia 2016)

```

(* CRITICAL: Type consistency is NOT TRANSITIVE!
   This is essential for soundness at language boundaries.

   Example of why transitivity is unsound:
   int ⊢ ?      (consistent)
   ? ⊢ string   (consistent)
   int ⊢ string (INCONSISTENT!)
   If we assumed transitivity, we'd allow int → Any → string coercions. *)

type gradual_type =
| GTGround : ground_type → gradual_type
| GTDynamic : gradual_type (* The ? type - consistent with anything *)
| GTFunc : list gradual_type → gradual_type
| GTRef : gradual_type → gradual_type

(* Type consistency relation - reflexive, symmetric, but NOT transitive *)
val type_consistent : gradual_type → gradual_type → bool
let rec type_consistent t1 t2 =
  match t1, t2 with
  | GTDynamic, _ → true      (* ? ⊢ t for any t *)
  | _, GTDynamic → true      (* t ⊢ ? for any t *)

```

```

| GTGround g1, GTGround g2 → g1 = g2
| GTFunc p1 r1, GTFunc p2 r2 →
  List.length p1 = List.length p2 &&
  List.for_all2 type_consistent p2 p1 &&  (* Contravariant! *)
  type_consistent r1 r2
| GTRef t1', GTRef t2' → type_consistent t1' t2'
| _, _ → false

(* Cast insertion when types are consistent but not equal *)
type cast_result = CastOK of ir_expr | CastError of string

val insert_boundary_cast : gradual_type → gradual_type → ir_expr → cast_result
let insert_boundary_cast src tgt e =
  if src = tgt then CastOK e
  else if type_consistent src tgt then CastOK (ECast e tgt)
  else CastError ("Inconsistent types: cannot cast " ^ show src ^ " to " ^ show tgt)

```

Abstracting Gradual Typing (AGT) — Garcia, Clark & Tanter 2016

```

(* CRITICAL INSIGHT: Gradual types ARE abstract interpretations of static
   type sets. This connects Part IX to Part II (Abstract Interpretation).

   The unknown type ? represents the SET OF ALL STATIC TYPES:
    $\gamma(?) = \{\text{all static types}\}$  -- ? abstracts everything
    $\gamma(\text{Int}) = \{\text{Int}\}$  -- ground types abstract themselves
    $\gamma(? \rightarrow \text{Int}) = \{T \rightarrow \text{Int} \mid \text{any } T\}$  -- partial knowledge

   GALOIS CONNECTION (connects to Section 2.1.2):
   Concrete domain:  $P(\text{StaticType})$  ordered by subset inclusion
   Abstract domain:  $\text{GradualType}$  ordered by precision
    $\alpha$ : Set of static types  $\rightarrow$  minimal gradual type covering set
    $\gamma$ : Gradual type  $\rightarrow$  set of static types it represents

   TYPE CONSISTENCY DERIVED (not stipulated):
    $G1 \sqsubset G2$  iff  $\gamma(G1) \cap \gamma(G2) \neq \text{empty}$ 

   This derivation EXPLAINS why consistency is non-transitive:
    $\gamma(\text{Int}) \cap \gamma(?) = \{\text{Int}\} \neq \text{empty}$  --  $\text{Int} \sqsubset ?$ 
    $\gamma(?) \cap \gamma(\text{String}) = \{\text{String}\} \neq \text{empty}$  --  $? \sqsubset \text{String}$ 
    $\gamma(\text{Int}) \cap \gamma(\text{String}) = \text{empty}$  --  $\text{Int} \not\sqsubset \text{String}$  *)

(* Concretization: gradual type  $\rightarrow$  set of static types it represents *)
val  $\gamma_{\text{gradual}}$  : gradual_type  $\rightarrow$  set static_type
let rec  $\gamma_{\text{gradual}}$  gt = match gt with
| GTDynamic  $\rightarrow$  all_static_types  (* ? represents ALL static types *)
| GTGround g  $\rightarrow$  singleton (to_static g)
| GTFunc params ret  $\rightarrow$ 
  cartesian_func (List.map  $\gamma_{\text{gradual}}$  params) ( $\gamma_{\text{gradual}}$  ret)
| GTRef t  $\rightarrow$  set_map STRef ( $\gamma_{\text{gradual}}$  t)

(* AGT-derived consistency: equivalent to type_consistent above *)
val agt_consistent : gradual_type  $\rightarrow$  gradual_type  $\rightarrow$  bool
let agt_consistent g1 g2 =
  not (Set.is_empty (Set.intersect ( $\gamma_{\text{gradual}}$  g1) ( $\gamma_{\text{gradual}}$  g2)))

(* THEOREM: AGT consistency equals operational consistency (Siek 2006) *)
val agt_equals_siek :
  g1:gradual_type  $\rightarrow$  g2:gradual_type  $\rightarrow$ 
  Lemma (agt_consistent g1 g2  $\Leftrightarrow$  type_consistent g1 g2)

(* PRECISION ORDERING: G1 is more precise than G2 if  $\gamma(G1) \subset \gamma(G2)$  *)
val precision_leq : gradual_type  $\rightarrow$  gradual_type  $\rightarrow$  bool
let precision_leq g1 g2 =
  Set.subset ( $\gamma_{\text{gradual}}$  g1) ( $\gamma_{\text{gradual}}$  g2)

```

```

(* GTDynamic is LEAST precise (bottom in precision order) *)
val dynamic_is_bottom :
  g:gradual_type →
  Lemma (precision_leq GTDynamic g ∨ g = GTDynamic)

```

Gradual Guarantee (Garcia 2016, Theorem 3)

Making types **LESS** precise (more ?) preserves semantics but may add runtime checks. Making types **MORE** precise removes runtime checks.

If $\vdash e : G$ and G' is less precise than G , then $\vdash e : G'$ and:

- If e evaluates to v under G , it evaluates to v under G'
- Under G' , more runtime checks may occur (and may fail)

This connects to the **pay-as-you-go principle** [Siek06]:

- Fully annotated code: no runtime checks, static guarantees
- Partially annotated: checks at ? boundaries
- Unannotated (?): maximum runtime checking

Consistent Subtyping (AGT for Systems with Subtyping)

(For languages with record subtyping (JS objects, Python dicts, TS interfaces), we need CONSISTENT SUBTYPING, not just consistency: *)*

$G1 \prec G2$ iff $\exists T1$ in $\gamma(G1)$, $T2$ in $\gamma(G2)$. $T1 <: T2$

Equivalently (Garcia 2016, Theorem 3):

*$G1 \prec G2$ iff $\exists G$. $(G1 \prec G)$ and $(G <: G2)$
iff $\exists G$. $(G1 <: G)$ and $(G \prec G2)$ *)*

```

val consistent_subtype : gradual_type → gradual_type → bool
let consistent_subtype g1 g2 =
  Set.∃ (γ_gradual g1) (fun t1 →
    Set.∃ (γ_gradual g2) (fun t2 →
      static_subtype t1 t2))

(* Use consistent_subtype for record types at boundaries *)
val boundary_check_record : gradual_type → gradual_type → bool
let boundary_check_record src tgt =
  match src, tgt with
  | GTGround (TRecord _), GTGround (TRecord _) → consistent_subtype src tgt
  | _, _ → type_consistent src tgt

```

In gradual typing systems, when a runtime type check fails, it is essential to provide **precise blame information** that identifies exactly which boundary caused the error. The **evidence** type below tracks *how* type consistency was established during compilation. When two types are consistent through the dynamic type (?), evidence records this fact. At runtime, if a cast fails, the evidence structure can be traversed to produce actionable error messages like “Type mismatch at boundary Python→Rust: expected Int, got String.” Evidence composition (`compose_evidence`) handles chained casts across multiple boundaries.

Evidence for Precise Blame Tracking (Garcia 2016, Section 5)

(Evidence tracks HOW consistency was established during type checking.
When a runtime check fails, evidence pinpoints the BOUNDARY that caused it.*

*This improves upon simple cast insertion by providing actionable error
messages: "Type mismatch at boundary Python→Rust: expected Int, got String" *)*

```

type evidence =
  | EvRefl : gradual_type → evidence          (* G ↦ G *)
  | EvDynL : gradual_type → evidence          (* ? ↦ G *)
  | EvDynR : gradual_type → evidence          (* G ↦ ? *)

```

```

| EvFunc : list evidence → evidence → evidence      (* Function evidence *)
| EvRecord : list (string * evidence) → evidence     (* Record evidence *)

type evidence_result =
| EvOK : evidence → evidence_result
| EvFail : blame:string → evidence_result

(* Evidence composition - may fail at runtime *)
val compose_evidence : evidence → evidence → evidence_result
let rec compose_evidence ev1 ev2 = match ev1, ev2 with
| EvRefl _, ev → EvOK ev
| ev, EvRefl _ → EvOK ev
| EvDynL g1, EvDynR g2 →
    if type_consistent g1 g2 then EvOK (EvRefl (meet_gradual g1 g2))
    else EvFail ("Incompatible at boundary: " ^ show g1 ^ " vs " ^ show g2)
| EvDynR _, EvDynL _ → EvOK (EvDynL GTDynamic)
| EvFunc ps1 r1, EvFunc ps2 r2 →
    (* Contravariant for params, covariant for return *)
    compose_func_evidence ps1 r1 ps2 r2
| _, _ → EvFail "Evidence structure mismatch"

(* Enhanced cast with evidence for blame tracking *)
type cast_with_blame = CastBlameOK of ir_expr * evidence | CastBlameFail of string

val insert_boundary_cast_with_blame :
  source_lang:string → target_lang:string →
  gradual_type → gradual_type → ir_expr → cast_with_blame

```

Occurrence Typing Complements Gradual Typing (Tobin-Hochstadt 2008 + Siek 2006)

```

(* CRITICAL INSIGHT: These two systems solve DIFFERENT problems:
- Gradual typing: Handle ? types at MODULE BOUNDARIES
- Occurrence typing: Refine types WITHIN modules via type tests

TOGETHER they provide a complete solution for dynamic language analysis:
- At boundary: Use type consistency (Siek 2006, derived via AGT)
- Within module: Use occurrence typing refinement (Tobin-Hochstadt 2008)

AGT INTERPRETATION: Occurrence typing NARROWS the concretization set.
Given gradual type G with  $\gamma(G) = \{T1, T2, T3, \dots\}$ , a type test
"if isinstance(x, T1)" narrows  $\gamma$  to  $\{T1\}$  in the true branch. *)

type refined_gradual_type = {
  base_type : gradual_type;          (* The declared/inferred type *)
  refinements : list type_prop;      (* Active refinement propositions *)
  narrowed_type : option gradual_type; (* Refined type if different *)
}

(* Refine a gradual type based on occurrence typing propositions *)
val refine_gradual_type : gradual_type → type_prop → refined_gradual_type

(* At module boundary: forget refinements, use only base type *)
val boundary_type : refined_gradual_type → gradual_type
let boundary_type rgt = rgt.base_type

(* Within module: use refined type if available *)
val effective_type : refined_gradual_type → gradual_type
let effective_type rgt =
  match rgt.narrowed_type with
  | Some t → t
  | None → rgt.base_type

```

The `boundary_issue` algebraic data type enumerates the categories of problems that can arise when values cross language boundaries. Each constructor captures a specific risk class

identified during boundary analysis: null values entering null-safe languages, unclear ownership semantics, type mismatches, unsanitized tainted data propagation, lifetime violations, thread-safety concerns, and serialization incompatibilities. These issues are reported by the boundary analyzer and can trigger guard generation (Section 9.3) or compilation errors.

Boundary Issues — What Can Go Wrong

```
type boundary_issue =
| IssueNullCrossing : loc:node_id → boundary_issue
  (* Nullable value enters null-free language *)
| IssueOwnershipUnclear : loc:node_id → boundary_issue
  (* Ownership not specified at boundary *)
| IssueTypeMismatch : expected:string → actual:string → loc:node_id
  → boundary_issue
  (* Type doesn't match across boundary *)
| IssueTaintedCrossing : source:taint_source → loc:node_id → boundary_issue
  (* Tainted data crosses without sanitization *)
| IssueLifetimeMismatch : loc:node_id → boundary_issue
  (* Reference may outlive its target *)
| IssueThreadSafety : loc:node_id → boundary_issue
  (* Non-thread-safe value crosses to concurrent context *)
| IssueSerializationUnsafe : type_:string → loc:node_id → boundary_issue
  (* Type may not round-trip through serialization *)
```

The `analyze_boundary` function takes a CPG, a boundary specification, and the arguments being passed across the boundary. It iterates through each argument, checking for violations of the axioms that the source language guarantees but the target language does not. The `detect_boundaries` function scans the CPG for FFI and RPC call sites, constructing boundary records for each cross-language call discovered.

Boundary Analysis and Detection

```
val analyze_boundary :
  cpg →
  boundary →
  args:list (node_id * abstract_value) →
  list boundary_issue

let analyze_boundary cpg boundary args =
  let risks = boundary_risks boundary in
  let issues = ref [] in
  List.iter (fun (arg_node, arg_val) →
    (* Null safety risk *)
    if Set.mem AxNullSafe risks then
      if may_be_null cpg arg_node then
        issues := IssueNullCrossing arg_node :: !issues;
    (* Memory safety risk *)
    if Set.mem AxMemSafe risks then
      if is_reference arg_val then
        issues := IssueOwnershipUnclear arg_node :: !issues;
    (* Type safety risk *)
    if Set.mem AxTypeSafe risks then
      let expected = get_expected_type boundary arg_node in
      let actual = get_actual_type cpg arg_node in
      if not (types_compatible expected actual) then
        issues := IssueTypeMismatch expected actual arg_node :: !issues;
    (* Race freedom risk *)
    if Set.mem AxRaceFree risks then
      if is_shared_mutable arg_val then
        issues := IssueThreadSafety arg_node :: !issues;
  ) args;
  !issues

val detect_boundaries : cpg → list boundary
let detect_boundaries cpg =
```

```

fold_nodes cpg (fun boundaries node →
  match node.kind with
  | NCall when is_ffi_call cpg node.id →
    let source = get_source_language cpg node.id in
    let target = get_target_language cpg node.id in
    { source_lang = source;
      target_lang = target;
      mechanism = BoundaryFFI;
      call_site = node.id } :: boundaries
  | NCall when is_rpc_call cpg node.id →
    { source_lang = get_source_language cpg node.id;
      target_lang = infer_target_language cpg node.id;
      mechanism = BoundaryRPC;
      call_site = node.id } :: boundaries
  | _ → boundaries
) []

```

29.3 Realizability Models for Semantic Soundness

Paper: [Patterson22] (Semantic Soundness for Language Interoperability)

Limitation of Syntactic Approach

Matthews’ boundary terms (Section 9.1.1) provide operational semantics but don’t answer: “When is a type conversion **SEMANTICALLY SOUND**?”

Patterson’s critique: “Matthews-Findler-style boundaries give an elegant, abstract model but they don’t reflect reality... understanding of what datatypes should be convertible depends on how sources are **COMPILED** and how data is **REPRESENTED** in the target.”

For **REAL** soundness proofs, we need **SEMANTIC** models.

Realizability Model [Patterson22]:

$V[\tau]$ = set of TARGET values that “behave as” source type τ

Instead of syntactic conversion rules, we:

1. Interpret types from BOTH languages as sets of target terms
2. Define convertibility as GLUE CODE that preserves type membership
3. Prove soundness by showing glue code maps $V[\tau_1]$ into $V[\tau_2]$

Convertibility Judgment:

$\tau_A \sim \tau_B$ means: types are interconvertible with sound glue code.

Requires:

- $\text{glue}_{A \rightarrow B} : \text{target} \rightarrow \text{target}$
- $\text{glue}_{B \rightarrow A} : \text{target} \rightarrow \text{target}$
- $\forall v. v \in V[\tau_A] \Rightarrow \text{glue}_{A \rightarrow B}(v) \in V[\tau_B]$
- $\forall v. v \in V[\tau_B] \Rightarrow \text{glue}_{B \rightarrow A}(v) \in V[\tau_A]$

Example: Python int \leftrightarrow C long

- $V[\text{Python int}] = \text{arbitrary-precision integers}$
- $V[\text{C long}] = 64\text{-bit signed integers}$
- $\text{glue}_{Py \rightarrow C}(n) = \text{if } |n| > 2^{63} \text{ then raise OverflowError else } n$
- NOT identity! Glue code enforces target constraints.

Theorem 29.1 (Shared Memory Identity Constraint (Patterson 2022, Section 4.2)). *To share ref τ_1 with ref τ_2 via IDENTITY (no copy):*

$$\text{glue}_{1 \rightarrow 2} = \text{id} \wedge \text{glue}_{2 \rightarrow 1} = \text{id} \quad \Rightarrow \quad V[\tau_1] = V[\tau_2]$$

Type interpretations must be **IDENTICAL**.

Implications:

- ✓ Rust `&mut i32` ↔ C `int*`: Both have same target representation. Identity glue is safe.
- × Python `list` ↔ C `array`: Different representations. MUST copy or use opaque handle.
- × Java `String` ↔ Rust `String`: UTF-16 vs UTF-8. MUST convert encoding + copy.

Realizability Analysis Extension

```
(* Target representation equivalence *)
type target_repr =
  | TRWord of nat                (* n-bit word *)
  | TRPointer of target_repr    (* pointer to *)
  | TRStruct of list target_repr
  | TROpaque                    (* Cannot compare *)

let get_target_repr (lang : language_config) (ty : ir_type) : target_repr =
  match lang.name, ty with
  | "Rust", TInt 32 → TRWord 32
  | "C", TInt 32 → TRWord 32
  | "Python", TInt _ → TROpaque (* Arbitrary precision *)
  | _, TPointer inner → TRPointer (get_target_repr lang inner)
  | _ → TROpaque

(* Check if shared mutable reference is safe *)
let check_shared_ref_safe (b : boundary) (ty1 ty2 : ir_type) : option boundary_issue =
  let repr1 = get_target_repr b.source_lang ty1 in
  let repr2 = get_target_repr b.target_lang ty2 in
  if repr1 ≠ repr2 then
    Some (IssueSharedRefUnsafe ty1 ty2 b.call_site)
  else
    None

type boundary_issue =
  (* ... existing issues ... *)
  | IssueSharedRefUnsafe : ty1:ir_type → ty2:ir_type → site:node_id
    → boundary_issue (* NEW *)
```

GC-Linear Interoperability Asymmetry:

Direction matters:

- **Linear** → **GC**: SAFE to convert directly. Linear guarantee = no aliases exist. `gcmov` instruction transfers ownership to GC. No copy needed.
- **GC** → **Linear**: REQUIRES COPY. GC reference may have unknown aliases. Cannot guarantee uniqueness. Must copy data to establish linear ownership.

This asymmetry affects:

- Rust → Python: can pass `Box<T>` directly (linear → GC)
- Python → Rust: must copy `PyObject` to owned `T` (GC → linear)

See Section 12.7 for full realizability theorems.

29.4 Cross-Language Dynamic Information Flow Analysis (PolyCruise)

Paper: [Li22] — “PolyCruise: A Cross-Language Dynamic Information Flow Analysis”

Critical Insight

Single-language analyzers miss vulnerabilities at language boundaries. Even if individual language units are secure, the combined system may not be.

Example (Python-C): Buffer overflow in NumPy

- Python: `data = input()` # Unconstrained user input
- C (via FFI): `memcpy(buf, data)` # No bounds check → buffer overflow

Neither Python nor C analysis alone sees the full flow!
PolyCruise found **14 unknown cross-language vulnerabilities** (8 CVEs).

The Challenge:

- Static semantic unification across languages is EXTREMELY HARD
- Different memory models, type systems, evaluation strategies
- Pure static cross-language analysis doesn't scale

PolyCruise Solution: Hybrid Static + Dynamic

- **LIGHT STATIC:** Determine WHAT to instrument (cheap)
- **HEAVY DYNAMIC:** Track ACTUAL flows at runtime (precise)

Key insight: While SEMANTIC analysis across languages is hard, SYNTACTIC analysis (def/use) can be unified.

29.4.1 Language-Independent Symbolic Representation (LISR)

PolyCruise's key innovation is the **Language-Independent Symbolic Representation (LISR)**, which provides a unified intermediate form for def/use analysis across different programming languages. While full semantic unification across languages is extremely difficult due to differences in memory models and type systems, **syntactic** def/use analysis can be unified. The following F* code defines LISR's core types: `symbolic_name` abstracts language-specific identifiers into a canonical form, and `lizr_stmt` represents statements in a language-agnostic way. The `LizrBoundary` constructor explicitly marks cross-language calls.

Language-Independent Symbolic Representation (LISR) — Li 2022

```
module BrrrMachine.CrossLang.LISR

(* Symbolic names abstract away language-specific identifiers *)
type symbolic_name =
  | SymLocal : func_id:string → name:string → symbolic_name
  | SymGlobal : module_id:string → name:string → symbolic_name
  | SymParam : func_id:string → index:nat → symbolic_name
  | SymReturn : func_id:string → symbolic_name
  | SymField : base:symbolic_name → field:string → symbolic_name

(* LISR statements - language-agnostic representation *)
type lizr_stmt =
  | LizrAssign : lhs:symbolic_name → rhs:lizr_expr → lizr_stmt
  | LizrCall : ret:option symbolic_name → callee:symbolic_name →
    args:list lizr_expr → lizr_stmt
  | LizrOutput : sink_kind:taint_sink → arg:lizr_expr → lizr_stmt
  | LizrInput : source_kind:taint_source → target:symbolic_name → lizr_stmt
  | LizrBranch : cond:lizr_expr → lizr_stmt
  | LizrBoundary : direction:boundary_dir → mechanism:boundary_mechanism →
    inner:lizr_stmt → lizr_stmt (* Cross-language call marker *)

type lizr_expr =
  | LExprSym : symbolic_name → lizr_expr
  | LExprConst : value → lizr_expr
  | LExprBinop : op:binop → lhs:lizr_expr → rhs:lizr_expr → lizr_expr
  | LExprField : base:lizr_expr → field:string → lizr_expr
  | LExprIndex : base:lizr_expr → index:lizr_expr → lizr_expr

type boundary_dir = BoundaryOut | BoundaryIn (* Calling out / returning in *)
```

Def/Use Extraction from LISR

```
(* This is the key to cross-language analysis: language-agnostic def/use. *)

val def_set : lizr_stmt → set symbolic_name
let def_set stmt = match stmt with
```

```

| LIsrAssign lhs _ → Set.singleton lhs
| LIsrCall (Some ret) _ _ → Set.singleton ret
| LIsrInput _ target → Set.singleton target
| LIsrBoundary _ _ inner → def_set inner
| _ → Set.empty

let rec use_expr_set expr = match expr with
| LExprSym sym → Set.singleton sym
| LExprConst _ → Set.empty
| LExprBinop _ l r → Set.union (use_expr_set l) (use_expr_set r)
| LExprField base _ → use_expr_set base
| LExprIndex base idx → Set.union (use_expr_set base) (use_expr_set idx)

val use_set : lIsr_stmt → set symbolic_name
let use_set stmt = match stmt with
| LIsrAssign _ rhs → use_expr_set rhs
| LIsrCall _ _ args → Set.unions (List.map use_expr_set args)
| LIsrOutput _ arg → use_expr_set arg
| LIsrBranch cond → use_expr_set cond
| LIsrBoundary _ _ inner → use_set inner
| _ → Set.empty

(* Convert language-specific IR to LISR *)
val to_lIsr : language_config → ir_func → list lIsr_stmt

```

29.4.2 Symbolic Dependence Analysis (SDA)

Symbolic Dependence Analysis (SDA) computes which program statements are potentially dependent on security-relevant criteria (taint sources and sinks). This determines the **instrumentation scope** for dynamic analysis—only statements in the dependence set need runtime instrumentation. The key insight is to include **both** true dependencies (def flows to use) **AND** anti-dependencies (use followed by def of same symbol), which ensures soundness without requiring expensive pointer analysis. The following F* code defines the **symbolically_dependent** relation and the **compute_sym_dep_set** function that computes the transitive closure of symbolic dependencies.

Symbolic Dependence Analysis (SDA) — Li 2022

```

(* SDA computes which statements are symbolically dependent on criteria
   (taint sources/sinks). This determines the INSTRUMENTATION SCOPE for
   dynamic analysis - we only instrument statements in the dependence set.

   KEY INSIGHT: Include BOTH true dependencies AND anti-dependencies.
   This ensures soundness without expensive pointer analysis. *)

module BrrrMachine.CrossLang.SDA

(* Symbolic dependence relation *)
val symbolically_dependent : lIsr_stmt → lIsr_stmt → bool
let symbolically_dependent si sj =
  let di = def_set si in
  let ui = use_set si in
  let dj = def_set sj in
  let uj = use_set sj in
  (* True/flow dependence: D(Si) intersect U(Sj) != empty *)
  not (Set.is_empty (Set.inter di uj)) ||
  (* Anti-dependence: U(Si) intersect D(Sj) != empty - ensures soundness without aliasing *)
  not (Set.is_empty (Set.inter ui dj))

(* Compute transitive symbolic dependence set from criteria *)
val compute_sym_dep_set :
  program:list lIsr_stmt →
  criteria:set symbolic_name → (* Source/sink symbols *)

```

```

set nat   (* Statement indices in symbolic dependence set *)

(* Symbolic dependence summary for interprocedural analysis *)
type sds_summary = {
  func_id : string;
  return_depends_on_params : list bool;  (* Per-parameter: does return depend on it? *)
  params_depend_on_criteria : list bool;  (* Per-parameter: reachable from criteria? *)
}

```

29.4.3 Dynamic Information Flow Graph (DIFG)

The **Dynamic Information Flow Graph (DIFG)** is constructed at runtime from instrumented execution events. Unlike static data flow graphs, the DIFG captures **actual** flows that occur during program execution, including those that cross language boundaries via FFI. The following F* code defines the runtime event types (`execution_event`), the DIFG node types (`difg_node`), and the edge types (`difg_edge`). The `is_cross_language` field on edges explicitly tracks whether a flow crossed a language boundary, enabling detection of cross-language vulnerabilities.

Dynamic Information Flow Graph (DIFG) — Li 2022

```

(* DIFG is built at RUNTIME from instrumented execution events.
   It captures ACTUAL flows, including those crossing language boundaries. *)

module BrrrMachine.CrossLang.DIFG

(* Runtime execution events (from instrumentation) *)
type execution_event =
  | EventDef : stmt_id:nat → sym:symbolic_name → value:runtime_value →
    lang:language_config → execution_event
  | EventUse : stmt_id:nat → sym:symbolic_name → value:runtime_value →
    lang:language_config → execution_event
  | EventBoundaryCross : direction:boundary_dir → mechanism:boundary_mechanism →
    args:list (symbolic_name * runtime_value) → execution_event
  | EventCall : caller_lang:language_config → callee_lang:language_config →
    callee:symbolic_name → execution_event
  | EventReturn : func:symbolic_name → ret_val:option runtime_value → execution_event

(* DIFG node types *)
type difg_node =
  | DifgSource : source:taint_source → stmt_id:nat → lang:language_config → difg_node
  | DifgSink : sink:taint_sink → stmt_id:nat → lang:language_config → difg_node
  | DifgIntermediate : stmt_id:nat → sym:symbolic_name → lang:language_config → difg_node
  | DifgBoundary : boundary:boundary → difg_node

(* DIFG edge types *)
type difg_edge = {
  from_node : difg_node;
  to_node : difg_node;
  flow_type : difg_flow_type;
  is_cross_language : bool;
}

type difg_flow_type =
  | FlowDirect      (* Direct assignment *)
  | FlowParameter   (* Function parameter passing *)
  | FlowReturn      (* Function return value *)
  | FlowFFI         (* Cross-language FFI boundary *)
  | FlowCallback    (* C calling back to Python *)

(* Extract vulnerability witnesses from completed DIFG *)
type cross_lang_vulnerability = {
  source : taint_source;

```

```

source_lang : language_config;
sink : taint_sink;
sink_lang : language_config;
path : list difg_node;
boundary_crossings : list boundary;
}

val find_cross_lang_flows : difg_state → list cross_lang_vulnerability

```

29.4.4 Hybrid Cross-Language Taint Analysis

Hybrid Static-Dynamic Cross-Language Analysis — Li 2022

```

(* KEY INSIGHT: Use static analysis to determine WHAT to instrument,
    use dynamic analysis to track ACTUAL flows.)

This avoids:
- Scalability issues of pure dynamic (ORBS)
- Semantic disparity issues of pure static cross-language analysis

PERFORMANCE (from paper):
- Static phase: <3 seconds for 220 KSLLOC, <3 minutes for 6,419 KSLLOC
- Runtime overhead: 2.71x - 11.96x slowdown
- Precision: 93.5%, Recall: 100% *)

module BrrrMachine.CrossLang.HybridAnalysis

(* Multi-language program representation *)
type language_unit = {
  language : language_config;
  source_file : string;
  lisr : list lisr_stmt;
  entry_points : list symbolic_name;
}

type multilang_program = {
  units : list language_unit;
  boundaries : list boundary;
  main_entry : symbolic_name;
}

(* Instrumentation point - what to instrument in each language *)
type instrumentation_point = {
  stmt_id : nat;
  kind : instrumentation_kind;
  language : language_config;
}

type instrumentation_kind =
| InstrDef : symbolic_name → instrumentation_kind
| InstrUse : symbolic_name → instrumentation_kind
| InstrCall : callee:symbolic_name → instrumentation_kind
| InstrBoundary : boundary:boundary → instrumentation_kind

(* PHASE 1: Static - compute instrumentation scope via SDA *)
val compute_instrumentation_points :
  program:multilang_program →
  sources:set taint_source →
  sinks:set taint_sink →
  list instrumentation_point

(* PHASE 2: Dynamic - execute with instrumentation, build DIFG *)
val execute_with_instrumentation :
  program:multilang_program →

```

```

instr_points:list instrumentation_point →
inputs:list runtime_value →
difg_state

(* Combined analysis *)
val cross_language_taint_analysis :
  program:multilang_program →
  sources:set taint_source →
  sinks:set taint_sink →
  inputs:list runtime_value →
  list cross_lang_vulnerability

```

Integration with Synthesis

PolyCruise complements the primarily **STATIC** boundary analysis in 9.1.1–9.1.3: **Static (Sections 9.1.1–9.1.3)**:

- Boundary detection and risk classification [Matthews07]
- Type consistency checking [Siek06]
- Realizability models [Patterson22]
- FFI contract verification (VeriFFI, Section 9.4)

Dynamic (PolyCruise, Section 9.1.4):

- LISR for unified cross-language def/use
- SDA for instrumentation guidance
- DIFG for precise runtime flow tracking
- Actual vulnerability witness generation

Recommended Workflow:

1. Static boundary analysis (9.1.1–9.1.3) for risk assessment
2. PolyCruise SDA (9.1.4.2) to scope instrumentation
3. Dynamic DIFG (9.1.4.3) for specific test inputs
4. Feed results into taint classification (Section 12.3)

30. Hole Tagging for Evaluation Contexts

Source: [Matthews07] — “Operational Semantics for Multi-Language Programs”

30.1 The Language Bleeding Problem

Problem: Without hole tagging, reduction rules can “bleed” across languages.

Example (ML expression with embedded Scheme):

$$\text{ML}[1 + \text{MS}(\text{SM}[\text{Scheme-expr}])]$$

Without Tagging: If we reduce “ $1 + _$ ” first, the Scheme-expr is evaluated by ML rules! This violates language semantics.

Solution: Tag holes with their expected language.

30.2 Tagged Evaluation Contexts

Evaluation Hole with Language Tag

```

type eval_hole =
  | HoleML : eval_hole
  | HoleScheme : eval_hole
  | HolePython : eval_hole
  | HoleRust : eval_hole
  | HoleBoundary : source:lang_id → target:lang_id → eval_hole

(* Evaluation context with language annotations *)

```

```

type eval_context =
  | EHole : hole:eval_hole → eval_context
  | EBinOpL : op:binop → ctx:eval_context → rhs:expr → eval_context
  | EBinOpR : op:binop → lhs:value → ctx:eval_context → eval_context
  | EApp : ctx:eval_context → arg:expr → eval_context
  | EBoundary : direction:boundary_dir → ty:ir_type → ctx:eval_context
    → eval_context

(* Fill hole with expression *)
val fill : eval_context → expr → expr
let rec fill ctx e = match ctx with
  | EHole _ → e
  | EBinOpL op inner rhs → EBinOp op (fill inner e) rhs
  | EBinOpR op lhs inner → EBinOp op lhs (fill inner e)
  | EApp inner arg → EApp (fill inner e) arg
  | EBoundary dir ty inner → EBoundary dir ty (fill inner e)

```

30.3 Language-Respecting Reduction

Reduction Rule Constraint: A reduction rule for language L can only fire when the redex is at a hole tagged with L .

Formally: If $e \rightarrow e'$ by L -rule, then in context $C[e]$:

- C has hole tagged L
- All intervening boundaries are properly crossed

Implementation:

```

step(lang, ctx, expr) =
  let hole_lang = tag_of_hole(ctx) in
  if hole_lang = lang then
    apply_rules(lang, expr)
  else
    (* Cannot reduce here - wrong language context *)
    Stuck

```

30.4 Boundary Crossing Updates Tags

Boundary Crossing Tag Updates

```

(* When crossing a boundary, update the hole tag *)
val cross_boundary : eval_context → boundary_dir → lang_id → eval_context
let cross_boundary ctx dir target_lang =
  (* The inner context now expects target_lang *)
  update_inner_hole ctx (HoleBoundary ... target_lang)

(* Example: ML calling Python *)
let ml_python_example =
  EApp (EHole HoleML) (* ML context *)
    (EBoundary ToPython int_ty (* Boundary *)
      (EHole HolePython)) (* Now Python context *)

```

31. Guard Generation Algorithm

Source: [Matthews07] Section 3.3

31.1 Guard Polarity

Key Insight: Guards have **POLARITY** that determines checking direction.

Positive polarity: Value entering STRICTER type system

- Dynamic \rightarrow Static (e.g., Python \rightarrow Rust)

- Must CHECK at runtime (source doesn't guarantee)
 - Negative polarity:** Value from STRICTER type system
- Static \rightarrow Dynamic (e.g., Rust \rightarrow Python)
- Can SKIP check (source guarantees type)
 - Critical:** Polarity **FLIPS** at function argument position! (Contravariance in domain)

31.2 Polarity Determination

Polarity Determination

```
(* Determine initial polarity based on language pair *)
val initial_polarity : lang_id  $\rightarrow$  lang_id  $\rightarrow$  guard_polarity
let initial_polarity source target =
  let source_strict = type_strictness source in
  let target_strict = type_strictness target in
  if target_strict > source_strict then Positive
  else Negative

(* Flip polarity (used at function arguments) *)
val flip : guard_polarity  $\rightarrow$  guard_polarity
let flip Positive = Negative
let flip Negative = Positive
```

31.3 Guard Generation

Guard Generation Algorithm

```
(* Generate guard for type at given polarity *)
val generate_guard : ir_type  $\rightarrow$  guard_polarity  $\rightarrow$  guard
let rec generate_guard ty pol = match ty, pol with

  (* Base types *)
  | TInt, Positive  $\rightarrow$  GCheckNumber
  | TInt, Negative  $\rightarrow$  GNoCheck
  | TBool, Positive  $\rightarrow$  GCheckBool
  | TBool, Negative  $\rightarrow$  GNoCheck
  | TString, Positive  $\rightarrow$  GCheckString
  | TString, Negative  $\rightarrow$  GNoCheck

  (* Nullable / Option types *)
  | TOption inner, Positive  $\rightarrow$ 
    GCompose GCheckNotNull (generate_guard inner Positive)
  | TOption inner, Negative  $\rightarrow$  GNoCheck

  (* Function types - FLIP polarity for argument! *)
  | TArrow arg_ty ret_ty, pol  $\rightarrow$ 
    GFunctionWrap
      (generate_guard arg_ty (flip pol)) (* CONTRAVARIANT *)
      (generate_guard ret_ty pol) (* COVARIANT *)

  (* Product types *)
  | TProd t1 t2, pol  $\rightarrow$ 
    GCompose (generate_guard t1 pol) (generate_guard t2 pol)

  (* Reference types need type check *)
  | TRef inner, Positive  $\rightarrow$  GCheckType (TRef inner)
  | TRef inner, Negative  $\rightarrow$  GNoCheck

  (* Polymorphic types *)
  | TForall _ _, Positive  $\rightarrow$  GDynamic (* Must check dynamically *)
  | TForall _ _, Negative  $\rightarrow$  GNoCheck

  (* Any/Dynamic type *)
```

31.4 Function Wrapping Example

Example: Python function → Rust callback

- Python function type: (Any) → Any
- Rust callback type: (i32) → String
 - At boundary (Python → Rust), polarity = Positive
 - Generate guard for (i32) → String at Positive:
- Argument i32: flip(Positive) = Negative → GNoCheck (Rust provides i32, Python receives)
- Return String: Positive → GCheckString (Python returns, must verify is string)

Wrapped Function:

```
fn wrapped(arg: i32) -> String {
  let py_result = python_fn(arg); // No check on arg
  GCheckString(py_result)         // Check return
}
```

31.5 Integration with Boundary Analysis

Boundary Crossing with Guards

```
(* Complete boundary crossing with guards *)
val cross_with_guards :
  source:lang_id → target:lang_id →
  ty:ir_type → value:value →
  brrr_result value

let cross_with_guards source target ty v =
  let pol = initial_polarity source target in
  let guard = generate_guard ty pol in
  match apply_guard guard v with
  | GPass v' → Ok v'
  | GFail expected actual →
    Error (BoundaryTypeError {
      at_boundary = (source, target);
      expected_type = expected;
      actual_value = actual;
    })

(* Static analysis: verify guards sufficient *)
val verify_boundary_safety :
  cpg → boundary_node → list boundary_issue

let verify_boundary_safety cpg bnd =
  let guard = generate_guard bnd.ty (initial_polarity bnd.source bnd.target) in
  let incoming_types = analyze_incoming_types cpg bnd in
  List.filter_map (fun ty →
    if guard_covers ty guard then None
    else Some (IssueGuardInsufficient bnd ty guard)
  ) incoming_types
```

31.6 Guards with Type Refinement Propositions

Source: [TobinHochstadt08] + [Matthews07]

Guards Produce Type Propositions

Key Insight: Guards are not just runtime checks — they also establish **TYPE KNOWLEDGE** for subsequent code.

When a guard CHECK PASSES:

- We know the value HAS the checked type
- This is a VISIBLE PREDICATE [TobinHochstadt08]
- Subsequent code can use this type information

When a guard CHECK FAILS:

- Exception/error raised — unreachable code
- But in analysis, we can refine with NEGATIVE proposition

Polarity Determines Which Branch Gets Which Refinement:

- **Positive guard** (checking incoming value): Pass branch: value HAS type τ ; Fail branch: unreachable
- **Negative guard** (checking outgoing value): Pass branch: value HAS type τ (trivially); Fail branch: BUG in source

Guards with Refinement Information

```
(* Guard result includes type proposition for refinement *)
type guard_with_refinement = {
  guard : guard;
  positive_prop : type_prop;  (* Proposition when guard passes *)
  negative_prop : type_prop;  (* Proposition when guard fails *)
  target_var : option string; (* Variable being checked, if known *)
}

(* Generate guard with associated type propositions *)
val generate_guard_with_prop :
  ir_type → guard_polarity → option string → guard_with_refinement

let generate_guard_with_prop ty pol var_opt =
  let guard = generate_guard ty pol in
  let make_prop = match var_opt with
    | Some v → (fun t → PropHasType v t)
    | None → (fun _ → PropTrue)
  in
  match pol with
  | Positive →
    (* Checking incoming value - pass means has type *)
    { guard;
      positive_prop = make_prop ty;
      negative_prop = PropFalse;  (* Unreachable - error raised *)
      target_var = var_opt }
  | Negative →
    (* Checking outgoing value - should always pass *)
    { guard;
      positive_prop = PropTrue;  (* Source guarantees type *)
      negative_prop = PropFalse; (* Bug if reached *)
      target_var = var_opt }

(* Apply guard and get refined type environment *)
type guard_application_result =
  | GuardPassed of {
    value : ir_value;
    env_refinement : prop_type_env → prop_type_env;
  }
  | GuardFailed of {
    expected : ir_type;
    actual : ir_value;
  }
```

31.7 Guard Composition Optimization

Optimization: Cancel redundant guards at boundary chains.

If value crosses: $A \rightarrow B \rightarrow C$, then $\text{Guard}(A \rightarrow B)$ followed by $\text{Guard}(B \rightarrow C)$ can often simplify to: $\text{Guard}(A \rightarrow C)$.

Rules:

- Positive \circ Positive = Positive (still need check)
- Negative \circ Negative = Positive (double negative = positive)
- Positive \circ Negative = Negative
- Negative \circ Positive = Negative

Optimization:

- $\text{GNoCheck} \circ G = G$
- $G \circ \text{GNoCheck} = G$
- $\text{GCheckT} \circ \text{GCheckT} = \text{GCheckT}$ (idempotent)

32. FFI Contracts

Source: VeriFFI [Wang25] — “A Verified Foreign Function Interface between Coq and C”

FFI Contract Framework

FFI boundaries require **EXPLICIT contracts** specifying:

- Input type representation requirements
- Output type representation guarantees
- Memory ownership transfer
- Side effect bounds

Without Contracts: FFI calls are opaque black boxes.

With Contracts: FFI calls can be verified for type safety.

Key Insight: Contracts bridge the semantic gap between languages by specifying both **WHAT** the function does and **HOW** values are represented.

Cross-References:

- Section 7.5: Representation predicates used in contracts
- Section 9.1.2: Contracts formalize boundary risk mitigation
- Section 9.3: Guards generated from contract type requirements
- Section 12.7: Realizability theorems justify contract soundness

32.1 Contract Structure

FFI contracts formalize the requirements for safely calling across language boundaries. The following F* code defines the `ffi_contract` type which captures:

- **Type mappings** between source and foreign types
- **Preconditions** that the caller must establish (representation predicates, ownership)
- **Postconditions** that the callee guarantees (memory effects, error handling)
- **Ownership transfer** semantics for resource management across boundaries

The `type_mapping` type describes how values are transformed when crossing the boundary: directly (`TMDirect`), via conversion (`TMConverted`), or as opaque references (`TMOpaque`).

FFI Contract Structure — VeriFFI (Wang 2025)

```
module BrrrMachine.FFIContract

(* Type Mappings Across Boundaries *)
type type_mapping =
| TMDirect
    (* Same representation in both languages.
       Example: i32 (Rust) <-> int (C) *)
| TMBoxed
    (* Heap-allocated wrapper around the value.
```

```

    Example: Python int  $\leftrightarrow$  boxed arbitrary-precision integer *)
| TMConverted of {
    convert_fn : ir_value  $\rightarrow$  foreign_value;
    unconvert_fn : foreign_value  $\rightarrow$  option ir_value;
}
    (* Explicit conversion required.
    Example: Rust String (UTF-8)  $\leftrightarrow$  Java String (UTF-16) *)
| TMOpaque
    (* No access across boundary - treat as opaque handle.
    Example: Python object  $\leftrightarrow$  void* in C *)

(* FFI Contract Type *)
type ffi_contract = {
    foreign_function : string;
    source_language : language_id;
    target_language : language_id;
    param_types : list (ir_type * foreign_type * type_mapping);
    return_type : (ir_type * foreign_type * type_mapping);
    precondition : ffi_precondition;
    postcondition : ffi_postcondition;
    ownership_transfer : ownership_spec;
    effect_bounds : effect_row;
}

```

Precondition Structure

```

type rep_requirement =
| RepRequired : t:Type  $\rightarrow$  rep_predicate t  $\rightarrow$  rep_requirement
| RepNonNull : rep_requirement
| RepInitialized : rep_requirement
| RepAligned : alignment:nat  $\rightarrow$  rep_requirement
| RepSized : min_size:nat  $\rightarrow$  rep_requirement

type ffi_precondition = {
    rep_requirements : list (var_id * rep_requirement);
    ownership_requirements : list (var_id * access_permission);
    value_constraints : list (var_id * ir_expr);
    thread_requirements : option thread_safety_requirement;
}

type thread_safety_requirement =
| TSRSingleThreaded    (* Must be called from single thread *)
| TSRMainThread        (* Must be called from main thread *)
| TSRWithLock of lock_id (* Must hold specified lock *)
| TSRAnyThread         (* Safe to call from any thread *)

```

Postcondition Structure

```

type memory_effect_spec =
| MEUnchanged          (* Memory not modified *)
| MEModifiedOnly of set address (* Only specified addresses modified *)
| MEMayAllocate        (* May allocate new memory *)
| MEMayFree of set address (* May free specified addresses *)
| MEArbitrary          (* No guarantees (unsafe) *)

type ownership_transfer =
| OTRetained           (* Ownership unchanged *)
| OTTransferred        (* Ownership transferred to callee *)
| OTBorrowed of lifetime_bound (* Temporarily borrowed *)
| OTShared             (* Shared access granted *)

type ffi_postcondition = {
    return_rep : rep_requirement;
    memory_effects : memory_effect_spec;
    ownership_effects : list (var_id * ownership_transfer);
    error_conditions : list (ir_expr * error_behavior);
}

```

```

    may_trigger_gc : bool;
}

type error_behavior =
| EBEException of ir_type      (* Throws exception of type *)
| EBReturnCode of ir_value     (* Returns error code *)
| EBSetErrno      (* Sets errno *)
| EBAbort         (* Aborts process *)
| EBUndefined     (* Undefined behavior *)

```

32.2 Contract Generation

Automatic Contract Generation — VeriFFI Section 5

```

(* For common patterns, we can generate contracts automatically
   from type signatures and calling conventions. *)

val generate_ffi_contract :
  source_type : ir_type →
  target_type : foreign_type →
  calling_convention : calling_conv →
  ffi_contract

val infer_type_mapping : ir_type → foreign_type → type_mapping
let infer_type_mapping src tgt =
  match src, tgt with
  | TInt I32 true, ForeignInt32 → TMDirect
  | TInt I64 true, ForeignInt64 → TMDirect
  | TFloat F32, ForeignFloat → TMDirect
  | TFloat F64, ForeignDouble → TMDirect
  | TBool, ForeignBool → TMDirect
  | TPtr inner_src, ForeignPtr inner_tgt →
    let inner_mapping = infer_type_mapping inner_src inner_tgt in
    if inner_mapping = TMDirect then TMDirect
    else TMOpaque
  | TString, ForeignCString →
    TMConverted {
      convert_fn = string_to_cstring;
      unconvert_fn = cstring_to_string;
    }
  | TStruct _ _, _ → TMOpaque
  | TVariant _ _, _ → TMOpaque
  | _, _ → TMOpaque

type calling_conv =
| CCDefault      (* Default C calling convention *)
| CCStdcall      (* Windows stdcall *)
| CCFastcall     (* Fast register-based calling *)
| CCRust         (* Rust ABI *)
| CCOwned        (* Ownership transferred *)
| CCBorrowed     (* Immutable borrow *)
| CCMutBorrowed  (* Mutable borrow *)

```

Annotation-Based Contract Refinement

```

type ffi_annotation =
| AnnNonnull of var_id      (* Parameter cannot be null *)
| AnnNullable of var_id     (* Parameter may be null *)
| AnnOwned of var_id        (* Ownership transferred *)
| AnnBorrowed of var_id     (* Borrowed reference *)
| AnnOut of var_id          (* Output parameter *)
| AnnInOut of var_id        (* Input/output parameter *)
| AnnArraySize of var_id * nat (* Array with fixed size *)
| AnnBufferSize of var_id * var_id (* Buffer sized by another param *)
| AnnMayAllocate            (* May allocate memory *)

```

```

| AnnMayFree           (* May free memory *)
| AnnPure              (* No side effects *)
| AnnThreadSafe        (* Safe for concurrent calls *)

val refine_contract_with_annotations :
  ffi_contract → list ffi_annotation → ffi_contract

```

32.3 Contract Verification

FFI Violation Types

```

type ffi_violation =
| RepViolation :
  param:var_id →
  expected:rep_requirement →
  actual:abs_value →
  ffi_violation
| OwnershipViolation :
  param:var_id →
  required:access_permission →
  actual:access_permission →
  ffi_violation
| TypeMismatch :
  expected:foreign_type →
  actual:ir_type →
  ffi_violation
| NullViolation :
  param:var_id →
  ffi_violation
| UninitializedViolation :
  param:var_id →
  ffi_violation
| ThreadViolation :
  required:thread_safety_requirement →
  actual:thread_context →
  ffi_violation
| MissingContract :
  function_name:string →
  ffi_violation
| ConversionFailure :
  param:var_id →
  from_type:ir_type →
  to_type:foreign_type →
  ffi_violation

```

Verification Algorithm — VeriFFI Section 6

```

val verify_ffi_call :
  call_site : node_id →
  contract : ffi_contract →
  abstract_state : abs_state →
  result (abs_state, list ffi_violation)

val check_rep_requirement :
  abs_state → abs_value → rep_requirement → option string

let check_rep_requirement state val req =
  match req with
  | RepNonNull →
    if may_be_null state val then Some "may be null"
    else None
  | RepInitialized →
    if may_be_uninitialized state val then Some "may be uninitialized"
    else None
  | RepAligned align →

```

```

        if not (is_aligned state val align) then Some "misaligned"
        else None
    | RepSized min_size →
        if not (has_min_size state val min_size) then Some "too small"
        else None
    | RepRequired t rp →
        if not (may_satisfy_rep graph addr rp) then Some "rep predicate fails"
        else None

val permission_satisfies : access_permission → access_permission → bool

```

32.4 Reified Type Descriptors

Reified Type Descriptors — VeriFFI Section 4

```

(* Type descriptors that exist at runtime, enabling:
- Dynamic type checking at FFI boundaries
- Automatic marshalling code generation
- Reflection-based contract verification *)

```

```

module BrrrMachine.ReifiedTypes

type type_descriptor =
| TDPrimitive :
    prim:primitive_type →
    size:nat →
    align:nat →
    type_descriptor
| TDPointer :
    pointee:type_descriptor →
    type_descriptor
| TDStruct :
    name:string →
    fields:list field_descriptor →
    type_descriptor
| TDArray :
    element:type_descriptor →
    length:nat →
    type_descriptor
| TDUnion :
    variants:list (string * type_descriptor) →
    type_descriptor
| TDFunction :
    params:list type_descriptor →
    ret:type_descriptor →
    type_descriptor
| TDOpaque :
    name:string →
    type_descriptor

type primitive_type =
| PTVoid
| PTBool
| PTInt8 | PTInt16 | PTInt32 | PTInt64
| PTUInt8 | PTUInt16 | PTUInt32 | PTUInt64
| PTFloat32 | PTFloat64
| PTChar | PTWChar

type field_descriptor = {
  field_name : string;
  field_type : type_descriptor;
  field_offset : nat;
  field_padding : nat;
}

```

```

val reify_type : ir_type → type_descriptor
val size_of : type_descriptor → nat
val alignment_of : type_descriptor → nat
val validate_against_descriptor : raw_bytes → type_descriptor → bool

```

32.5 Integration with Boundary Analysis

Integration: FFI Contracts with Boundary Analysis

FFI contracts connect to the broader boundary analysis framework:

Section 9.1.2 (Boundary Risk Analysis): When BoundaryFFI edges are detected, verify against FFI contracts. Contracts specify: representation predicate requirements, ownership transfer semantics, memory effect bounds.

Section 9.3 (Guard Generation): Guards are generated from contract type requirements. Contract specifies WHAT to check; guards specify HOW.

Part XI (IR Specification): IR types can be reified into runtime type descriptors (Section 9.4.4) for FFI boundary verification.

Verification Flow:

1. Detect FFI call site
2. Look up or generate contract
3. Verify preconditions via abstract state
4. Generate guards for runtime checks
5. Apply postcondition to compute post-state
6. Report violations

Verify All FFI Calls in CPG

```

val verify_all_ffi_calls :
  cpg →
  contracts:map string ffi_contract →
  list (node_id * ffi_violation)

let verify_all_ffi_calls cpg contracts =
  let ffi_calls = find_ffi_call_sites cpg in
  List.concat_map (fun call_site →
    let func_name = get_called_function cpg call_site in
    match Map.find func_name contracts with
    | Some contract →
      let state = get_abstract_state_at cpg call_site in
      (match verify_ffi_call call_site contract state with
       | Ok _ → []
       | Error violations →
         List.map (fun v → (call_site, v)) violations)
    | None →
      [(call_site, MissingContract func_name)]
  ) ffi_calls

```

32.6 Multilingual Type Inference for FFI

Source: [Furr08] — “Checking Type Safety of Foreign Function Calls”

Multilingual Type Inference for FFI (O-Saffire/J-Saffire)

Key Insight: C glue code has a RICHER view of types than source language. C can observe physical representations (boxed vs unboxed, tags, offsets).

Approach:

1. Extract type info from high-level language (OCaml, Java)
2. Infer REPRESENTATIONAL TYPES for C glue code

3. Compare inferred types against declared types
4. Report inconsistencies as potential FFI bugs

Representational Types: Model C’s low-level view of high-level data.

- OCaml: (n, π) where n = nullary constructor count, π = sum of products
- Java: `jt jobject` where `jt` embeds Java type info into C’s `jobject`

Flow-Sensitivity (O-Saffire for OCaml): Types of form $ct\{B, I, T\}$ where:

- B = boxedness (boxed | unboxed | unknown)
- I = offset into structured block
- T = tag value or integer value

Polymorphism (J-Saffire for JNI): Unification-based analysis with polymorphic signatures. Essential for analyzing wrapper functions and the 200+ JNI API functions. JNI uses string singletons to track class names and field descriptors across C code.

GC Safety: Track which functions may invoke garbage collector via effects. Ensure pointers to GC’d heap are registered before GC-triggering calls.

Bugs Found (experimentally validated):

- O-Saffire: 24 errors + 22 suspicious patterns in 11 OCaml benchmarks
- J-Saffire: 156 errors + 124 suspicious patterns in 12 JNI benchmarks

The following F* code defines **representational types** that model how C glue code views high-level language data. The `rep_type` algebraic data type captures sum types as (nullary_count, products) pairs, where `nullary_count` is the number of nullary constructors and `products` describes the fields of non-nullary constructors. The `flow_type` extends this with flow-sensitive information: `boxedness` (is it a pointer or an immediate value?), `offset` (position within a structured block), and `tag` (known discriminator value). The `gc_effect` type tracks whether a function may trigger garbage collection, which is critical for ensuring that pointers to the managed heap are properly registered before GC-triggering calls.

Representational Types for FFI Analysis — Furr & Foster 2008

```
module BrrrMachine.FFISafety

type rep_type =
  | RepInt                                (* Unbounded integer - value unknown *)
  | RepConstructor of nat                 (* Known nullary constructor tag *)
  | RepSum of {
      nullary_count : nat;                (* Number of nullary constructors *)
      products : list rep_product;       (* Nonnullary constructor representations *)
    }
  | RepFunc of rep_type → gc_effect → rep_type
  | RepVar of string                     (* Type variable for inference *)

and rep_product = list rep_type          (* Fields of a structured block *)

and gc_effect =
  | GCNone                                (* Cannot trigger GC *)
  | GCMay                                 (* May trigger GC *)
  | GCVar of string                       (* Effect variable for inference *)

(* Flow-sensitive type extensions *)
type boxedness =
  | BBoxed                                (* Known to be pointer to heap block *)
  | BUnboxed                              (* Known to be unboxed (integer/tag) *)
  | BUnknown                              (* Boxedness unknown *)
  | BBottom                              (* Unreachable code *)

type flow_type = {
  base_type : rep_type;                  (* The representational type *)
  boxedness : boxedness;                 (* Is it boxed or unboxed? *)
}
```

```

offset : option nat;          (* Offset into structured block *)
tag : option nat;             (* Known tag value *)
}

```

FFI Type Violation Detection

```

type ffi_type_violation =
| ViolationTypeMismatch of {
    location : node_id;
    expected : rep_type;
    inferred : rep_type;
}
| ViolationUnregisteredGCPtr of {
    location : node_id;
    pointer_var : string;
    gc_triggering_call : string;
}
| ViolationInvalidOffset of {
    location : node_id;
    offset : nat;
    block_size : nat;
}
| ViolationWrongTag of {
    location : node_id;
    tested_tag : nat;
    max_valid_tag : nat;
}

(* Translate source type to representational type *)
val translate_source_type : ir_type → rep_type

(* Analyze C glue code *)
val analyze_ffi_code :
  ir_func →
  source_types : map string ir_type →
  ffi_inference_result

```

33. Occurrence Typing Analysis

Source: [TobinHochstadt08] (Typed Scheme)

Occurrence Typing: Complete Analysis Algorithm

This section provides the full occurrence typing analysis, integrating:

- Type predicate detection (Section 9.5.1)
- Conditional refinement propagation (Section 9.5.2)
- Union type narrowing (Section 9.5.3)
- Language-specific handling (Section 9.5.4)

Integration with CPG:

- Type propositions flow along CFG edges
- DDG edges track type dependencies
- Guards (Section 9.3) produce type propositions

Complements:

- Gradual typing (Section 9.1.2) at module boundaries
- Taint analysis (Section 4.2) for security
- Nullability analysis (Section 2.1.7) for null safety

33.1 Type Predicate Detection

Occurrence typing refines variable types based on runtime type tests. Different languages have different idioms for type testing: Python uses `isinstance()` and `is None`, TypeScript uses

`typeof` and `instanceof`, Go uses type assertions and type switches. The following F* code defines a unified representation `detected_type_test` that captures the essential information from any type test: the tested expression, the type being tested for, and both positive and negative type propositions. The `positive_prop` is established when the test succeeds (true branch), and `negative_prop` when it fails (false branch).

Type Predicate Detection — Tobin-Hochstadt 2008, Section 4

```
module BrrrMachine.OccurrenceTyping

(* Unified type test representation *)
type detected_type_test = {
  tested_expr : ir_expr;           (* The expression being tested *)
  tested_var : option string;      (* Variable if direct var test *)
  test_type : ir_type;            (* Type being tested for *)
  positive_prop : type_prop;       (* Proposition if test is true *)
  negative_prop : type_prop;       (* Proposition if test is false *)
  language : language_id;         (* Source language *)
  node_id : node_id;             (* Location in CPG *)
}

(* Python: isinstance(x, T), type(x) is T *)
val detect_python_type_test : ir_expr → node_id → option detected_type_test
let detect_python_type_test expr node =
  match expr with
  (* isinstance(x, T) *)
  | ECall (EVar "isinstance") [EVar var; EType ty] →
    Some {
      tested_expr = EVar var;
      tested_var = Some var;
      test_type = ty;
      positive_prop = PropHasType var ty;
      negative_prop = PropNotType var ty;
      language = Python;
      node_id = node;
    }
  (* x is None *)
  | EBinOp OpIs (EVar var) ENone →
    Some {
      tested_expr = EVar var;
      tested_var = Some var;
      test_type = TNone;
      positive_prop = PropHasType var TNone;
      negative_prop = PropNotType var TNone;
      language = Python;
      node_id = node;
    }
  | _ → None
```

TypeScript/JavaScript Type Test Detection

```
val detect_typescript_type_test : ir_expr → node_id → option detected_type_test
let detect_typescript_type_test expr node =
  match expr with
  (* typeof x === "string" etc. *)
  | EBinOp (OpEq | OpStrictEq) (ECall (EVar "typeof") [EVar var]) (EString ty_name) →
    let ty = typescript_typeof_to_type ty_name in
    Some {
      tested_expr = EVar var;
      tested_var = Some var;
      test_type = ty;
      positive_prop = PropHasType var ty;
      negative_prop = PropNotType var ty;
      language = TypeScript;
    }
```

```

    node_id = node;
  }
  (* x instanceof T *)
| EBinOp OpInstanceOf (EVar var) (EType ty) →
  Some {
    tested_expr = EVar var;
    tested_var = Some var;
    test_type = ty;
    positive_prop = PropHasType var ty;
    negative_prop = PropNotType var ty;
    language = TypeScript;
    node_id = node;
  }
  (* "prop" in obj - discriminated union check *)
| EBinOp OpIn (EString prop) (EVar var) →
  Some {
    tested_expr = EVar var;
    tested_var = Some var;
    test_type = TWithProperty prop;
    positive_prop = PropHasType var (TWithProperty prop);
    negative_prop = PropNotType var (TWithProperty prop);
    language = TypeScript;
    node_id = node;
  }
| _ → None

(* TypeScript typeof string to type mapping *)
let typescript_typeof_to_type (s : string) : ir_type =
  match s with
  | "string" → TString
  | "number" → TNumber
  | "boolean" → TBool
  | "undefined" → TUndefined
  | "object" → TObject
  | "function" → TFunction
  | "symbol" → TSymbol
  | "bigint" → TBigInt
  | _ → TAny

```

33.2 Conditional Refinement Propagation

Conditional Refinement Propagation — Tobin-Hochstadt 2008, Section 3.3

```

type branch_refinement = {
  condition_node : node_id;
  true_branch : node_id;
  false_branch : node_id;
  true_refinement : prop_type_env → prop_type_env;
  false_refinement : prop_type_env → prop_type_env;
}

(* Analyze a conditional and extract branch refinements *)
val analyze_conditional :
  cpg → node_id → language_id → option branch_refinement

(* Truthiness test: if x: ... (Python), if (x) { ... } (JS/TS) *)
val analyze_truthiness_test :
  cpg → node_id → language_id → option branch_refinement

(* Falsy types by language *)
let get_falsy_types (lang : language_id) : list ir_type =
  match lang with
  | Python → [TNone; TBool (* False *); TInt (* 0 *); TString (* "" *)]
  | JavaScript | TypeScript →

```

```

[TNull; TUndefined; TBool; TNumber (* 0, NaN *); TString (* "" *)]
| _ → []

```

Forward Propagation Algorithm

```

type occurrence_analysis_state = {
  env_at_node : map node_id prop_type_env;
  worklist : set node_id;
}

val run_occurrence_analysis : cpg → language_id → occurrence_analysis_state

let run_occurrence_analysis cpg lang =
  let entry = get_entry_node cpg in
  let initial_env = create_initial_env cpg in
  let state = ref {
    env_at_node = Map.singleton entry initial_env;
    worklist = Set.singleton entry;
  } in
  (* Worklist algorithm *)
  while not (Set.is_empty !state.worklist) do
    let node = Set.choose !state.worklist in
    state := { !state with worklist = Set.remove node !state.worklist };
    let current_env = Map.find_default initial_env node !state.env_at_node in
    if is_conditional_node cpg node then begin
      match analyze_conditional cpg node lang with
      | Some br →
          let true_env = br.true_refinement current_env in
          let false_env = br.false_refinement current_env in
          propagate_env state br.true_branch true_env;
          propagate_env state br.false_branch false_env
      | None →
          propagate_to_successors cpg state node current_env
    end
  else begin
    let new_env = occurrence_transfer current_env (get_stmt cpg node) in
    propagate_to_successors cpg state node new_env
  end
done;
!state

```

33.3 Union Type Narrowing

When occurrence typing refines a union type, it can either **narrow** the union (keeping only types that overlap with the test type) or **remove** specific types from the union. The following F* code implements these operations. The `narrow_union` function filters a union to types that overlap with the tested type—if the result is `TBottom`, the branch is unreachable (useful for exhaustiveness checking). The `remove_from_union` function removes types that are subtypes of the tested type, used in the false branch of type tests. TypeScript’s discriminated unions receive special handling via the `discriminated_union` type.

Union Type Narrowing — Tobin-Hochstadt 2008, Section 3.2

```

(* Narrow a union type by restricting to a subtype *)
val narrow_union : ir_type → ir_type → ir_type
let rec narrow_union original test_type =
  match original with
  | TUnion types →
      let narrowed = List.filter (fun t →
        types_overlap t test_type
      ) types in
      (match narrowed with
      | [] → TBottom (* Contradiction - unreachable *)
      | [t] → t (* Single type remaining *)
      )

```

```

    | ts → TUnion ts)
  | _ →
    if types_overlap original test_type then original
    else TBottom

(* Narrow a union type by removing a subtype *)
val remove_from_union : ir_type → ir_type → ir_type
let rec remove_from_union original remove_type =
  match original with
  | TUnion types →
    let remaining = List.filter (fun t →
      not (subtype t remove_type)
    ) types in
    (match remaining with
    | [] → TBottom
    | [t] → t
    | ts → TUnion ts)
  | _ →
    if subtype original remove_type then TBottom
    else original

(* Check if two types have any overlap *)
val types_overlap : ir_type → ir_type → bool
let rec types_overlap t1 t2 =
  match t1, t2 with
  | TBottom, _ | _, TBottom → false
  | TAny, _ | _, TAny → true
  | TUnion ts1, _ → List.∃ (fun t → types_overlap t t2) ts1
  | _, TUnion ts2 → List.∃ (fun t → types_overlap t1 t) ts2
  | _, _ → subtype t1 t2 || subtype t2 t1

```

Discriminated Union Handling (TypeScript)

```

(* TypeScript discriminated union: union of objects with common literal property *)
type discriminated_union = {
  discriminant : string;           (* Property name used to discriminate *)
  variants : list (ir_value * ir_type); (* (literal value, variant type) *)
}

(* Detect discriminated union pattern *)
val detect_discriminated_union : ir_type → option discriminated_union

(* Narrow discriminated union based on discriminant check *)
val narrow_discriminated_union :
  discriminated_union → string → ir_value → ir_type

(* Example:
  type Shape =
    | { kind: "circle", radius: number }
    | { kind: "square", side: number }

  After: if (shape.kind == "circle")
  Narrowed to: { kind: "circle", radius: number } *)

```

33.4 Language-Specific Type Refinement

Python-Specific Refinement

```

(* Python: hasattr() check *)
val refine_hasattr : string → string → prop_type_env → prop_type_env
let refine_hasattr var attr env =
  refine_positive env (PropHasType var (TWithAttribute attr))

(* Python: callable() check *)
val refine_callable : string → prop_type_env → prop_type_env

```

```

let refine_callable var env =
  refine_positive env (PropHasType var TCallable)

(* Python: Literal type narrowing from match statement *)
val refine_match_case :
  string → ir_pattern → prop_type_env → prop_type_env

```

TypeScript-Specific Refinement

```

(* TypeScript: Array.isArray() *)
val refine_is_array : string → prop_type_env → prop_type_env
let refine_is_array var env =
  refine_positive env (PropHasType var (TArray TAny))

(* TypeScript: type predicate functions *)
type type_predicate_fn = {
  fn_name : string;
  param_index : nat;
  asserted_type : ir_type;
}

(* Detect user-defined type predicates: function isString(x): x is string *)
val detect_type_predicate : ir_func → option type_predicate_fn

(* Apply user-defined type predicate *)
val refine_type_predicate :
  type_predicate_fn → list ir_expr → prop_type_env → prop_type_env

```

Go-Specific Refinement

```

(* Go: type switch *)
type go_type_switch = {
  switched_var : string;
  cases : list (ir_type * node_id);  (* type, case body *)
  default_case : option node_id;
}

val refine_type_switch_case :
  go_type_switch → ir_type → prop_type_env → prop_type_env

(* Go: nil check for interfaces *)
val refine_nil_check : string → bool → prop_type_env → prop_type_env
let refine_nil_check var is_nil env =
  if is_nil then
    refine_positive env (PropHasType var TNil)
  else
    refine_positive env (PropNotType var TNil)

```

33.5 Integration with CPG Analysis

CPG Integration for Occurrence Typing

```

(* CPG node annotation for occurrence typing *)
type occurrence_annotation = {
  refined_types : map string ir_type;  (* var → refined type at this point *)
  active_props : list type_prop;      (* Active propositions *)
}

(* Annotate CPG with occurrence typing results *)
val annotate_cpg_with_occurrences :
  cpg → occurrence_analysis_state → unit

(* Query refined type at a specific node *)
val get_refined_type : cpg → node_id → string → option ir_type
let get_refined_type cpg node var =
  match get_node_annotation cpg node "occurrence" with

```

```

| Some ann → Map.find_opt var ann.refined_types
| None → None

(* Use refined types in taint analysis *)
val refine_taint_with_occurrences :
  cpg → taint_state → node_id → taint_state

(* Use refined types for null analysis *)
val refine_null_with_occurrences :
  cpg → null_state → node_id → null_state

```

Cross-Reference Summary:

- Section 2.1.7b: Occurrence type domain definition
- Section 9.1.2: Integration with gradual typing
- Section 9.3.5b: Guards produce type propositions
- Section 12.3.5: F* soundness theorem
- Section 4.2: Integration with taint analysis
- Section 2.1.7: Integration with nullability domain

Part X

Incrementality and Scalability

Scalability: Two Complementary Strategies

10.1 INCREMENTALITY: Avoid re-analyzing unchanged code

- Adapton-style demand-driven computation
- Dependency tracking for minimal re-computation

10.2 TIME BUDGETS: Graceful degradation under resource pressure

- Per-function/file/total budgets
- Adaptive precision levels based on available time

Together: Scale from laptop to datacenter while maintaining useful results.

34. Demand-Driven Incremental Computation

Papers: [Hammer14] (Adapton), [Wagner98]

Real-world analysis must be incremental: when a file changes, we shouldn't re-analyze the entire codebase.

34.1 The Adapton Model

Definition 34.1 (Adapton's Key Insight [Hammer14]). Track *dependencies* between computations. When input changes, only recompute *affected* outputs.

34.1.1 Performance Characteristics

Performance Gains: $7\times$ to $2000\times$ speedups over naive recomputation for typical edit-reanalyze workflows. Traditional eager IC often incurs $4\times$ – $500\times$ *slowdowns* due to unnecessary recomputation.

34.1.2 The Demanded Computation Graph (DCG)

Definition 34.2 (Demanded Computation Graph). The DCG consists of:

- **Nodes:** Thunks (suspended computations) and refs (mutable cells)
- **Edges:** Dependencies (which thunks read which refs/thunks)

34.1.3 Adapton Primitives

- **thunk(e)** — Create suspended computation
- **force(t)** — Evaluate thunk, cache result, record dependencies
- **ref(v)** — Create mutable reference cell
- **get(r)** — Read ref (records dependency)
- **set(r, v)** — Write ref (triggers dirtying phase)

34.1.4 Inner/Outer Layer Separation

Critical for Correctness

OUTER LAYER (Driver):

- Allocates and mutates ref cells (set operations)
- Demands results by forcing thunks
- NOT incrementalized (runs once per change)

INNER LAYER (Analysis computations):

- May only READ refs (get) and force other thunks
- Cannot allocate new refs or mutate existing ones
- This restriction enables incremental reuse

34.1.5 Two-Phase Algorithm

Phase 1: Dirtying (on input change)

When `set(r, new_value)` is called:

1. Mark *r* as dirty
2. Walk *backwards* through DCG edges
3. Mark all transitive dependents as dirty
4. Stop at thunks that don't depend on *r*

Phase 2: Propagation (on demand)

When `force(t)` is called on dirty thunk:

1. Re-execute *t*'s computation
2. If result **unchanged** from cached value: Mark *t* clean, *don't* propagate to dependents
3. If result **changed**: Update cache, dependents stay dirty until forced

Theorem 34.3 (Key Insight). *Phase 2 is lazy — only runs when results demanded. This enables “demand-driven” incremental computation.*

34.1.6 Sharing, Swapping, Switching

Why DCG beats total-order IC:

- **SHARING**: Same subcomputation reused in different contexts
- **SWAPPING**: Reorder computations without invalidating memos
- **SWITCHING**: Toggle between computations, restore previous results

Traditional IC (Acar's self-adjusting computation) uses *total order*, which prevents these reuse patterns. DCG uses *partial order*.

34.1.7 Adapton Example

```
CPG(file1) ---+---> Analysis1 ---+---> Report
CPG(file2) ---+          |
CPG(file3) -----> Analysis2 ---+
```

If file1 changes:

- DIRTYING: CPG(file1), Analysis1, Report marked dirty
- CPG(file2), CPG(file3), Analysis2 stay clean

On Report demand:

- force(Report) -> force(Analysis1) -> force(CPG(file1))
- CPG(file1) recomputes, Analysis1 recomputes
- Analysis2 NOT recomputed (never forced, not dirty)

34.2 Wagner 1998: Optimal Incremental Parsing

Theorem 34.4 (Wagner's Incremental Parsing [Wagner98]). *Optimal algorithm for LR parsing with incremental edits.*

Complexity: $\mathcal{O}t + s \lg N$ where:

- *t* = new terminal symbols introduced
- *s* = modification sites (edit points)
- *N* = total tree nodes

This is asymptotically optimal for incremental LR parsing.

34.2.1 Key Technique: Sentential-Form Parsing

Traditional incremental parsing stores *parse states* in tree nodes. Wagner’s approach: **zero** per-node storage overhead beyond tree structure! Instead: Compute parse states *on demand* via breakdown procedures.

34.2.2 Balanced Sequences (Critical Insight)

Problem: Statement lists like “`stmt; stmt; stmt; ...`” are *linear*. With linear structure, incremental parsing degenerates to $\mathcal{O}N$.

Solution: Transform grammar to allow *balanced* tree representation.

Before: $L^* \rightarrow L L^* \mid \varepsilon$ (right-recursive, linear)

After: $L^* \rightarrow B \mid \varepsilon; \quad B \rightarrow L \mid B B$ (nondeterministic, balanced)

Runtime: Parser chooses balanced structure during reduce.

Result: Logarithmic access to any statement in a list.

34.2.3 Relation to Tree-sitter

Tree-sitter uses a *different* incremental algorithm (error-recovery-based). Wagner’s algorithm is more theoretically optimal but requires grammar transformation. Tree-sitter is more practical for arbitrary grammars.

Brrr-machine: Use tree-sitter for parsing, apply balanced sequence concepts to CPG representation for incremental analysis efficiency.

34.3 Incremental Analysis Strategy

The following F* formalization captures the core Adapton abstractions. The key insight is the `thunk_state` type which tracks whether a computation is *clean* (cached value is valid), *dirty* (needs recomputation), or *unevaluated* (never computed). The `deps` field in `ThunkClean` records which other thunks this computation depends on, enabling transitive invalidation.

Incremental Analysis (Adapton simplified)

```
(* =====
   INCREMENTAL ANALYSIS
   Source: Adapton (Hammer 2014), simplified
   ===== *)
module BrrrMachine.Incremental

(* -----
   THUNKS --- Cached computations with dependencies
   ----- *)
type thunk_id = nat

type thunk_state (a : Type) =
| ThunkClean : value:a → deps:set thunk_id → thunk_state a
| ThunkDirty : thunk_state a
| ThunkUnevaluated : thunk_state a

type thunk (a : Type) = {
  id : thunk_id;
  compute : unit → a;
  mutable state : thunk_state a;
}

(* Global thunk registry *)
let thunk_registry : ref (map thunk_id (∃ a. thunk a)) = ref Map.empty

(* -----
   CORE OPERATIONS
   ----- *)
```

```

(* Create a new thunk *)
val create_thunk : #a:Type → (unit → a) → thunk a
let create_thunk #a compute =
  let id = fresh_thunk_id () in
  let t = { id; compute; state = ThunkUnevaluated } in
  thunk_registry := Map.add id (| a, t |) !thunk_registry;
  t

(* Force a thunk (evaluate if needed) *)
val force : #a:Type → thunk a → a
let force #a t =
  match t.state with
  | ThunkClean v _ → v
  | ThunkDirty | ThunkUnevaluated →
    (* Track dependencies during evaluation *)
    push_dependency_context t.id;
    let result = t.compute () in
    let deps = pop_dependency_context () in
    t.state ← ThunkClean result deps;
    (* Register reverse dependencies *)
    Set.iter (fun dep_id →
      add_dependent dep_id t.id
    ) deps;
    result

(* Mark a thunk as dirty *)
val mark_dirty : thunk_id → unit
let rec mark_dirty id =
  match Map.find id !thunk_registry with
  | Some (| _, t |) →
    begin match t.state with
    | ThunkClean _ _ →
      t.state ← ThunkDirty;
      (* Propagate to dependents *)
      let dependents = get_dependents id in
      Set.iter mark_dirty dependents
    | _ → ()
    end
  | None → ()

```

The core operations `create_thunk`, `force`, and `mark_dirty` implement the two-phase algorithm:

- `force` performs lazy evaluation: if the thunk is clean, return the cached value; otherwise recompute, cache, and record dependencies.
- `mark_dirty` performs backward propagation: when a dependency changes, mark all dependent thunks as dirty recursively.

The following code shows how to build an incremental CPG where each file has its own thunk, and the merged CPG thunk depends on all file thunks:

```

Incremental CPG Construction
(* -----
   INCREMENTAL CPG
   ----- *)
type incremental_cpg = {
  (* Per-file CPG thunks *)
  file_cpgs : map string (thunk cpg);
  (* Merged CPG thunk *)
  full_cpg : thunk cpg;
  (* Analysis result thunks *)
  analyses : map string (thunk analysis_result);
}

```

```

val create_incremental_cpg : list string → incremental_cpg
let create_incremental_cpg files =
  (* Create per-file CPG thunks *)
  let file_cpgs = List.fold_left (fun m file →
    let thunk = create_thunk (fun () → parse_and_build_cpg file) in
    Map.add file thunk m
  ) Map.empty files in
  (* Create merged CPG thunk *)
  let full_cpg = create_thunk (fun () →
    let cpgs = Map.map force file_cpgs in
    merge_cpgs (Map.values cpgs)
  ) in
  { file_cpgs; full_cpg; analyses = Map.empty }

(* Update when file changes *)
val file_changed : incremental_cpg → string → unit
let file_changed icpg file =
  match Map.find file icpg.file_cpgs with
  | Some thunk → mark_dirty thunk.id
  | None → () (* New file, need to add *)

(* Get current CPG (recomputes if dirty) *)
val get_cpg : incremental_cpg → cpg
let get_cpg icpg = force icpg.full_cpg

```

The `incremental_cpg` type encapsulates a two-level thunk hierarchy: file-level CPG thunks and the merged full CPG thunk. When `file_changed` is called, only the affected file's thunk is marked dirty, and the full CPG thunk (which depends on it) becomes dirty too. On the next `get_cpg` call, only the changed file is re-parsed.

Analysis results can also be incrementalized by wrapping them in thunks that depend on the CPG:

Incremental Analysis Registration

```

(* -----
   INCREMENTAL ANALYSIS
   ----- *)
val add_analysis :
  incremental_cpg →
  name:string →
  (cpg → analysis_result) →
  incremental_cpg
let add_analysis icpg name analyze =
  let analysis_thunk = create_thunk (fun () →
    let cpg = force icpg.full_cpg in
    analyze cpg
  ) in
  { icpg with analyses = Map.add name analysis_thunk icpg.analyses }

val get_analysis : incremental_cpg → string → option analysis_result
let get_analysis icpg name =
  match Map.find name icpg.analyses with
  | Some thunk → Some (force thunk)
  | None → None

```

34.3.1 Cross-References: Incrementality Integration Points

Integration with Other Analysis Components

Section 6.5 (Memory Model): Incremental data race detection: When code changes, only re-check happens-before relations affected by the change. Use DCG to track which `candidate_execution` components depend on changed CPG nodes.

Section 8.1 (Taint Analysis): Incremental taint propagation: When taint sources/sinks

change, only recompute IFDS paths through affected graph regions. Use Wagner’s balanced sequences for efficient path updates. Use DRedL (Section 34.4) for lattice-based incremental IFDS.

Section 5.4–5.5 (Shape Analysis): Incremental shape analysis: Cache symbolic heaps per function. When function body changes, invalidate only its footprint summary. Frame rule (Section 7.4) ensures callers reuse unchanged summaries. Use compositional bi-abduction (Section 34.5) for summary reuse.

Section 12.12 (Bi-Abduction): Compositional analysis enables near-linear scaling [Dis-tefano19]. Function summaries cached and invalidated incrementally. Expected speedup: $65\times$ – $243\times$ for typical edit-reanalyze workflows.

Part XI (IR): IR node identity preservation enables incremental analysis. When source changes, map old IR nodes to new ones for cache reuse.

34.4 Incremental Lattice-Based Analysis (DRedL)

Paper: [Szabo18] — “Incrementalizing Lattice-Based Program Analyses in Datalog”

34.4.1 The Challenge

Adapton handles general incremental computation, but program analysis has *special structure*: lattice-based fixpoints. Standard incremental Datalog (DRed algorithm) only supports the powerset lattice. Many practical analyses (interval, points-to with strong updates, string analysis) require *custom lattices* with aggregation.

34.4.2 DRedL Key Insight: Increasing Replacements Can Skip Deletion

When a lattice value changes from *old_val* to *new_val*:

- If $new_val \geq old_val$ (in lattice order): **Increasing Replacement**
 - The change is *monotonic*
 - Skip expensive delete-rederive cycle
 - Propagate new value directly

34.4.3 Change Classification

MONOTONIC:

- All insertions
- Deletions that are part of increasing replacements (where $new_value \geq old_value$)

ANTI-MONOTONIC:

- All other deletions
- Require full delete-rederive treatment

34.4.4 Three-Phase Algorithm

Phase 1: ANTI-MONOTONIC (Delete)

- Process all deletions NOT in increasing replacements
- May over-delete due to cyclic dependencies
- Iterate to fixpoint within each SCC

Phase 2: RE-DERIVATION

- For deleted tuples with positive support count: re-insert
- For deleted aggregates: recompute from remaining aggregands
- Fixes over-deletion from Phase 1

Phase 3: MONOTONIC (Insert)

- Process insertions and increasing replacements
- Standard semi-naive evaluation

34.4.5 Support Tracking

For non-aggregating relations: **SUPPORT COUNTS**

- Count how many derivations produce each tuple
- Tuple deleted when count reaches 0

For aggregating relations: **SUPPORT MULTISETS**

- Track which aggregands contribute to each lattice value
- Recompute aggregate from remaining aggregands on change

34.4.6 Performance (from IncA evaluation)

- Strong-update points-to: 2.5ms median update time
- String analysis: 3.8ms median update time
- Speedup: $65\times$ – $243\times$ vs from-scratch recomputation
- Target: sub-second updates for IDE integration

34.4.7 F* Formalization of DRedL

The following F* code formalizes DRedL’s lattice requirements. The `lattice_requirements` record captures the three key assumptions from Szabo et al.: (A1) monotonic aggregation, (A2) finite lattice height for termination, and (A3) functional aggregation. The `finite_height` field uses F*’s `squash` type to carry a termination proof.

The `lattice_change` type distinguishes the four kinds of changes: insertions, deletions, increasing replacements (monotonic), and decreasing changes (anti-monotonic). The key insight is that `LatticeIncreasingReplace` carries a proof that $new_val \geq old_val$, enabling the algorithm to skip deletion.

Incremental Lattice-Based Analysis (DRedL)

```
(* =====
   INCREMENTAL LATTICE-BASED ANALYSIS (DRedL)
   Source: Szabo et al. 2018
   ===== *)
module BrrrMachine.Incremental.DRedL

(* -----
   LATTICE REQUIREMENTS (DRedL Assumptions A1-A3)
   ----- *)
type lattice_requirements (a : Type) = {
  leq : a → a → bool;                (* Partial order *)
  join : a → a → a;                  (* Least upper bound *)
  bot : a;                           (* Bottom element *)
  finite_height : squash (has_finite_height leq); (* Termination guarantee *)
}

(* Monotonic aggregation: S1 subset S2 ⇒ aggr(S1) ≤ aggr(S2) *)
type monotonic_aggregation (#a:Type) (lat : lattice_requirements a)
  (aggr : multiset a → a) =
  squash (∀ s1 s2. Multiset.subset s1 s2 ⇒ lat.leq (aggr s1) (aggr s2))

(* -----
   CHANGE CLASSIFICATION
   ----- *)
type lattice_change (a : Type) =
| LatticeInsert : key:grouping_key → new_val:a → lattice_change a
| LatticeDeletion : key:grouping_key → old_val:a → lattice_change a
| LatticeIncreasingReplace : key:grouping_key → old_val:a → new_val:a →
  pf:squash (lat.leq old_val new_val) → lattice_change a
| LatticeDecreasingChange : key:grouping_key → old_val:a → new_val:a →
  lattice_change a
```

```

(* Split changeset into monotonic/anti-monotonic *)
val split_changes :
  #a:Type →
  lat:lattice_requirements a →
  changes:list (lattice_change a) →
  (list (lattice_change a) * list (lattice_change a))
let split_changes #a lat changes =
  List.partition (fun c →
    match c with
    | LatticeInsert _ _ → true
    | LatticeIncreasingReplace _ _ _ → true (* KEY: treat as monotonic! *)
    | _ → false
  ) changes

```

The `split_changes` function is the core of DRedL’s change classification. Note the critical line: `LatticeIncreasingReplace` is treated as *monotonic*, not anti-monotonic. This is the key optimization that avoids expensive delete-rederive cycles for increasing updates.

The support tracking data structures maintain provenance information for each derived tuple:

DRedL Support Tracking

```

(* -----
   SUPPORT TRACKING
   ----- *)

type support_store = {
  counts : map (relation_name * tuple) nat;
  multisets : map (relation_name * grouping_key) (multiset lattice_value);
}

val add_aggregand :
  support_store → relation_name → grouping_key → lattice_value →
  support_store
val remove_aggregand :
  support_store → relation_name → grouping_key → lattice_value →
  support_store
val recompute_aggregate :
  #lat:lattice_requirements lattice_value →
  support_store → relation_name → grouping_key →
  option lattice_value

```

The `support_store` tracks two kinds of information: (1) `counts` maps each (relation, tuple) pair to the number of distinct derivations that produce it, and (2) `multisets` tracks which aggregands contribute to each lattice value. When a tuple’s support count drops to zero, it must be deleted. When aggregands change, the aggregate is recomputed from the remaining multiset.

The main algorithm processes changes through SCCs in topological order, executing the three phases for each component:

DRedL Main Algorithm

```

(* -----
   MAIN ALGORITHM
   ----- *)

val maintain_incrementally :
  prog:datalog_program →
  state:relation_store →
  support:support_store →
  changes:changeset →
  (relation_store * support_store)
let maintain_incrementally prog state support changes =
  (* Process SCCs in topological order *)
  List.fold_left (fun (st, sup) scc →
    maintain_component prog st sup scc changes
  ) (state, support) prog.sccs

```

```

val maintain_component :
  prog:datalog_program →
  state:relation_store →
  support:support_store →
  component:set relation_name →
  changes:changeset →
  (relation_store * support_store)
let maintain_component prog state support component changes =
  let (mon, anti) = split_changes changes in
  (* Phase 1: Anti-monotonic to fixpoint *)
  let (state1, support1, deleted) =
    anti_monotonic_fixpoint prog state support component anti in
  (* Phase 2: Re-derive over-deleted *)
  let (state2, support2) = rederive prog state1 support1 deleted in
  (* Phase 3: Monotonic *)
  monotonic_phase prog state2 support2 component mon

```

The `maintain_component` function shows the three-phase structure clearly: first anti-monotonic changes (deletions) are processed to fixpoint, then over-deleted tuples are re-derived, and finally monotonic changes (insertions and increasing replacements) are propagated. The correctness theorem states that incremental maintenance produces exactly the same result as from-scratch recomputation:

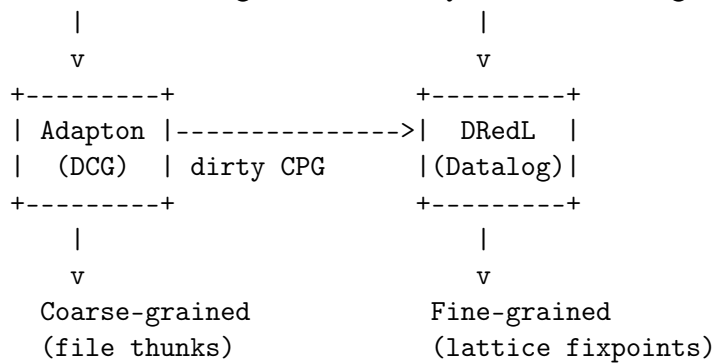
```

DRedL Correctness Theorem
(* -----
   CORRECTNESS THEOREM
   ----- *)
val dredl_correctness :
  prog:datalog_program →
  state:relation_store →
  support:support_store →
  changes:changeset →
  Lemma (
    let (inc_state, _) = maintain_incrementally prog state support changes in
    let scratch_state = compute_fixpoint_from_scratch prog (apply_changes state changes) in
    inc_state == scratch_state
  )
[SMTPat (maintain_incrementally prog state support changes)]

```

34.4.8 Integration with Adapton (Hierarchical Incrementality)

File-level changes: Analysis-level changes:



- Use Adapton for **file-level** incrementality (CPG per file)
- Use DRedL for **analysis-level** incrementality (within fixpoint)
- **Result:** Sub-second updates even for large codebases

34.5 Diff-Based Industrial Deployment (Infer)

Paper: [Distefano19] — “Scaling Static Analyses at Facebook”

Theorem 34.5 (The Key Insight). *Deployment context matters as much as analysis precision.*

34.5.1 Traditional (Batch) Deployment

1. Run analysis
2. Generate bug list
3. Present to engineers

Result: Engineers overwhelmed, bugs ignored, 0% fix rate (“bug bankruptcy”)

34.5.2 Diff-Time Deployment (Infer’s Model)

1. Engineer submits code change
2. Analysis runs *only* on diff
3. Results posted as code review comments

Result: 70% fix rate observed at Facebook scale

34.5.3 Why Diff-Time Works

1. **CONTEXT:** Developer is already reviewing the code
2. **OWNERSHIP:** The bug is in code they just wrote
3. **MANAGEABLE:** Only a few reports, not thousands
4. **TIMELY:** Fix now, not in some future cleanup sprint
5. **INCREMENTAL:** Only recompute for changed code

34.5.4 Bug Bankruptcy Anti-Pattern

When bug list exceeds ~50–100 items:

- Engineers stop triaging
- Fix rate approaches 0%
- Trust in tool decreases
- Technical debt accumulates

Prevention:

- Cap reports per diff (max 5–10)
- Prioritize by severity AND actionability
- Never let bug lists grow unbounded

34.5.5 Compositional Analysis for Scale

Infer uses bi-abduction (Section 12.12) for compositionality:

- Each procedure analyzed *independently*
- Summaries capture (precondition, postcondition) pairs
- Interprocedural analysis via summary composition
- Result: Near-linear scaling with codebase size

Definition 34.6 (Bi-Abduction Judgment).

$$p * ?M \vdash q * ?F$$

where:

- p = current state
- M = anti-frame (missing precondition requirement)
- F = frame (unchanged state passed through)

34.5.6 Scale Achieved at Facebook

- **Codebase size:** 10–100 million LOC
- **Analysis time per diff:** < 1 minute
- **Memory per analysis:** < 4GB (enables parallelization)
- **Bugs found:** 100,000+ fixed before production
- **Fix rate:** 70% diff-time vs ~5% batch

34.5.7 Zoncolan (Taint Analysis for Security)

- Codebase: 100M LOC Hack/PHP
- Taint analysis with custom sources/sinks
- Compositional summary-based propagation
- Outperforms all other security detection methods

34.5.8 F* Formalization of Deployment Strategy

The following F* code formalizes deployment modes and their expected outcomes. The `deployment_mode` type captures four distinct scenarios: diff-time (code review integration), batch (periodic runs), IDE-integrated (real-time feedback), and security engineer (deep audits). Each mode has different time/memory budgets and fix rate expectations.

The `expected_fix_rate` function encodes the empirical observation from Facebook: diff-time deployment to all engineers achieves 70% fix rate, while batch deployment drops to 5% due to the “bug bankruptcy” effect.

```
Diff-Based Industrial Deployment

(* =====
   DIFF-BASED INDUSTRIAL DEPLOYMENT
   Source: Distefano et al. 2019 (Infer at Facebook)
   ===== *)

module BrrrMachine.Deployment

(* -----
   DEPLOYMENT MODES
   ----- *)

type deployment_mode =
  | DiffTime : deployment_mode
    (* Analysis runs on code diff, reports as code review comments *)
  | BatchOffline : deployment_mode
    (* Analysis runs periodically, generates bug lists *)
  | IDE_Integrated : deployment_mode
    (* Real-time analysis in development environment *)
  | SecurityEngineer : deployment_mode
    (* Deep analysis for security team review *)

type target_audience =
  | AllEngineers : target_audience
  | SecurityTeam : target_audience
  | PlatformSpecific : platform:string → target_audience

(* Expected fix rate based on deployment context *)
val expected_fix_rate : deployment_mode → target_audience → float
let expected_fix_rate mode audience =
  match mode, audience with
  | DiffTime, AllEngineers → 0.70      (* 70% - Infer empirical *)
  | DiffTime, SecurityTeam → 0.85
  | BatchOffline, AllEngineers → 0.05  (* Bug bankruptcy effect *)
  | BatchOffline, SecurityTeam → 0.40
  | IDE_Integrated, AllEngineers → 0.80 (* Immediate context *)
  | SecurityEngineer, SecurityTeam → 0.90
  | _, _ → 0.30
```

The `analysis_budget` type captures the resource constraints for each deployment context. The key insight is that different contexts have dramatically different budgets: diff-time analysis must complete in under a minute with limited memory, while security audits can run for hours with generous resources. The `max_report_count` field prevents “bug bankruptcy” by capping how many issues are reported:

Analysis Budget per Context

```
(* -----
   ANALYSIS BUDGET PER CONTEXT
   ----- *)

type analysis_budget = {
  time_limit_per_diff : nat;      (* milliseconds *)
  memory_limit : nat;             (* megabytes *)
  max_paths_explored : nat;
  max_report_count : nat;         (* Cap to avoid overwhelming *)
}

val budget_for_context : deployment_mode → analysis_budget
let budget_for_context mode =
  match mode with
  | DiffTime → {
    time_limit_per_diff = 60000;   (* 1 minute *)
    memory_limit = 4096;           (* 4GB *)
    max_paths_explored = 1000;
    max_report_count = 5;          (* Don't overwhelm reviewer *)
  }
  | BatchOffline → {
    time_limit_per_diff = 3600000; (* 1 hour *)
    memory_limit = 32768;          (* 32GB *)
    max_paths_explored = 100000;
    max_report_count = 1000;
  }
  | IDE_Integrated → {
    time_limit_per_diff = 5000;    (* 5 seconds for interactivity *)
    memory_limit = 512;
    max_paths_explored = 100;
    max_report_count = 3;
  }
  | SecurityEngineer → {
    time_limit_per_diff = 7200000; (* 2 hours *)
    memory_limit = 65536;
    max_paths_explored = 1000000;
    max_report_count = 10000;
  }
}
```

The following code shows how to combine incremental analysis with summary caching. The `invalidation_set` function computes the transitive closure of affected functions when code changes: directly modified functions plus all their callers (since callee summaries may have changed). The `incremental_analyze` function reuses cached summaries for unchanged code and only recomputes what is necessary:

Incremental Analysis with Summary Caching

```
(* -----
   INCREMENTAL ANALYSIS WITH SUMMARY CACHING
   ----- *)

type change_δ = {
  added_functions : set func_id;
  modified_functions : set func_id;
  deleted_functions : set func_id;
  modified_callsites : set (func_id * node_id);
}

type cached_summaries = map func_id procedure_summary

(* Determine what needs recomputation *)
val invalidation_set :
  change_δ → call_graph → cached_summaries → set func_id
let invalidation_set δ cg cache =
  let directly_changed =
```

```

Set.union δ.added_functions
  (Set.union δ.modified_functions δ.deleted_functions) in
(* Transitively find affected callers *)
let rec find_affected current visited =
  if Set.is_empty current then visited
  else
    let callers = Set.fold (fun f acc →
      Set.union acc (get_callers cg f)
    ) current Set.empty in
    let new_affected = Set.diff callers visited in
    find_affected new_affected (Set.union visited new_affected)
in
find_affected directly_changed directly_changed

(* Incremental analysis - only recompute what's needed *)
val incremental_analyze :
  cpg:cpg →
  δ:change_δ →
  cache:cached_summaries →
  budget:analysis_budget →
  (cached_summaries * list bug_report)
let incremental_analyze cpg δ cache budget =
  let to_recompute = invalidation_set δ (extract_callgraph cpg) cache in
  (* Reuse unchanged summaries *)
  let valid_cache = Map.filter (fun f _ → not (Set.mem f to_recompute)) cache in
  (* Recompute affected functions *)
  let new_summaries = analyze_functions_with_cache cpg to_recompute valid_cache in
  let updated_cache = Map.union valid_cache new_summaries in
  (* Only report bugs from CHANGED code (diff-time principle) *)
  let new_bugs = extract_bugs_in_changed_code new_summaries δ in
  (* Apply report cap *)
  let capped_bugs = take budget.max_report_count
    (sort_by_actionability new_bugs) in
  (updated_cache, capped_bugs)

```

The critical insight from Infer’s deployment is that report *quality* determines fix rate. The `report_quality` record captures the key dimensions: precise location, execution trace, explanation, and suggested fix. The `actionability_score` function weights these factors based on empirical observations—reports involving changed code and short traces are most likely to be fixed:

Report Quality and Actionability

```

(* -----
  REPORT QUALITY AND ACTIONABILITY
  ----- *)
type report_quality = {
  has_precise_location : bool;
  has_trace : bool;
  has_explanation : bool;
  has_suggested_fix : bool;
  trace_length : nat;
  involves_changed_code : bool;
}

val actionability_score : bug_report → report_quality → float
let actionability_score report quality =
  let base = 0.3 in
  let score = base
    +. (if quality.has_precise_location then 0.15 else 0.0)
    +. (if quality.has_trace then 0.20 else 0.0)
    +. (if quality.has_explanation then 0.10 else 0.0)
    +. (if quality.has_suggested_fix then 0.15 else 0.0)
    +. (if quality.involves_changed_code then 0.20 else 0.0)

```

```

    -. (float_of_int quality.trace_length *. 0.01)
  in
    min 1.0 (max 0.0 score)

```

34.5.9 Industrial Deployment Checklist

- ☐ Diff-time integration with CI/CD pipeline
- ☐ Summary caching with invalidation tracking
- ☐ Report capping (max 5–10 per diff)
- ☐ Actionability scoring and prioritization
- ☐ Execution trace in reports
- ☐ Suggested fixes where possible
- ☐ Time budget enforcement (< 1 minute per diff)
- ☐ Memory budget enforcement (< 4GB)
- ☐ Graceful degradation on budget exhaustion
- ☐ Metrics: fix rate, time-to-fix, false positive rate

34.5.10 Target Metrics

- **Fix rate:** 70%+ for diff-time deployment
- **Analysis time:** < 60 seconds per diff
- **Memory:** < 4GB per analysis worker
- **Report cap:** 5 per diff (engineering), 1000 per batch (security)

35. Time Budgets and Graceful Degradation

IFDS is $\mathcal{O}ED^3$, but D (domain size) can explode. Pointer analysis on large codebases can run for hours. Real tools need time limits and graceful degradation.

Papers: [Tan22] (Qilin), [Sridharan05] (Demand-driven)

35.1 Degradation Strategy

35.1.1 Time Budgets

- Per-function budget: 100ms default
- Per-file budget: 10s default
- Total budget: configurable

35.1.2 Degradation Levels

Level 0: Full precision (Andersen + IFDS + path-sensitive)

Level 1: Reduced precision (Steensgaard + IFDS)

Level 2: Fast approximation (flow-insensitive)

Level 3: Syntactic only (pattern matching)

35.1.3 Adaptive Analysis

The following pseudo-F* code illustrates the basic strategy for adaptive analysis: attempt full-precision analysis first, but fall back to faster approximations when the time budget is exhausted. This enables graceful degradation rather than complete failure on complex code:

Adaptive Analysis with Timeout

```

val analyze_adaptive : cpg → time_budget:duration → findings
let analyze_adaptive cpg budget =
  let start = now () in
  try
    (* Start with full precision *)
    analyze_full cpg
  with Timeout →

```

```
(* Degrade and continue *)
let remaining = budget - (now () - start) in
analyze_approximate cpg remaining
```

35.1.4 Incremental Results

- Yield findings as they’re discovered
- Don’t wait for complete analysis
- Mark findings with “analysis complete” vs “partial”

35.2 Implementation TODOs

1. **Adaptive precision selection** per function based on complexity
2. **Progress reporting** for long-running analyses
3. **Checkpointing** for resumable analysis
4. **Resource monitoring** (memory, not just time)

35.3 Industrial Lessons

Source: [Distefano19]

35.3.1 Precision vs Performance Trade-off

Facebook’s Infer made a deliberate choice:

“We prioritize soundness less and precision more, accepting that some issues would be missed in exchange for fewer false positives.”

This is NOT abandoning soundness — it’s a *layered* approach:

1. **Core bi-abduction algorithm:** SOUND (formal guarantees)
2. **Industrial deployment:** UNDER-APPROXIMATE (bounded exploration)
 - Bounded symbolic execution paths
 - Timeouts on complex procedures
 - Heuristic prioritization

Result: Sound for analyzed paths, incomplete overall. Reported bugs are *real* bugs. Some bugs are *missed*.

35.3.2 Analysis Profile Configurations

| Profile | Precision | Max Time | Max Reports |
|--------------------------------|-----------|----------|-------------|
| EngineeringProfile (diff-time) | 0.90 | 60s | 5 |
| SecurityProfile (batch) | 0.50 | 3600s | 1000 |
| ComplianceProfile (audit) | 0.99 | 7200s | 10000 |

Table 35.1: Analysis Profile Configurations

Goals:

- **EngineeringProfile:** High fix rate (few false positives)
- **SecurityProfile:** Find all vulnerabilities (accept more FPs for coverage)
- **ComplianceProfile:** Full audit record (must be confident)

35.3.3 Bug Bankruptcy Prevention

When bug list exceeds ~50–100 items:

- Engineers stop triaging (overwhelmed)
- Fix rate approaches 0%

- Trust in tool decreases
- Technical debt accelerates

Prevention strategies:

- Cap reports per diff (max 5–10)
- Prioritize by severity AND actionability
- Triage queue with assignment
- Regular “bankruptcy” cleanup (declare amnesty, start fresh)
- Focus on NEW bugs in changed code

35.3.4 Actionability Requirements

Every bug report must answer:

1. **What is wrong?** (Clear error type)
2. **Where is it?** (Precise location)
3. **Why is it wrong?** (Execution trace)
4. **How to fix it?** (Actionable guidance)

Critical Observation

Reports without traces have significantly lower fix rates.

Part XI

The Brrr-Machine IR

36. Unified Intermediate Representation

All source languages are translated to a common IR. This enables:

- Single analysis implementation for all languages
- Consistent effect tracking
- Cross-language dataflow

36.1 IR Design Principles

IR Design Principles

PRINCIPLE 1: EXPLICIT EFFECTS

- Every side effect is represented explicitly.
- Pure expressions separated from effectful statements.
- No hidden control flow.

PRINCIPLE 2: EXPLICIT MEMORY

- All memory operations through uniform interface.
- Memory mode (GC/RC/owned/manual) is a parameter.

PRINCIPLE 3: SSA-LIKE STRUCTURE

- Each variable assigned once (in its scope).
- Phi nodes at control flow merge points.
- Enables efficient dataflow analysis.

PRINCIPLE 4: TYPE-PRESERVING

- Type information preserved when available.
- Abstract types for dynamic languages.
- Gradual types for mixed scenarios.

CRITICAL FOR RUST [Rupta24]: Do NOT lower Rust to LLVM IR for analysis—type info is LOST! LLVM IR loses: ownership, borrowing, lifetimes, trait bounds. Use MIR-level representation for Rust to preserve:

- Borrow regions and lifetimes
- Move vs copy semantics
- Trait object type info

See Section ?? for stack filtering which requires this info.

Pay-As-You-Go Representation (Siek 2006 Gradual Typing)

Source: [Siek06] – “Gradual Typing for Functional Languages”

KEY OPTIMIZATION: Runtime overhead only for dynamic types.

For gradual typing at language boundaries (Python↔Rust, JS↔Go):

KNOWN TYPES (statically typed):

- Use UNBOXED representation—direct machine values
- `int` → native `int64`, `bool` → native `bool`, `struct` → native layout
- No runtime type checks, no boxing overhead
- Same performance as fully static code

DYNAMIC TYPES (? / Any):

- Use BOXED representation with type tag
- Runtime type checks at cast boundaries
- { `tag: TypeTag`, `value: *void` } layout
- Overhead only where types are actually unknown

CANONICAL FORMS (Siek Lemma 5): Values of ground types (int, bool) have predictable representation:

- If $v : \text{Int}$ (not $?$) then v is unboxed integer
- If $v : ?$ then v is boxed with runtime tag

This enables efficient code generation at FFI boundaries.

APPLICATION TO IR:

- Any in `ir_type` signals “needs boxing”
- Known types (Int, Bool, TStruct) signal “unboxed OK”
- Code generator chooses representation based on type precision

Cross-reference: Section ?? for gradual typing at boundaries. Cross-reference: [Bierhoff07] review for tpestate + gradual integration.

36.2 Complete IR Specification

Brrr-Machine Intermediate Representation

```
(* =====
   BRRR-MACHINE INTERMEDIATE REPRESENTATION
   ===== *)
module BrrrMachine.IR

(* -----
   TYPES
   ----- *)
type ir_type =
  (* Primitives *)
  | TUnit : ir_type
  | TBool : ir_type
  | TInt : width:int_width → signed:bool → ir_type
  | TFloat : width:float_width → ir_type
  | TString : ir_type
  | TChar : ir_type
  (* References *)
  | TRef : pointee:ir_type → ir_type
  | TPtr : pointee:ir_type → ir_type (* Raw, potentially null *)
  | TOption : inner:ir_type → ir_type
  (* Aggregates *)
  | TArray : element:ir_type → size:option nat → ir_type
  | TStruct : name:string → fields:list (string * ir_type) → ir_type
  | TTuple : elements:list ir_type → ir_type
  | TVariant : name:string → cases:list (string * ir_type) → ir_type
  (* Functions *)
  | TFunc : params:list ir_type → ret:ir_type → effects:effect_row → ir_type
  | TClosure : captures:list ir_type → func:ir_type → ir_type
  (* Special *)
  | TAny : ir_type (* For dynamic languages *)
  | TNever : ir_type (* Bottom type, never returns *)
  | TTypeVar : name:string → ir_type (* Generic parameter *)

(* NOTE: In actual F*, int_width and float_width are declared separately
   as noeq types before ir_type. The 'and' syntax shown here is for
   illustration of the mutual dependency in the conceptual type system. *)
and int_width = | I8 | I16 | I32 | I64 | I128 | ISize | IBigInt
and float_width = | F32 | F64
```

The `ir_type` definition demonstrates several key design choices:

- `TInt` carries explicit width (I8–I128, ISize, IBigInt) and signedness, preserving platform-specific semantics from C, Rust, and Python
- `TRef` vs `TPtr` distinguishes safe references (guaranteed non-null, valid memory) from raw pointers (may be null, may dangle)

- TFunc includes an `effect_row` parameter, enabling the effect system from Part VI to track side effects at the type level
- TAny supports dynamic languages (Python, JavaScript) using gradual typing (Siek 2006)

Type Descriptors for FFI

```
(* -----
  TYPE DESCRIPTORS FOR FFI
  Source: VeriFFI (Wang et al. 2025)

  IR types can be reified into runtime type descriptors for FFI boundary
  verification. See Section 9.4.4 for the complete type_descriptor type
  and the reify_type function that converts ir_type to type_descriptor.

  This enables:
    - Dynamic type checking at FFI boundaries
    - Automatic marshalling code generation
    - Reflection-based contract verification

  Cross-reference: Section 7.5 (Representation Predicates) defines how
  type descriptors relate to actual memory layout via rep_predicate.
  ----- *)
```

The IR value type represents runtime values in a language-neutral way. Key features include language-specific null representations (VNull for C/Go, VUndefined for JavaScript, VNone for Python) and explicit width annotations for numeric types.

IR Values — Pure Data

```
(* -----
  VALUES --- Pure data

  NOTE: F* does not have a built-in 'float' type. In actual implementation,
  use FStar.Float64.t or an abstract floating-point type. The 'float' here
  is illustrative pseudo-code representing IEEE 754 floating-point values.
  ----- *)

assume type fp_value : Type (* Abstract floating-point value *)

type ir_value =
| VUnit : ir_value
| VBool : b:bool → ir_value
| VInt : i:int → width:int_width → ir_value
| VFloat : f:fp_value → width:float_width → ir_value
| VString : s:string → ir_value
| VNull : ir_value (* C/Go null pointer *)
| VUndefined : ir_value (* JavaScript undefined *)
| VNone : ir_value (* Python None singleton *)
```

IR expressions represent pure computations without side effects. The separation of pure expressions from effectful statements (defined below) is fundamental to the IR's design, enabling effect-free analysis of expression subgraphs.

IR Expressions — Pure Computations

```
(* -----
  EXPRESSIONS --- Pure computations

  Key design: expressions are PURE - they have no side effects. All memory
  access, I/O, and concurrency operations are in the statement layer.
  This separation enables compositional reasoning about expression semantics.
  ----- *)

type var_id = string
type func_id = string

type ir_expr =
  (* Atoms *)
  | Eval : v:ir_value → ir_expr
```

```

| EVar : v:var_id → ir_expr
| EGlobal : g:var_id → ir_expr
(* Arithmetic *)
| EBinOp : op:bin_op → e1:ir_expr → e2:ir_expr → ir_expr
| EUnOp : op:un_op → e:ir_expr → ir_expr
(* Comparisons *)
| ECmp : op:cmp_op → e1:ir_expr → e2:ir_expr → ir_expr
(* Type operations *)
| ECast : e:ir_expr → target:ir_type → ir_expr
| ETypeOf : e:ir_expr → ir_expr
| EInstanceOf : e:ir_expr → t:ir_type → ir_expr
(* Aggregate access (pure --- reads from value, not memory) *)
| EField : e:ir_expr → field:string → ir_expr
| EIndex : arr:ir_expr → idx:ir_expr → ir_expr
| ETupleProj : e:ir_expr → index:nat → ir_expr
(* Aggregate construction *)
| EStruct : name:string → fields:list (string * ir_expr) → ir_expr
| EArray : elements:list ir_expr → ir_expr
| ETuple : elements:list ir_expr → ir_expr
| EVariant : name:string → tag:string → payload:ir_expr → ir_expr
(* Functions *)
| ELambda : params:list (var_id * ir_type) →
    body:ir_stmt →
    captures:list var_id →
    ir_expr
| EFuncRef : f:func_id → ir_expr
(* Option/null handling *)
| ESome : e:ir_expr → ir_expr
| ENone : t:ir_type → ir_expr
| EIsSome : e:ir_expr → ir_expr
| EUnwrap : e:ir_expr → ir_expr (* Panics if None *)

and bin_op =
| OpAdd | OpSub | OpMul | OpDiv | OpMod | OpPow
| OpBitAnd | OpBitOr | OpBitXor | OpShl | OpShr
| OpAnd | OpOr

and un_op =
| OpNeg | OpNot | OpBitNot

and cmp_op =
| OpEq | OpNe | OpLt | OpLe | OpGt | OpGe
| OpRefEq (* Reference equality *)

```

IR Statements — Effectful Operations (Part 1)

```

(* -----
   STATEMENTS --- Effectful operations
   ----- *)

type ir_stmt =
(* Sequencing *)
| SSeq : s1:ir_stmt → s2:ir_stmt → ir_stmt
| SNop : ir_stmt
(* Variable binding *)
| SLet : var:var_id → typ:ir_type → init:ir_expr → body:ir_stmt → ir_stmt
| SLetMut : var:var_id → typ:ir_type → init:ir_expr → body:ir_stmt → ir_stmt
| SAssign : var:var_id → value:ir_expr → ir_stmt
(* Memory operations *)
| SAlloc : dst:var_id → typ:ir_type → ir_stmt
| SAllocArray : dst:var_id → elem_type:ir_type → size:ir_expr → ir_stmt
| SFree : ptr:ir_expr → ir_stmt
| SRead : dst:var_id → ptr:ir_expr → ir_stmt
| SWrite : ptr:ir_expr → value:ir_expr → ir_stmt
(* Aggregate memory operations *)

```

```

| SFieldRead : dst:var_id → obj:ir_expr → field:string → ir_stmt
| SFieldWrite : obj:ir_expr → field:string → value:ir_expr → ir_stmt
| SIndexRead : dst:var_id → arr:ir_expr → idx:ir_expr → ir_stmt
| SIndexWrite : arr:ir_expr → idx:ir_expr → value:ir_expr → ir_stmt
(* Control flow *)
| SIf : cond:ir_expr → then_:ir_stmt → else_:ir_stmt → ir_stmt
| SMatch : scrutinee:ir_expr → cases:list match_case → ir_stmt
| SWhile : cond:ir_expr → body:ir_stmt → ir_stmt
| SFor : var:var_id → init:ir_expr → cond:ir_expr → update:ir_stmt →
      body:ir_stmt → ir_stmt
| SForEach : var:var_id → iter:ir_expr → body:ir_stmt → ir_stmt
| SBreak : ir_stmt
| SContinue : ir_stmt
| SReturn : value:option ir_expr → ir_stmt
(* Function calls *)
| SCall : dst:option var_id → func:ir_expr → args:list ir_expr → ir_stmt
| STailCall : func:ir_expr → args:list ir_expr → ir_stmt
(* Exception handling *)
| SThrow : exn:ir_expr → ir_stmt
| STry : body:ir_stmt → catches:list catch_clause →
      finally:option ir_stmt → ir_stmt

```

IR Statements — Effectful Operations (Part 2: Concurrency)

```

(* Concurrency *)
| SSpawn : dst:var_id → func:ir_expr → args:list ir_expr → ir_stmt
| SJoin : handle:ir_expr → ir_stmt
| SSend : chan:ir_expr → value:ir_expr → ir_stmt
| SRecv : dst:var_id → chan:ir_expr → ir_stmt
| SLock : mutex:ir_expr → ir_stmt
| SUnlock : mutex:ir_expr → ir_stmt
| SAtomic : body:ir_stmt → ir_stmt
(* Channel operations --- Honda 1998/2008 session type primitives *)
| SChanCreate : dst:var_id → elem_type:ir_type → buffer_size:nat → ir_stmt
  (* Create channel: dst = make(chan elem_type, buffer_size) *)
| SChanClose : chan:ir_expr → ir_stmt
  (* Close channel: close(chan) *)
| SSelect : cases:list select_case → default:option ir_stmt → ir_stmt
  (* Select statement: select { case ch ← v: ...; case x := ←ch: ...; default: ... }
   Models both Go's select and Rust's tokio::select! *)
| SSelectSend : chan:ir_expr → value:ir_expr → body:ir_stmt → ir_stmt
  (* Select case for send: case chan ← value: body *)
| SSelectRecv : dst:var_id → chan:ir_expr → body:ir_stmt → ir_stmt
  (* Select case for receive: case dst := ←chan: body *)
| SChanDelegate : chan:ir_expr → delegated:ir_expr → ir_stmt
  (* Session delegation: transfer session capability through channel *)
| SChanBranch : chan:ir_expr → branches:list (string * ir_stmt) → ir_stmt
  (* Offer labeled branches on channel (session type branching) *)
| SChanSelect : chan:ir_expr → label:string → body:ir_stmt → ir_stmt
  (* Select labeled branch on channel (session type selection) *)

```

IR Statements — Effectful Operations (Part 3: Resources and SSA)

```

(* Resource management *)
| SAcquire : dst:var_id → resource:ir_expr → ir_stmt
| SRelease : resource:ir_expr → ir_stmt
| SDefer : action:ir_stmt → ir_stmt
| SWith : resource:ir_expr → var:var_id → body:ir_stmt → ir_stmt
(* SSA *)
| SPhi : dst:var_id → sources:list (var_id * block_id) → ir_stmt
(* Annotations *)
| SAssert : cond:ir_expr → msg:string → ir_stmt
| SAssume : cond:ir_expr → ir_stmt
| SAnnotate : annotation:ir_annotation → body:ir_stmt → ir_stmt

```

```

and match_case = {
  pattern : ir_pattern;
  guard : option ir_expr;
  body : ir_stmt;
}

(* Select case for channel select statements (Go select / Rust tokio::select!) *)
and select_case =
| SelectSend : chan:ir_expr → value:ir_expr → body:ir_stmt → select_case
  (* case chan ← value: body *)
| SelectRecv : dst:option var_id → chan:ir_expr → body:ir_stmt → select_case
  (* case dst := ←chan: body (dst=None for case ←chan: *) *)
| SelectDefault : body:ir_stmt → select_case
  (* default: body *)

and ir_pattern =
| PatWildcard : ir_pattern
| PatVar : var:var_id → ir_pattern
| PatLiteral : v:ir_value → ir_pattern
| PatVariant : tag:string → payload:ir_pattern → ir_pattern
| PatTuple : elements:list ir_pattern → ir_pattern
| PatStruct : fields:list (string * ir_pattern) → ir_pattern

and catch_clause = {
  exn_type : ir_type;
  exn_var : var_id;
  handler : ir_stmt;
}

and ir_annotation =
| AnnPure (* This code is pure *)
| AnnInline (* Inline at call sites *)
| AnnNoReturn (* This code never returns normally *)
| AnnUnsafe (* This code has unchecked assumptions *)
| AnnBoundary : source:string → target:string → ir_annotation
| AnnTainted (* Value is tainted *)
| AnnSanitized (* Value has been sanitized *)

and block_id = nat

```

36.3 SSA Form Considerations for Channels

SSA Form for Channels — Honda 1998/2008 Linearity

Channel variables have special SSA semantics due to their LINEAR nature:

1. Channel creation assigns the channel variable exactly once
2. Channel operations (send/recv/close) do NOT reassign the channel variable
3. Channel closing TERMINATES the channel's SSA lifetime
4. Select introduces phi-like merge points for channel state

Unlike regular SSA where variables can be assigned multiple values (with phi nodes merging), channel endpoints follow session type discipline:

- Each operation CONSUMES the current session type prefix
- Produces a NEW session type (the continuation)
- The channel VARIABLE stays the same, but its TYPE evolves

This is LINEAR SSA: assignment happens once, but type state changes.

Channel SSA Form Validation

```

(* Channel SSA form validation *)
val validate_channel_ssa : ir_func → list ssa_violation
let validate_channel_ssa func =

```

```

let violations = ref [] in
let channel_defs : map var_id node_id = Map.empty in
let channel_closed : set var_id = Set.empty in

let rec check_stmt stmt =
  match stmt with
  | SChanCreate dst _ _ →
    if Map.mem dst channel_defs then
      violations := SSAViolation_ChannelRedefined dst :: !violations
    else
      channel_defs := Map.add dst (current_node ()) channel_defs

  | SSend ch _ | SRecv _ ch →
    let ch_var = extract_var ch in
    if not (Map.mem ch_var channel_defs) then
      violations := SSAViolation_UseBeforeDef ch_var :: !violations;
    if Set.mem ch_var channel_closed then
      violations := SSAViolation_UseAfterClose ch_var :: !violations

  | SChanClose ch →
    let ch_var = extract_var ch in
    if Set.mem ch_var channel_closed then
      violations := SSAViolation_DoubleClose ch_var :: !violations;
    channel_closed := Set.add ch_var channel_closed

  | SSelect cases default →
    (* Select creates a merge point for channel state.
      After select, channel state depends on which case was taken.
      This is analogous to phi for channel type state. *)
    let merge_states = List.map (fun case →
      match case with
      | SelectSend ch _ _ → (extract_var ch, get_type_state ch)
      | SelectRecv _ ch _ → (extract_var ch, get_type_state ch)
      | _ → (fresh_var (), LTEnd)
    ) cases in
    (* Validate all cases lead to consistent merge state *)
    validate_select_merge merge_states

  | SSeq s1 s2 → check_stmt s1; check_stmt s2
  | SIf _ s1 s2 → check_stmt s1; check_stmt s2
  | SWhile _ body → check_stmt body
  | _ → ()
in
check_stmt func.body;
!violations

type ssa_violation =
| SSAViolation_ChannelRedefined : var_id → ssa_violation
| SSAViolation_UseBeforeDef : var_id → ssa_violation
| SSAViolation_UseAfterClose : var_id → ssa_violation
| SSAViolation_DoubleClose : var_id → ssa_violation
| SSAViolation_SelectMergeMismatch : list var_id → ssa_violation

```

Channel Type State Phi Node

```

(* Channel type state phi node for select statements *)
type chan_phi = {
  dst_chan : var_id;
  sources : list (block_id * local_session_type);
  (* After select, channel has type that is the "meet" of branch types *)
  merged_type : local_session_type;
}

(* Compute merged session type after select (conservative: use most permissive) *)

```

```

val merge_session_types : list local_session_type → local_session_type
let rec merge_session_types types =
  match types with
  | [] → LTEnd
  | [t] → t
  | t1 :: t2 :: rest →
    let merged = match t1, t2 with
    | LTEnd, _ → LTEnd
    | _, LTEnd → LTEnd
    | LTSend p1 c1, LTSend p2 c2 when p1 = p2 →
      LTSend p1 (merge_session_types [c1; c2])
    | LTRRecv p1 c1, LTRRecv p2 c2 when p1 = p2 →
      LTRRecv p1 (merge_session_types [c1; c2])
    | LTSelect bs1, LTSelect bs2 →
      (* Merge: intersection of offered labels *)
      let common = List.filter (fun (l, _) →
        List.∃ (fun (l', _) → l = l') bs2) bs1 in
      LTSelect common
    | LTBranch bs1, LTBranch bs2 →
      (* Merge: union of accepted labels *)
      LTBranch (bs1 @ bs2)
    | _, _ → LTEnd (* Incompatible types merge to end *)
    in
    merge_session_types (merged :: rest)

```

36.4 Functions and Programs

IR Functions and Programs

```

(* -----
   FUNCTIONS AND PROGRAMS
   ----- *)

type ir_func = {
  id : func_id;
  name : string;
  params : list (var_id * ir_type);
  return_type : ir_type;
  body : ir_stmt;
  effect_sig : effect_signature;
  is_public : bool;
  source_lang : string;
}

type ir_program = {
  functions : list ir_func;
  globals : list (var_id * ir_type * option ir_expr);
  entry : option func_id;
  language : language_config;
}

```

37. Semantic IR Principles for Interoperability

Source: [Patterson22] – “Semantic Soundness for Language Interoperability”

Semantic IR Design — Patterson & Ahmed 2022 (PLDI)

KEY INSIGHT: Language interoperability is soundly achieved by COMPILATION to a common target language, with convertibility relations defining glue.

PROBLEM STATEMENT: How can separately-compiled languages safely interoperate while PRESERVING source-level semantic properties (type safety, memory safety)?

SOLUTION FRAMEWORK:

1. **REALIZABILITY MODELS:** $\mathcal{V}[\tau]$ = target terms that behave as source τ

2. **CONVERTIBILITY**: $\tau_A \sim \tau_B$ iff types are interconvertible
3. **GLUE CODE**: Target-level conversion functions between representations
4. **SEMANTIC TYPE SOUNDNESS**: Source invariants preserved through compilation

CORE DEFINITIONS (Patterson Section 3):

REALIZABILITY INTERPRETATION (Definition 3.1):

$$\mathcal{V}[\tau] = \{\text{target_terms} \mid \text{they satisfy the source type's behavioral spec}\}$$

Example: $\mathcal{V}[\text{Int}] = \{0, 1, -1, 2, \dots\}$ target integers

$$\mathcal{V}[\text{Int} \rightarrow \text{Int}] = \{\text{target closures that map ints to ints}\}$$

CONVERTIBILITY RELATION (Definition 3.3): $\tau_A \sim \tau_B$ means there exists target-level glue code that:

- Converts $\mathcal{V}[\tau_A]$ to $\mathcal{V}[\tau_B]$ (AtoB direction)
- Converts $\mathcal{V}[\tau_B]$ to $\mathcal{V}[\tau_A]$ (BtoA direction)
- Preserves semantic properties (no UB introduced)

SEMANTIC TYPE SOUNDNESS (Theorem 3.5): If source program is well-typed, then:

- Compiled target program preserves source invariants
- Cross-language calls via glue code maintain type safety
- Memory safety violations in source are detected (not introduced)

CASE STUDIES (Patterson Section 5):

1. Shared memory: $\text{ptr}_A \sim \text{ptr}_B$ via pointer coercion
2. Affine/unrestricted: $\text{affine} \sim \text{linear}$ via drop tracking
3. GC/manual: $\text{gc_ptr} \sim \text{raw_ptr}$ via root registration

APPLICATION TO BRRR-MACHINE IR: The IR serves as the “target language” for interoperability. Source languages compile to IR preserving their semantic properties. Cross-language boundaries are EXPLICIT IR nodes with convertibility specs.

37.1 Realizability Models

Realizability models formalize the semantic meaning of types across language boundaries. Following Patterson & Ahmed (2022), we define $\mathcal{V}[\tau]$ as the set of IR values that “behave as” source type τ . This provides the theoretical foundation for:

- **Type-safe interoperability**: Values crossing language boundaries satisfy target type semantics
- **Glue code correctness**: Conversion functions preserve behavioral specifications
- **Sound compilation**: Source-level type invariants are preserved through compilation to IR

The `realizability_model` record captures three essential components: type interpretation as a predicate on values, soundness evidence linking interpretation to type safety, and language-specific compatibility rules.

Semantic IR for Language Interoperability

```
(* =====
  SEMANTIC IR FOR LANGUAGE INTEROPERABILITY
  Source: Patterson & Ahmed 2022, Section 3

  Formalize IR as a semantic target language where source properties
  are preserved through compilation via realizability models.
  ===== *)
```

```
module BrrrMachine.SemanticIR
```

```
(* -----
  REALIZABILITY MODELS
  V[τ] defines the set of IR values that "behave as" source type τ
```

```

----- *)

(* Semantic interpretation of types as sets of IR values *)
type realizability_model (source_lang : language_id) = {
  (* Interpret a source type as a predicate on IR values *)
  interpret : ir_type → (ir_value → bool);

  (* Evidence that interpretation is sound *)
  soundness : ∀ τ v.
    interpret τ v ⇒ ir_value_safe_at_type v τ;

  (* Language-specific type compatibility *)
  compatible : ir_type → ir_type → bool;
}

```

Rust Realizability Model

```

(* Build realizability model for each supported language *)
val rust_realizability : realizability_model
let rust_realizability = {
  interpret = (fun τ v → match τ, v with
    (* Rust integers are exact-width *)
    | TInt I32 true, VInt i I32 → Int.fits_i32 i
    | TInt I64 true, VInt i I64 → Int.fits_i64 i

    (* Rust references are non-null and point to valid memory *)
    | TRef pointee, VRef ptr →
      ptr ≠ null && valid_memory ptr (size_of pointee)

    (* Rust Options use discriminant tag *)
    | TOption inner, VVariant "Option" "Some" v → interpret inner v
    | TOption _, VVariant "Option" "None" VUnit → true

    | _, _ → false
  );
  soundness = (fun τ v → assume_rust_type_soundness τ v);
  compatible = rust_type_compatible;
}

```

The Rust realizability model above demonstrates how language-specific semantics are captured: `TInt I32 true` (signed 32-bit integer) realizes only values that fit in the i32 range; `TRef` requires non-null pointers to valid memory; `TOption` maps to Rust's `Option<T>` with discriminant tags. Each language defines its own model reflecting its type semantics.

Python Realizability Model

```

(* Python's dynamic typing means most values realize TAny *)
val python_realizability : realizability_model
let python_realizability = {
  interpret = (fun τ v → match τ, v with
    (* Python int is arbitrary precision *)
    | TInt IBigInt _, VInt _ _ → true

    (* Python None is singleton *)
    | TOption _, VNone → true
    | TOption inner, v → interpret inner v

    (* Python objects are always heap-allocated *)
    | TAny, _ → true

    | _, _ → false
  );
  soundness = (fun τ v → assume_python_gc_safety τ v);
  compatible = python_type_compatible;
}

```

37.2 Convertibility Relations

Convertibility relations define when types from different languages can be safely interconverted. The relation $\tau_A \sim \tau_B$ holds when there exist bidirectional conversion functions (“glue code”) that preserve semantic properties. This is the theoretical foundation for FFI type checking and automatic marshalling.

The convertibility record captures:

- Source and target types with their respective languages
- Bidirectional conversion expressions (`convert_AtoB`, `convert_BtoA`)
- Soundness proofs that conversions preserve realizability

Convertibility Definition

```
(* -----
   CONVERTIBILITY RELATIONS
    $\tau_A \sim \tau_B$  iff there  $\exists$  sound glue code between them
   ----- *)

type convertibility = {
  source_type : ir_type;
  target_type : ir_type;
  source_lang : language_id;
  target_lang : language_id;

  (* Glue code for conversion *)
  convert_AtoB : ir_expr; (* Convert source to target *)
  convert_BtoA : ir_expr; (* Convert target to source *)

  (* Soundness evidence *)
  sound_AtoB :  $\forall$  v.
    realizes source_lang source_type v  $\Rightarrow$ 
    realizes target_lang target_type (eval convert_AtoB v);

  sound_BtoA :  $\forall$  v.
    realizes target_lang target_type v  $\Rightarrow$ 
    realizes source_lang source_type (eval convert_BtoA v);
}
```

Establishing Convertibility Between Language Types

```
(* Establish convertibility between language types *)
val establish_convertibility :
  source_lang : language_id  $\rightarrow$ 
  target_lang : language_id  $\rightarrow$ 
  source_type : ir_type  $\rightarrow$ 
  target_type : ir_type  $\rightarrow$ 
  option convertibility

let establish_convertibility src_lang tgt_lang src_ty tgt_ty =
  match (src_lang, tgt_lang, src_ty, tgt_ty) with

  (* Rust $\leftrightarrow$ C: integers are directly compatible *)
  | (Rust, C, TInt w1 s1, TInt w2 s2) when w1 = w2 && s1 = s2  $\rightarrow$ 
    Some {
      source_type = src_ty;
      target_type = tgt_ty;
      source_lang = Rust;
      target_lang = C;
      convert_AtoB = ELambda [("x", src_ty)] (EVar "x") []; (* Identity *)
      convert_BtoA = ELambda [("x", tgt_ty)] (EVar "x") [];
      sound_AtoB = int_identity_sound w1 s1;
      sound_BtoA = int_identity_sound w1 s1;
    }
  | _  $\rightarrow$  None
```

```

(* Rust←→C: pointers require null check on Rust side *)
| (Rust, C, TRef pointee, TPtr pointee') when pointee = pointee' →
  Some {
    source_type = src_ty;
    target_type = tgt_ty;
    source_lang = Rust;
    target_lang = C;
    (* Rust ref to C ptr: direct cast (always non-null) *)
    convert_AtoB = ELambda [("x", src_ty)]
      (ECast (EVar "x") tgt_ty) [];
    (* C ptr to Rust ref: must check non-null *)
    convert_BtoA = ELambda [("x", tgt_ty)]
      (EIf (ECmp OpNe (EVar "x") ENull)
        (ECast (EVar "x") src_ty)
        (ECall (EVar "panic") [EString "null ptr"]))) [];
    sound_AtoB = rust_ref_to_c_ptr_sound pointee;
    sound_BtoA = c_ptr_to_rust_ref_sound pointee;
  }

(* Python←→Rust: requires boxing/unboxing *)
| (Python, Rust, TAny, TInt I64 true) →
  Some {
    source_type = TAny;
    target_type = TInt I64 true;
    source_lang = Python;
    target_lang = Rust;
    (* Python any to Rust i64: runtime type check + extract *)
    convert_AtoB = ELambda [("x", TAny)]
      (ECall (EVar "PyLong_AsLongLong") [EVar "x"]) [];
    (* Rust i64 to Python any: box into PyObject *)
    convert_BtoA = ELambda [("x", TInt I64 true)]
      (ECall (EVar "PyLong_FromLongLong") [EVar "x"]) [];
    sound_AtoB = python_int_extract_sound;
    sound_BtoA = python_int_box_sound;
  }

(* GC←→Manual memory: Patterson Case Study 3 *)
| (lang_gc, lang_manual, TRef pointee, TPtr pointee')
  when is_gc_language lang_gc && is_manual_language lang_manual
    && pointee = pointee' →
  Some {
    source_type = src_ty;
    target_type = tgt_ty;
    source_lang = lang_gc;
    target_lang = lang_manual;
    (* GC ref to manual ptr: register as GC root *)
    convert_AtoB = ELambda [("x", src_ty)]
      (ESeq
        (ECall (EVar "gc_register_root") [EAddr (EVar "x")])
        (ECast (EVar "x") tgt_ty)) [];
    (* Manual ptr to GC ref: verify not dangling, adopt into GC *)
    convert_BtoA = ELambda [("x", tgt_ty)]
      (ECall (EVar "gc_adopt_external") [EVar "x"]) [];
    sound_AtoB = gc_to_manual_sound lang_gc pointee;
    sound_BtoA = manual_to_gc_sound lang_manual pointee;
  }

| _ → None (* No automatic convertibility *)

```

37.3 Semantic Type Soundness for Cross-Language Calls

Cross-Language Call Definition

```
(* -----  
    SEMANTIC TYPE SOUNDNESS FOR CROSS-LANGUAGE CALLS  
    ----- *)  
  
(* Cross-language call with convertibility specification *)  
type cross_lang_call = {  
  caller_lang : language_id;  
  callee_lang : language_id;  
  callee_func : func_id;  
  arg_conversions : list convertibility;  
  ret_conversion : convertibility;  
  call_site : node_id;  
}
```

Verify Semantic Soundness of Cross-Language Calls

```
(* Verify semantic soundness of cross-language call *)  
val verify_cross_lang_soundness :  
  call : cross_lang_call →  
  caller_state : abstract_state →  
  callee_contract : ffi_contract →  
  result (abstract_state, list semantic_violation)  
  
let verify_cross_lang_soundness call caller_state contract =  
  (* Phase 1: Verify all argument conversions are sound *)  
  let arg_violations = List.filter_map (fun conv →  
    if not conv.sound_AtoB then  
      Some (UnsoundArgConversion conv.source_type conv.target_type)  
    else  
      None  
  ) call.arg_conversions in  
  
  (* Phase 2: Verify preconditions hold after conversion *)  
  let converted_args = List.map2 (fun arg conv →  
    apply_conversion conv.convert_AtoB arg  
  ) (get_args caller_state) call.arg_conversions in  
  
  let precond_violations = verify_ffi_preconditions  
    converted_args contract.precondition in  
  
  (* Phase 3: Apply callee effects and convert return value *)  
  let callee_post_state = apply_ffi_effects  
    caller_state contract.effect_bounds in  
  
  let return_conversion_sound =  
    call.ret_conversion.sound_BtoA in  
  
  let final_state = if return_conversion_sound then  
    apply_conversion call.ret_conversion.convert_BtoA callee_post_state  
  else  
    callee_post_state (* Return type error *) in  
  
  (* Phase 4: Collect all violations *)  
  let all_violations = arg_violations @  
    precond_violations @  
    (if return_conversion_sound then [] else [UnsoundReturnConversion]) in  
  
  if List.is_empty all_violations then  
    Ok final_state  
  else  
    Error (final_state, all_violations)
```

```

type semantic_violation =
  | UnsoundArgConversion of ir_type * ir_type
  | UnsoundReturnConversion
  | RealizabilityViolation of ir_value * ir_type
  | GlueCodeFailure of string

```

37.4 IR Annotations for Semantic Properties

Semantic Annotations for IR Nodes

```

(* -----
   IR ANNOTATIONS FOR SEMANTIC PROPERTIES
   Mark IR nodes with semantic preservation requirements
   ----- *)

(* Extend IR annotations for semantic IR *)
type semantic_annotation =
  | AnnRealizesType of ir_type * language_id
    (* Value realizes source type in language *)
  | AnnConvertible of convertibility
    (* Conversion is semantically sound *)
  | AnnPreservesInvariant of string
    (* Code preserves named source invariant *)
  | AnnBoundaryGlue of cross_lang_call
    (* This is glue code for cross-language call *)

```

37.5 Integration with Boundary Analysis

Integration with Part IX Boundary Analysis

Semantic IR connects to boundary detection and FFI contracts through the following integration flow:

1. **Boundary Detection** (Section ??): When `BoundaryFFI` edge is found, create `cross_lang_call` record.
2. **Convertibility Establishment** (Section ??): For each arg/return type pair, `establish_convertibility`. If `None`, report as type mismatch.
3. **Glue Code Generation** (Section ??): Generate guards from convertibility's `convert_AtoB/convert_BtoA`. Guards implement the runtime checks for conversion.
4. **Semantic Verification**: `verify_cross_lang_soundness` ensures:
 - All conversions are sound (soundness evidence exists)
 - Preconditions hold post-conversion
 - Return value correctly converted back
5. **FFI Contract Synthesis** (Section ??): Furr's type inference infers representational types. Patterson's convertibility validates inferred specs. Combined: inferred types + semantic soundness proofs.

CROSS-REFERENCES:

- Section ?? (Representation Predicates): `rep_predicate` connects to realizability interpretation $\mathcal{V}[\tau]$
- Section 29.2 (Boundary Risk): convertibility failure = high risk
- Sections ??–?? (FFI Contracts): contracts specify what convertibility must establish
- Section ?? (Furr 2008): representational types align with Patterson's target-level type interpretation

Part XII

Complete F* Formalization

38. Module Structure

Brrr-Machine F* Module Structure

```
(* =====
   BRRR-MACHINE F* MODULE STRUCTURE
   ===== *)

(*
  BrrrMachine
  +-- Core
  |   +-- AbstractDomain      -- Lattices, Galois connections
  |   +-- Domains             -- Concrete domains (intervals, taint, etc.)
  |   +-- Effects              -- Effect types and rows
  |   +-- Ownership            -- Resource algebras, ownership states
  |
  +-- Representation
  |   +-- IR                   -- Intermediate representation
  |   +-- CPG                  -- Code Property Graph types
  |   +-- CPG.Builder           -- CPG construction
  |   +-- CPG.Traversal         -- Graph traversal primitives
  |
  +-- Analysis
  |   +-- IFDS                  -- IFDS algorithm
  |   +-- PointerAnalysis       -- Andersen, Steensgaard
  |   +-- TaintAnalysis         -- Source-sink analysis
  |   +-- NullAnalysis          -- Nullability tracking
  |   +-- ResourceAnalysis      -- Lifecycle tracking
  |   +-- OwnershipAnalysis     -- Rust-style ownership
  |   +-- RaceDetection          -- Data race detection
  |
  +-- Security
  |   +-- Taint                 -- Taint sources, sinks, sanitizers
  |   +-- Vulnerabilities       -- Vulnerability types
  |   +-- SARIF                 -- Output format
  |
  +-- Boundary
  |   +-- Languages             -- Language configurations
  |   +-- Boundaries            -- Boundary detection and analysis
  |   +-- Risks                 -- Risk calculation
  |
  +-- Incremental
  |   +-- Thunks                -- Cached computations
  |   +-- IncrementalCPG        -- Incremental graph
  |
  +-- Theorems
  |   +-- Soundness              -- Soundness proofs
  |   +-- Termination            -- Termination proofs
  |   +-- Correctness            -- Correctness lemmas
  *)
```

39. Key Soundness Theorems

Foundational References

- [Cousot77] — Abstract Interpretation Soundness
- [Reps95] — IFDS Algorithm Soundness and Completeness
- [HerlihyWing90] — Linearizability and Locality Theorem
- [SabelfeldMyers03] — Information Flow Soundness
- [Zilberstein23] — Outcome Logic Theorems
- [Bruni23] — Local Completeness

Soundness Theorems Module

module BrrrMachine.Theorems.Soundness

```
(* -----
   ABSTRACT INTERPRETATION SOUNDNESS
   Source: Cousot 1977
   ----- *)
(* If abstract analysis says "safe", concrete execution is safe *)
val abstract_interpretation_sound :
  #c:Type → #a:Type →
  gc:galois_connection c a →
  transfer_concrete:(c → c) →
  transfer_abstract:(a → a) →
  (* Transfer function is sound *)
  (∀ x. gc.α (transfer_concrete x) 'leq'
    transfer_abstract (gc.α x)) →
  (* Fixpoint is sound *)
  Lemma (gc.α (lfp transfer_concrete) 'leq'
    compute_fixpoint transfer_abstract)
```

Theorem 39.1 (IFDS Soundness — [Reps95]). *The IFDS solution is sound: all reported facts actually hold.*

IFDS Soundness

```
val ifds_sound :
  #d:Type →
  problem:ifds_problem d →
  node:node_id →
  fact:d →
  (* If fact is in the solution *)
  (node, fact) 'mem' solve problem →
  (* Then there ∃ a concrete execution where fact holds *)
  Lemma (∃ exec. reaches exec node ∧ fact_holds exec node fact)
```

Theorem 39.2 (IFDS Completeness for Distributive Problems). *For distributive dataflow problems, IFDS is complete.*

IFDS Completeness

```
val ifds_complete :
  #d:Type →
  problem:ifds_problem d →
  (* Problem is distributive *)
  (∀ e s1 s2. problem.transfer e (s1 'union' s2) ==
    problem.transfer e s1 'union' problem.transfer e s2) →
  node:node_id →
  fact:d →
  (* If fact actually holds in some execution *)
  (∃ exec. reaches exec node ∧ fact_holds exec node fact) →
  (* Then it's in the solution *)
  Lemma ((node, fact) 'mem' solve problem)
```

Taint Analysis Soundness

```
(* No false negatives: all actual taint flows are detected *)
val taint_analysis_sound :
  cpq:cpg →
  sources:(node_id → option taint_source) →
  sinks:(node_id → option taint_sink) →
  sanitizers:(node_id → option sanitizer) →
  result:taint_analysis_result →
  (* Result is from our analysis *)
  result == run_taint_analysis cpq sources sinks sanitizers →
  (* For any concrete execution *)
  ∀ (exec : concrete_execution).
  (* If tainted data reaches a sink unsanitized *)
```

```

(∃ src sink. is_source src ∧ is_sink sink ∧
  reaches_unsanitized exec src sink) →
(* Then we reported it *)
Lemma (∃ flow. flow 'mem' result.flows ∧
  flow.source_location == src ∧ flow.sink_location == sink)

```

Theorem 39.3 (Locality Theorem — [HerlihyWing90]). *Linearizability is compositional.*

Locality Theorem

```

val locality_theorem :
  h:history →
  specs:(string → sequential_spec) →
  Lemma (linearizable h (combined_spec specs) ≤⇒
    (∀ obj. linearizable (subhistory h obj) (specs obj)))

```

Information Flow and Outcome Logic

```

(* If implicit flow analysis reports no leaks, noninterference holds *)
val implicit_analysis_sound :
  cpg → labeling:(string → security_level) →
  (run_implicit_analysis cpg labeling = []) →
  Lemma (termination_insensitive_noninterference (semantics cpg) labeling)

(* Falsification completeness: if spec is violated, we can prove it *)
val falsification_complete :
  pre:outcome_assertion → prog:program → post:outcome_assertion →
  Lemma (requires (not (valid_ol_triple pre prog post)))
    (ensures (valid_ol_triple pre prog (OAConj (OANot post) OATop)))

(* If locally complete, abstract analysis is EXACT for that input *)
val local_completeness_precision :
  #a:Type →
  dom:abstract_domain a →
  f:(concrete → concrete) →
  f_sharp:(a → a) →
  c:concrete →
  is_locally_complete dom f c →
  (∀ x. dom.lat.α (f x) 'leq' f_sharp (dom.lat.α x)) →
  Lemma (dom.lat.α (f c) = f_sharp (dom.lat.α c))

```

40. Provable Bug Classification (Manifest/Latent)

Replacing Heuristics with Provable Classification

Source: [Le22] (ISL) via [Vanegue25] — “Non-Termination Proving”

OLD (HEURISTIC): confidence = 0.0 to 1.0 float

Problem: No formal semantics. What does “0.7 confidence” mean?

NEW (PROVABLE): Manifest vs Latent classification

- **Manifest:** Bug triggers IN ALL CALLING CONTEXTS
- **Latent:** Bug triggers ONLY IN SPECIFIC CONTEXTS

KEY THEOREM (True Positives Property):

Manifest bugs are GUARANTEED to be real bugs (no false positives).

This is a THEOREM, not a heuristic!

Definition 40.1 (ISL Triple — [Le22]). An Incorrectness Separation Logic Triple $[p] C [q; exit]$ consists of:

- p = PRESUMPTION (existentially quantified precondition)
- C = CODE
- q = RESULT (postcondition)
- $exit$ = EXIT CONDITION (Ok or Er for error)

Semantics: If C starts in state satisfying p and terminates with exit condition, then result satisfies q .

ISL Triple and Manifest/Latent Classification

```

module BrrrMachine.ManifestLatent

type exit_condition = Ok | Er  (* Normal vs Error termination *)

type isl_triple = {
  presumption : assertion;      (* p - starting states (existential) *)
  code : cpg_slice;             (* C - analyzed code *)
  result : assertion;           (* q - ending states *)
  exit_cond : exit_condition;   (* Ok or Er *)
}

(* -----
   MANIFEST vs LATENT CLASSIFICATION (Le 2022, Definition 3.3)
   A bug is MANIFEST if it satisfies ALL of these conditions:
   1. Exit condition is Er (error)
   2. Presumption is emp ∧ true (empty heap, no constraints)
   3. Result assertion is satisfiable
   4. All heap locations in result are existentially quantified
   5. Pure constraints in result are universally satisfiable

   MANIFEST = Bug triggers regardless of calling context
   LATENT = Bug requires specific calling context to trigger
   ----- *)
type bug_classification =
| Manifest : proof:manifest_proof → bug_classification
| Latent : required_context:assertion → bug_classification
| RelaxedManifest : violations:list manifest_violation → bug_classification

type manifest_proof = {
  triple : isl_triple;
  empty_pre : squash (triple.presumption 'equiv' (emp 'conj' true_pure));
  result_sat : squash (satisfiable triple.result);
  locs_existential : squash (locs (spatial triple.result) 'subset'
                                existential_vars triple.result);
  pure_universal : squash (∀_instantiations_sat (pure triple.result));
}

```

Theorem 40.2 (True Positives Property — [Le22], Theorem 3.4). *If a bug is classified as MANIFEST, then either:*

- (a) *The code is dead (unreachable), OR*
- (b) *There EXISTS a concrete input that triggers the bug*

This is the fundamental soundness theorem for bug detection. It guarantees NO FALSE POSITIVES for manifest bugs.

True Positives Property

```

val true_positives_property :
  cpg:cpg → triple:isl_triple → proof:manifest_proof →
  Lemma (is_dead_code cpg triple.code ∨
    (∃ input. triggers_bug cpg input triple))

```

Theorem 40.3 (Falsification Completeness — [Le22], Theorem 3.5). *If a bug is LATENT with required context C , then ANY calling context satisfying C will trigger the bug.*

Falsification Completeness

```

val falsification_completeness :
  cpg:cpg → triple:isl_triple → required_ctx:assertion →
  Latent? (classify_bug triple) →
  classify_bug triple = Latent required_ctx →

```

Lemma $(\forall \text{ ctx_state. ctx_state 'satisfies' required_ctx} \Rightarrow \text{triggers_bug cpg ctx_state triple})$

Definition 40.4 (Relaxed Manifest Criterion — [Le22] Section 4). Strict manifest requires emp precondition (empty heap). RELAXED allows non-emp precondition IF:

- All heap cells in precondition are POSITIVE (allocated, not deallocated)
- No negative cells (freed memory) in precondition

A heap cell is “positive” if it represents allocated memory ($x \mapsto v$), not deallocated memory or an absence constraint.

ISL Consequence Rule

```
(* -----
   ISL CONSEQUENCE RULE (Le 2022)
   NOTE: Entailment direction is REVERSED from Hoare logic!

   Hoare: p' != p, {p} C {q}, q != q'  =>  {p'} C {q'}
   ISL:   p != p', [p] C [q], q != q'  =>  [p'] C [q']
   ~~~~~
   WEAKENED precondition entails ORIGINAL (opposite direction!)

   REASON: ISL is under-approximate. Weaker precondition means FEWER
   starting states, which is sound for "bug ∃" claims.
   ----- *)
val isl_consequence :
  triple:isl_triple →
  p':assertion{p' != triple.presumption} → (* p' entails p *)
  q':assertion{triple.result != q'} →      (* q entails q' *)
  isl_triple
```

40.1 Occurrence Typing Soundness Theorems

Occurrence Typing Soundness — [TobinHochstadt08]

The key theorem: Type refinements are SOUND with respect to runtime values.

If occurrence typing says variable x has type τ at program point p , then at runtime, the value of x is indeed a member of τ .

CRITICAL INSIGHT: Soundness requires VISIBLE predicates to be accurate. A visible predicate can only be established by an ACTUAL runtime check.

Occurrence Typing Soundness

```
module BrrrMachine.Theorems.OccurrenceTyping

(* A visible predicate is a proposition known to hold at the current point *)
type visible_predicate =
  | VPHasType : var:string → ty:ir_type → visible_predicate
  | VPNotType : var:string → ty:ir_type → visible_predicate
  | VPAnd : visible_predicate → visible_predicate → visible_predicate
  | VPOr : visible_predicate → visible_predicate → visible_predicate
  | VPTrue : visible_predicate
  | VPFalse : visible_predicate

(* Restrict operation: narrow σ to τ *)
val restrict : ir_type → ir_type → ir_type
let rec restrict σ τ =
  match σ with
  | TUnion types →
    let filtered = List.filter (fun t → types_overlap t τ) types in
    type_union_of_list filtered
  | TTop → τ
  | _ →
    if subtype σ τ then σ
```

```

    else if subtype  $\tau$   $\sigma$  then  $\tau$ 
    else TBottom (* No overlap *)

(* THEOREM: restrict is sound *)
val restrict_sound :
   $\sigma$ :ir_type  $\rightarrow$   $\tau$ :ir_type  $\rightarrow$ 
  v:runtime_value  $\rightarrow$ 
  Lemma (requires value_has_type v  $\sigma$  && value_has_type v  $\tau$ )
    (ensures value_has_type v (restrict  $\sigma$   $\tau$ ))

```

41. Under-Approximation Theorems

Dual Soundness: Over-Approximation vs Under-Approximation

VERIFICATION SOUNDNESS (over-approx): $\alpha(\text{concrete}) \subseteq \text{abstract}$

“If analysis says safe, truly safe” — may have FALSE POSITIVES

DETECTION SOUNDNESS (under-approx): $\text{abstract} \subseteq \text{concrete}$

“If analysis says bug, truly a bug” — may have FALSE NEGATIVES

The synthesis REQUIRES BOTH for complete analysis.

Under-Approximation Theorems — O’Hearn 2020

```

module BrrrMachine.Theorems.UnderApproximation

(* -----
   UNDER-APPROXIMATE TRIPLE SEMANTICS
   [p] C [q] means: every state in q is REACHABLE from some state in p.
   This is the DUAL of Hoare’s {p} C {q} (over-approximation).
   ----- *)

type under_triple (a : Type) = {
  presumption : a;          (* Starting states *)
  code : cpq;               (* Program *)
  result_ok : a;            (* Normal termination states *)
  result_err : a;           (* Error termination states *)
}

(* Semantic validity: result UNDER-approximates reachable states *)
val valid_under_approx : #a:Type  $\rightarrow$  { | abstract_domain a | }  $\rightarrow$ 
  under_triple a  $\rightarrow$  bool
let valid_under_approx #a #d t =
   $\forall$  (s : concrete_state).
  s ‘in_concretization’ t.result_ok  $\Rightarrow$ 
  ( $\exists$  (s0 : concrete_state).
    s0 ‘in_concretization’ t.presumption  $\wedge$ 
    can_reach t.code s0 s)

```

Theorem 41.1 (Reversed Consequence Rule). *CRITICAL DIFFERENCE* from Hoare logic:

- Hoare: stronger precondition, weaker postcondition
- Incorrectness: WEAKER precondition, STRONGER postcondition

Reversed Consequence Rule

```

val under_approx_consequence :
  #a:Type  $\rightarrow$  { | d : abstract_domain a | }  $\rightarrow$ 
  t:under_triple a  $\rightarrow$ 
  p':a  $\rightarrow$  q':a  $\rightarrow$ 
  (* WEAKER precondition: p  $\Rightarrow$  p' (note direction!) *)
  d.leq t.presumption p' == true  $\rightarrow$ 
  (* STRONGER postcondition: q'  $\Rightarrow$  q (note direction!) *)
  d.leq q' t.result_ok == true  $\rightarrow$ 
  valid_under_approx t  $\rightarrow$ 
  Lemma (valid_under_approx { t with presumption = p'; result_ok = q' })

```

Theorem 41.2 (Disjunction Rule: Path Dropping is Sound).

$$\frac{[p_1] C [q_1] \quad [p_2] C [q_2]}{[p_1 \vee p_2] C [q_1 \vee q_2]}$$

***CRITICAL:** Can FORGET paths in under-approximation! This is UNSOUND in over-approximation but SOUND here. Justifies partial coverage in bug finding.*

Disjunction Rule

```
val under_approx_disjunction :
  #a:Type → {/ d : abstract_domain a /} →
  t1 t2 : under_triple a →
  valid_under_approx t1 →
  valid_under_approx t2 →
  t1.code == t2.code → (* Same program *)
  Lemma (valid_under_approx {
    presumption = d.join t1.presumption t2.presumption;
    code = t1.code;
    result_ok = d.join t1.result_ok t2.result_ok;
    result_err = d.join t1.result_err t2.result_err;
  })
```

42. Manifest Error Theorems

Manifest Error Characterization — [Zilberstein23], Lemma 6.7

Some bugs are MANIFEST (context-independent) while others are LATENT (context-dependent). Outcome Logic provides the formal framework for distinguishing these.

43. Institution Theory Foundations

Institutions: Abstract Model Theory — [GoguenBurstall92]

Institutions provide the MATHEMATICAL FOUNDATION for reasoning across multiple logics and languages.

An institution $\mathcal{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ consists of:

- **Sign**: Category of signatures
- **Sen**: $\mathbf{Sign} \rightarrow \mathbf{Set}$, sentence functor
- **Mod**: $\mathbf{Sign}^{op} \rightarrow \mathbf{Cat}$, model functor (contravariant)
- \models : Satisfaction relation

THE SATISFACTION CONDITION (Goguen’s Key Axiom):

“Truth is invariant under change of notation.”

For any signature morphism $\sigma : \Sigma \rightarrow \Sigma'$, model M' in $\mathbf{Mod}(\Sigma')$, sentence ϕ in $\mathbf{Sen}(\Sigma)$:

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\phi) \iff \mathbf{Mod}(\sigma)(M') \models_{\Sigma} \phi$$

Institution Theory Formalization

```
module BrrrMachine.Institutions

(* Signature category *)
type signature
type sig_morphism = { source : signature; target : signature }

(* Sentence functor *)
type sentence (s : signature)
val translate_sentence : sig_morphism → sentence s.source → sentence s.target

(* Model functor (contravariant) *)
type model (s : signature)
```

```

val translate_model : sig_morphism → model s.target → model s.source

(* Satisfaction relation *)
val satisfies : #s:signature → model s → sentence s → bool

(* THE SATISFACTION CONDITION - FUNDAMENTAL THEOREM *)
val satisfaction_condition :
  σ : sig_morphism →
  m' : model σ.target →
  phi : sentence σ.source →
  Lemma (satisfies m' (translate_sentence σ phi) ≤⇒
    satisfies (translate_model σ m') phi)

```

Theorem 43.1 (Theory Colimits — [GoguenBurstall92], Theorem 11). *If the category of signatures has colimits, then the category of theories has colimits.*

Application: Multi-language analysis combines language theories via pushouts:

Python theory ← Common theory → Rust theory

44. C11 Memory Model Theorems

C11 Memory Model — [Batty11]

“Mathematizing C++ Concurrency”

This section provides the complete formal verification of the C11 memory model theorems, building on Section 6.5’s definitions.

Theorem 44.1 (DRF-SC: Data Race Freedom implies Sequential Consistency). **THE FUNDAMENTAL THEOREM OF THE C11 MEMORY MODEL**

If a program has no data races under SC semantics, then all its behaviors are sequentially consistent.

This is the contract between programmer and implementation:

- *Programmer ensures: no data races (via proper synchronization)*
- *Implementation guarantees: SC semantics (simple reasoning)*

Formal Statement:

$$\text{DRF}(P) \Rightarrow \forall \text{exec}. \text{consistent}(\text{exec}) \Rightarrow \exists \text{sc_exec}. \text{SC}(\text{sc_exec}) \wedge \text{observable}(\text{exec}) = \text{observable}(\text{sc_exec})$$

DRF-SC Theorem

```

val data_race_free_implies_sc :
  program:program_type →
  (* Precondition: Program has no data races when executed under SC *)
  Lemma (requires is_drf_program_formal program)
  (* Postcondition: ALL consistent executions are SC-equivalent *)
  (ensures ∀ (exec : candidate_execution).
    is_execution_of program exec ⋈⋈
    is_consistent exec ⇒
    ∃ (sc_exec : candidate_execution).
      is_execution_of program sc_exec ⋈⋈
      is_sc_execution sc_exec ⋈⋈
      same_observable_behavior exec sc_exec)

```

CRITICAL WARNING: Thin-Air Limitation

The C11 axiomatic model ([Batty11]) permits “out-of-thin-air” values in programs using relaxed atomics. Coherence axioms are NECESSARY but NOT SUFFICIENT to prevent thin-air values.

Safe Subset: Programs using ONLY release/acquire synchronization (no relaxed atomics) are safe. See Theorem 44.2.

Recommended: For sound reasoning about programs with relaxed atomics, use Promising Semantics 2.0 ([Lee20]).

Theorem 44.2 (Release-Acquire Programs are DRF). *Programs using only release-acquire synchronization (no relaxed atomics) are automatically DRF and thus SC.*

Release-Acquire Soundness

```
val only_release_acquire_is_drf :
  program:program_type →
  Lemma (requires uses_only_release_acquire program)
    (ensures is_drf_program_formal program)
```

45. Effect Absence Theorems

Effect Absence — [Leijen14]

“Koka: Programming with Row-polymorphic Effect Types”

Key semantic insight: if an effect is NOT present in the inferred effect row, the program CANNOT exhibit that behavior. This gives us mathematically proven bug detection capabilities.

Theorem 45.1 (Exception Soundness — [Leijen14], Theorem 2). *If the inferred effect row does NOT contain the exception effect (exn), then evaluation can NEVER produce an unhandled exception.*

Formal Statement:

$$\forall e : \text{Expr}, \tau : \text{Type}, \epsilon : \text{Effect}. \quad (\Gamma \vdash e : \tau \mid \epsilon) \wedge \text{exn} \notin \epsilon \Rightarrow \neg(e \rightarrow^* \text{throw } c)$$

Exception Soundness

```
val exception_soundness :
  γ:typing_context →
  e:expr →
  τ:type_ →
  eff:effect_row →
  (* Precondition: e is well-typed with effect eff, and exn not in eff *)
  Lemma (requires
    type_judgment γ e τ eff &&
    effect_definitely_absent eff LExn)
  (* Postcondition: evaluation cannot produce unhandled exception *)
  (ensures
    ∀ (outcome : eval_outcome).
      evaluates_to e outcome ==>
      not (OutThrow? outcome) &&
      not (OutHeapThrow? outcome))
```

Theorem 45.2 (Termination Guarantee — [Leijen14], Theorem 4). *If the inferred effect row does NOT contain the divergence effect (div), then evaluation is GUARANTEED to terminate.*

Termination Guarantee

```
val termination_guarantee :
  γ:typing_context →
  e:expr →
  τ:type_ →
  eff:effect_row →
  Lemma (requires
    type_judgment γ e τ eff &&
    effect_definitely_absent eff LDiv)
  (ensures
    ∃ (outcome : eval_outcome).
      evaluates_to e outcome &&
      not (OutDiverge? outcome))
```

Theorem 45.3 (State Isolation — [Leijen14], Theorem 3). *If the inferred effect row does NOT contain a state effect ($\text{st}\langle h \rangle$), then evaluation CANNOT produce heap-bound answers for that heap.*

46. Session Type Theorems

Multiparty Asynchronous Session Types — [Honda08], [Honda98]

This section formalizes the theoretical foundations of multiparty session types for analyzing communication protocols in concurrent and distributed systems.

Global and Local Type Syntax

```
(* Global type syntax - describes conversation from global viewpoint *)
type global_type =
  | GMsg : sender:participant → receiver:participant →
        channel:nat → payload:sort → continuation:global_type → global_type
  | GBranch : sender:participant → receiver:participant →
        channel:nat → branches:list (label * global_type) → global_type
  | GPar : left:global_type → right:global_type → global_type
  | GRec : var:type_var → body:global_type → global_type
  | GVar : var:type_var → global_type
  | GEnd : global_type

(* Local type syntax - describes behavior from single participant's viewpoint *)
type local_type =
  | LSend : channel:nat → payload:sort → continuation:local_type → local_type
  | LRecv : channel:nat → payload:sort → continuation:local_type → local_type
  | LSelect : channel:nat → branches:list (label * local_type) → local_type
  | LBranch : channel:nat → branches:list (label * local_type) → local_type
  | LRec : var:type_var → body:local_type → local_type
  | LVar : var:type_var → local_type
  | LEnd : local_type
```

Theorem 46.1 (Communication Safety — [Honda08], Theorem 5.5). *For well-typed, coherent processes, communication is always safe: linearity is preserved and delivered values match receiving prefixes.*

Theorem 46.2 (Session Fidelity — [Honda08], Corollary 5.6). *Process execution follows global specification: if $\Delta(s) = \text{encode}(G)$ and the process reduces at s_k , then there exists G' such that G reduces at k to G' and $\text{encode}(G') = \Delta'(s)$.*

Theorem 46.3 (Progress — [Honda08], Theorem 5.12). *For simple, well-linked, well-typed processes: every reachable state is either terminated or can take a step.*

47. Sparse Value-Flow Analysis Theorems

SVF — [Sui12], [Sui16]

ISSTA 2012: “Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis”

CC 2016: “SVF: Interprocedural Static Value-Flow Analysis in LLVM”

Theorem 47.1 (Memory SSA Soundness). *If pts is a sound may-alias analysis, then Memory SSA construction captures all concrete def-use relationships for address-taken variables.*

Theorem 47.2 (SVFG Construction Preserves Def-Use — [Sui12], Theorem 1). *For every def-use pair (d, u) in the concrete semantics, there exists a path in the SVFG from the node representing d to the node representing u .*

Theorem 47.3 (Leak Detection Correctness — [Sui12], Theorem 3). *If `detect_leak` returns `DefinitelyLeaks`, then every concrete execution that allocates at that source will leak that allocation.*

48. Selective Context Sensitivity Theorems (ZIPPER)

ZIPPER — [Li20]

“A Principled Approach to Selective Context Sensitivity for Pointer Analysis”

Key insight: Only approximately 38% of methods benefit from context sensitivity. The remaining 62% receive CS treatment that wastes analysis time.

ZIPPER identifies precision-critical methods via three observable value-flow patterns: Direct, Wrapped, and Unwrapped flows.

Theorem 48.1 (PFG Captures Flows). *If a flow pattern exists from In method M_1 to Out method M_2 of class C , then there exists a path in PFG_C from a parameter of M_1 to a return of M_2 .*

Theorem 48.2 (Precision Preservation (Empirical)). *ZIPPER-guided selective CS preserves 98.8% of full 2obj precision.*

Source: [Li20], Table 1

49. Chopped Symbolic Execution Theorems

Chopped Symbolic Execution — [Trabish18]

“Chopped Symbolic Execution” (ICSE 2018)

CSE provides directed symbolic execution by skipping irrelevant code and recovering on-demand when dependencies are discovered.

Theorem 49.1 (CSE Soundness). *All bugs found by CSE are real bugs (reachable in concrete execution).*

Exception: *Non-termination in skipped functions is not detected.*

CSE Soundness

```
val cse_soundness :
  prog:ir_program →
  skip_funcs:skip_set →
  target:program_point →
  bug:bug_witness →
  found:bool{found = cse_finds_bug prog skip_funcs target bug} →
  terminates:bool{terminates = all_skips_terminate prog skip_funcs bug.input_model} →
  Lemma (found && terminates ==> reachable_concrete prog target bug)
```

Theorem 49.2 (Relative Completeness). *For bugs NOT depending on skip internals, CSE will find them (if full SE would find them).*

50. Python Function Type Graph Theorems

JARVIS — [Huang23]

“JARVIS: Scalable and Precise Application-Centered Call Graph Construction for Python”

FTG provides flow-sensitive type inference for Python with strong updates and C3 MRO support.

Theorem 50.1 (FTG Type Safety). *If FTG infers type T for expression e , then at runtime e will have a type that is a subtype of T .*

Theorem 50.2 (Call Graph Soundness). *All actual calls during execution are captured in the computed call graph. No false negatives for call edges.*

51. Data Structure Analysis (DSA) Theorems

DSA — [Lattner07]

“Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World”

Theorem 51.1 (Completeness Preservation). *If a node has the Complete flag, it will never be merged with another node in subsequent analysis phases.*

Theorem 51.2 (DSA Soundness). *If two pointers p and q may alias in concrete execution, then they point to the same DS node (or at least one is incomplete).*

Theorem 51.3 (Heap Cloning Distinguishes Instances). *If two pointers point to objects allocated at different call contexts to the same allocation function, they are in different DS nodes.*

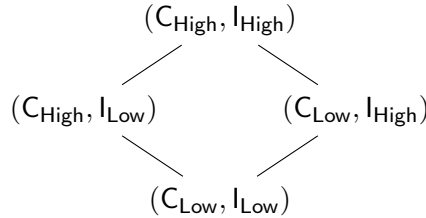
52. Robust Declassification Theorems

Robust Declassification — [ChongMyers04]

“Security Policies for Downgrading”

Security under declassification requires tracking BOTH confidentiality AND integrity. Robustness is fundamentally an INTEGRITY property: low-integrity (attacker-controlled) code cannot influence what information is declassified.

Definition 52.1 (4-Point Security Lattice). The minimal structure needed for robust declassification:



Ordering: $(c_1, i_1) \sqsubseteq (c_2, i_2)$ iff $c_1 \sqsubseteq_C c_2 \wedge i_2 \sqsubseteq_I i_1$

Note: Integrity is INVERTED in the product lattice.

Definition 52.2 (Semantic Security Under Declassification). A program is semantically secure under a declassification policy iff for all low-equivalent initial memories, the visible outputs are equivalent:

$$M_1 \sim_L M_2 \Rightarrow \text{visible}(P(M_1)) \sim_{vis} \text{visible}(P(M_2))$$

Theorem 52.3 (Robustness Implies Security — [ChongMyers04], Theorem 3). *If the static robustness check passes for all declassification points, then semantic security holds.*

Robustness Condition: For a declassification at program point p with expression e :

1. $\text{PC_integrity}(p) = \text{HIGH}$ (no low-integrity code on path to p)
2. For all $v \in \text{deps}(e)$: $\text{integrity}(v) = \text{HIGH}$ (no low-integrity data)

52.1 Speculative Non-Interference Theorems

SPECTECTOR — [Guarnieri20]

Speculative Non-Interference ensures that speculative execution does not reveal more than non-speculative execution.

Definition 52.4 (SNI). A program satisfies Speculative Non-Interference iff:

$$\forall m_1, m_2. \quad m_1 \sim_L m_2 \Rightarrow (\text{trace}_{\text{spec}}(m_1) = \text{trace}_{\text{spec}}(m_2) \Rightarrow \text{trace}_{\text{std}}(m_1) = \text{trace}_{\text{std}}(m_2))$$

Theorem 52.5 (Worst-Case Predictor Soundness). *SNI with worst-case predictor implies SNI with any concrete predictor.*

52.2 Constant-Time Product Program Theorems

CT-Verif — [Almeida16]

Constant-time security is a 2-safety hyperproperty that can be reduced to assertion checking via product program construction.

Theorem 52.6 (Product Reduction — [Almeida16]). *A program is constant-time secure iff its product program is assertion-safe:*

$$\text{is_ct_secure}(\text{prog}, \text{model}, \text{pub}) \iff \text{is_assertion_safe}(\text{construct_product}(\text{prog}, \text{model}))$$

This is the main soundness theorem, verified in Coq by the CT-Verif authors.

53. Capability Algebra Theorems

Capability Algebra — [CraryWalkerMorrisett99]

“Typed Memory Management”

Ownership states track LIFECYCLE (acquired \rightarrow in-use \rightarrow released). Multiplicities track ALIASING (unique vs potentially aliased). Both are needed for sound memory management.

Definition 53.1 (Multiplicity Types). • MUnique: Provably no aliases exist. CAN free safely.

• MDup: May have aliases. Can only ACCESS, not free.

This is STRONGER than “owned” — owned but aliased CANNOT free.

Theorem 53.2 (Free Requires Unique). *Only unique capabilities can safely free memory. This prevents use-after-free from aliased pointers.*

Free Requires Unique

```

val can_free : capability → region → bool
let can_free c r =
  match Map.find r c with
  | Some MUnique → true
  | _ → false

val free_unique_safe :
  heap:heap → cap:capability → r:region →
  can_free cap r == true →
  (∀ ptr. ptr 'points_to' r ==> ptr 'in' cap_owned_ptrs cap r) →
  Lemma (free heap r 'results_in' safe_heap)

```

Theorem 53.3 (Complete Collection — [Crary99], Section 4.2). *Well-typed terminating programs return ALL memory to the system. This proves LEAK-FREEDOM for the type system.*

Complete Collection

```

val complete_collection :
  prog:ir_program →
  well_typed prog →

```

```

Lemma (
  diverges prog ∨ (* Either it runs forever... *)
  (∃ result. terminates prog result ∧
    final_heap prog == empty_heap) (* ...or heap is empty *)
)

```

54. Realizability Theorems

Realizability for Multi-Language Soundness — [Patterson22]

“Semantic Soundness for Interoperability”

Matthews-Findler syntactic boundaries are insufficient. For SEMANTIC soundness, interpret source types as sets of TARGET values.

Theorem 54.1 (Shared Memory Identity — [Patterson22], Section 4.2). *Sharing* ref τ_1 with ref τ_2 (no copy) requires $V[\tau_1] = V[\tau_2]$.

Examples:

- Rust `Umut i32` \leftrightarrow C `int*` (same target representation)
- Python `list` \leftrightarrow C `array` (different representations, must copy)

55. Verified Interpreter Pattern

Verified Interpreter Pattern — [Watt18]

“Mechanising and Verifying the WebAssembly Specification”

The verified interpreter pattern separates RELATIONAL specification from EXECUTABLE implementation, then proves equivalence. This provides a verified executable that is provably correct with respect to the declarative semantics.

Key Insight: Declarative specifications are easy to reason about but cannot execute. Executable implementations can run but are complex. By proving equivalence, we get the best of both worlds.

The following F* code demonstrates the core pattern: a relational reduction specification alongside an executable step function, with soundness and completeness theorems proving their equivalence.

Verified Interpreter Core Pattern

```

module BrrrMachine.Theorems.VerifiedInterpreter

(* -----
  SEPARATION OF SPECIFICATION AND IMPLEMENTATION
  Relational spec: declarative, easy to reason about
  Executable impl: can actually run, may be complex
  Prove they're equivalent → verified executable.
  ----- *)

(* Relational specification (declarative) *)
val reduce : cpg → state → state → bool
(* "reduce cpg s1 s2" means s1 reduces to s2 in one step *)

(* Executable implementation (can run) *)
val step : cpg → state → option state
(* "step cpg s" computes next state if possible *)

(* SOUNDNESS: step produces valid reductions *)
val step_sound :
  cpg:cpg → s:state → s':state →
  step cpg s = Some s' →
  Lemma (reduce cpg s s')

```

```

(* COMPLETENESS: all reductions can be computed *)
val step_complete :
  cpg:cpg → s:state → s':state →
  reduce cpg s s' →
  Lemma (step cpg s = Some s')

(* Combined: step is a decision procedure for reduce *)
val step_decides_reduce :
  cpg:cpg → s:state → s':state →
  Lemma (step cpg s = Some s' ≤⇒ reduce cpg s s')

```

The control result type below handles WebAssembly-style control flow without complex evaluation context encoding. This simplifies the interpreter while maintaining semantic precision.

Control Result and Evaluation

```

(* -----
   CONTROL RESULT TYPE
   WebAssembly-style control flow handling.
   Avoids complex evaluation context encoding.
   ----- *)
type control_result =
  | Normal : state:state → control_result
  | Break : depth:nat → values:list value → control_result
  | Return : values:list value → control_result
  | Trap : reason:string → control_result

(* Evaluation with control flow *)
val eval_with_control : cpg → state → control_result
let rec eval_with_control cpg state =
  match step cpg state with
  | None → Normal state (* Stuck = terminated *)
  | Some state' →
    match check_control state' with
    | Some (Break d vs) → Break d vs
    | Some (Return vs) → Return vs
    | Some (Trap r) → Trap r
    | None → eval_with_control cpg state'

```

Theorem 55.1 (Verified Type Checker — [Watt18], Section 4). *The same pattern applies to type checking: a relational typing judgment paired with an executable type checker, with proofs of soundness and completeness.*

Verified Type Checker

```

(* Relational typing judgment *)
val has_type : context → expr → ir_type → bool

(* Executable type checker *)
val check_type : context → expr → option ir_type

(* Type checker soundness *)
val type_check_sound :
  ctx:context → e:expr → τ:ir_type →
  check_type ctx e = Some τ →
  Lemma (has_type ctx e τ)

(* Type checker completeness *)
val type_check_complete :
  ctx:context → e:expr → τ:ir_type →
  has_type ctx e τ →
  Lemma (check_type ctx e = Some τ)

```

Locale-Style Abstraction for Portable Proofs

By abstracting over arithmetic implementation details (e.g., IEEE 754 NaN behavior), proofs work for ALL conforming implementations. This is essential for cross-platform analysis correctness.

Abstract Arithmetic Interface

```
(* Abstract arithmetic interface *)
class arithmetic_impl (a : Type) = {
  add : a → a → a;
  mul : a → a → a;
  div : a → a → option a;  (* Partial for division by zero *)
  (* Axioms implementations must satisfy *)
  add_comm : squash (∀ x y. add x y == add y x);
  add_assoc : squash (∀ x y z. add (add x y) z == add x (add y z));
  (* Non-determinism specification for IEEE 754 *)
  nan_behavior : a → bool;
}

(* Analysis sound for ALL conforming implementations *)
val analysis_sound_for_all_arith :
  #a:Type → {| arithmetic_impl a |} →
  cpg:cpg → result:analysis_result →
  result == run_analysis cpg →
  Lemma (∀ (impl : arithmetic_impl a).
    sound_wrt_impl cpg result impl)
```

56. System Dependence Graph Theorems

SDG and Two-Phase Slicing — [HorwitzRepsBinkley90]

“Interprocedural Slicing Using Dependence Graphs”

Two-phase slicing prevents spurious interprocedural paths. Weiser’s transitive closure is WRONG (context-insensitive).

Theorem 56.1 (Two-Phase Context Sensitivity). *The two-phase algorithm produces context-sensitive slices: only nodes on REALIZABLE paths are included.*

Two-Phase Slicing

```
val interprocedural_slice : sdg → criterion:node_id → set node_id
let interprocedural_slice sdg crit =
  let phase1_exclude = [DefOrder; ParamOut] in
  let phase1_marked = reach_backwards sdg crit phase1_exclude in
  let phase2_exclude = [DefOrder; Call; ParamIn] in
  let phase2_marked = reach_backwards sdg phase1_marked phase2_exclude in
  Set.union phase1_marked phase2_marked

val two_phase_context_sensitive :
  sdg:sdg → crit:node_id →
  slice:set node_id { slice = interprocedural_slice sdg crit } →
  Lemma (∀ n. Set.mem n slice ⇒
    ∃ path. is_realizable_path sdg path ∧
      head path = n ∧ last path = crit)
```

57. Probabilistic Analysis Theorems

Probabilistic Abstract Interpretation — [CousotMonerau12]

“Probabilistic Abstract Interpretation”

Probabilistic programs are measurable functions from scenario space to deterministic semantics. Classical abstract interpretation is the special case where the scenario space is a singleton. This insight enables:

- Branch probability estimation without profiling
- Confidence scoring for bug reports
- Graceful degradation from full probabilistic to non-deterministic analysis

The following F* code formalizes scenario spaces as probability spaces over which deterministic semantics are indexed. This generalizes classical AI where $\Omega = \{*\}$ (singleton).

Scenario Space Abstraction

```
module BrrrMachine.Theorems.Probabilistic

(* -----
SCENARIO SPACE ABSTRACTION
S_p[[P]] : Omega → D
Where:
- Omega = probability space (scenarios)
- D = deterministic semantics
- mu = probability measure on Omega
----- *)

type scenario_space (ω : Type) = {
  space : ω;
  measure : ω → real;
  (* Measure axioms *)
  measure_positive : squash (∀ s. measure s ≥ 0.0);
  measure_total : squash (integral measure = 1.0);
}

type prob_semantics (ω : Type) (d : Type) = {
  scenarios : scenario_space ω;
  sem : ω → d; (* Interpretation per scenario *)
}

(* -----
CLASSICAL AI AS SPECIAL CASE
When Omega = {*} singleton, probabilistic AI collapses to classical AI.
----- *)

type singleton_scenario = unit

val classical_is_singleton :
  #a:Type → dom:abstract_domain a →
  prob:prob_semantics singleton_scenario a →
  prob.scenarios.space == () →
  Lemma (prob_domain_of prob = dom)
```

Law abstraction projects to probability distributions, enabling precision-vs-cost tradeoffs. The law ordering reflects information content: more precise analyses assign higher probability to precise properties.

Law Abstraction and Branch Probability

```
(* -----
LAW ABSTRACTION
Abstract to probability distributions (laws).
Law ordering reflects precision.
----- *)
```

```

----- *)
type law (a : Type) = (a → bool) → real (* Pr(property holds) *)

val law_leq : #a:Type → law a → law a → bool
let law_leq l1 l2 =
  (* l1 more precise than l2 if assigns higher prob to precise properties *)
  ∀ (q : a → bool). l1 (downset q) ≥ l2 (downset q)

(* Law abstraction preserves soundness *)
val law_abstraction_sound :
#a:Type → {| d : abstract_domain a |} →
concrete_law : law a →
abstract_law : law a →
law_leq concrete_law abstract_law →
Lemma (∀ prop. concrete_law prop ≥ abstract_law prop)

(* -----
BRANCH PROBABILITY ESTIMATION
Compute branch probability from abstract domain constraints.
No profiling needed!
----- *)
val estimate_branch_prob :
cpg:cpg → cond:node_id →
input:abstract_state →
real (* Probability condition is true *)
let estimate_branch_prob cpg cond input =
let cond_expr = get_condition cpg cond in
let true_states = restrict input cond_expr in
let false_states = restrict input (negate cond_expr) in
(* Estimate based on abstract domain volume *)
let true_vol = volume true_states in
let false_vol = volume false_states in
true_vol /. (true_vol +. false_vol)

```

Theorem 57.1 (Graceful Degradation). *Analysis precision can be traded for performance by merging scenarios. Soundness is preserved: degraded analysis may have less information but no false negatives for detected bugs.*

Precision Level and Degradation

```

type precision_level =
| FullProbabilistic      (* All scenarios distinct *)
| PartialProbabilistic   (* Some scenarios merged *)
| NonDeterministic       (* Omega = singleton, no probability *)

val analyze_with_precision :
cpg:cpg → level:precision_level → budget:duration →
analysis_result

(* Degradation preserves soundness *)
val degradation_sound :
cpg:cpg →
full:analysis_result →
degraded:analysis_result →
full == analyze_with_precision cpg FullProbabilistic infinity →
degraded == analyze_with_precision cpg NonDeterministic infinity →
(* Degraded may have less info but no false negatives *)
Lemma (∀ bug. bug 'mem' full.bugs ==> bug 'mem' degraded.bugs)

```

58. Unified Constraint Domain Theorems

Unified Constraint Domain — [XiPfenning99]

“Dependent Types in Practical Programming”

All abstract domains (interval, taint, nullability, resource, ownership) share a common structure: they are instantiations of Dependent ML with different constraint systems C . This insight unifies:

- **Implementation:** One parametric domain module, many instantiations
- **Composition:** Cross-domain reasoning via constraint conjunction
- **Decidability:** Constraint satisfiability gives analysis termination
- **Theory:** Single soundness proof covers all domains

The DML(C) pattern indexes types by constraint domain values. The following signature captures all our abstract domains uniformly.

Unified Constraint Domain Signature

```
module BrrrMachine.Theorems.ConstraintDomains

(* -----
   UNIFIED CONSTRAINT DOMAIN SIGNATURE
   Every abstract domain is an instance of this interface.
   ----- *)

class constraint_domain (c : Type) = {
  (* Index sorts and terms *)
  index_sort : Type;
  index_term : Type;
  (* Constraint language *)
  constraint_ : Type;
  (* Core operations *)
  satisfiable : constraint_ → bool;
  implies : constraint_ → constraint_ → bool;
  (* Abstract interpretation operations *)
  meet : constraint_ → constraint_ → constraint_;
  join : constraint_ → constraint_ → constraint_;
  widen : constraint_ → constraint_ → constraint_;
  (* Axioms *)
  implies_refl : squash (∀ c. implies c c);
  implies_trans : squash (∀ a b c. implies a b && implies b c => implies a c);
  meet_glb : squash (∀ a b. implies (meet a b) a && implies (meet a b) b);
  join_lub : squash (∀ a b. implies a (join a b) && implies b (join a b));
}
```

Each of our analysis domains becomes an instance of this unified interface:

Domain Instantiations

```
(* Interval domain: C = linear integer arithmetic *)
instance interval_constraint_domain : constraint_domain interval = {
  index_sort = IntSort;
  index_term = LinExpr; (* a*x + b*y + ... + c *)
  constraint_ = LinConstraint;
  satisfiable = fourier_motzkin_sat;
  implies = fun c1 c2 → not (satisfiable (And [c1; Not c2]));
  meet = fun c1 c2 → And [c1; c2];
  join = fun c1 c2 → convex_hull c1 c2;
  widen = interval_widen;
}

(* Nullability domain: C = boolean constraints *)
instance null_constraint_domain : constraint_domain nullability = {
  index_sort = BoolSort;
```

```

index_term = BoolExpr;
constraint_ = BoolConstraint;
satisfiable = sat_solve;
implies = fun c1 c2 → not (satisfiable (And [c1; Not c2]));
meet = fun c1 c2 → And [c1; c2];
join = fun c1 c2 → Or [c1; c2];
widen = id;  (* Boolean domain is finite *)
}

(* Taint domain: C = security lattice constraints *)
instance taint_constraint_domain : constraint_domain taint = {
  index_sort = SecurityLevel;
  index_term = LevelExpr;
  constraint_ = FlowConstraint;  (* l1 ≤ l2 *)
  satisfiable = lattice_sat;
  implies = fun c1 c2 → lattice_implies c1 c2;
  meet = fun c1 c2 → And [c1; c2];
  join = fun c1 c2 → lub_constraint c1 c2;
  widen = id;  (* Security lattice is finite *)
}

```

Theorem 58.1 (Array Bounds Verification). *Using the unified constraint framework, array bounds can be statically verified via constraint solving, eliminating runtime checks when provable.*

Array Bounds Check Elimination

```

type bounds_check_result =
  | StaticallyVerified : proof:constraint_proof → bounds_check_result
  | NeedsRuntimeCheck : bounds_check_result
  | AlwaysOutOfBounds : witness:index_term → bounds_check_result

val check_array_bounds :
  ctx:index_constraint →  (* What we know *)
  arr:refined_array →
  idx:index_term →
  bounds_check_result
let check_array_bounds ctx arr idx =
  let bounds_ok = ICAAnd
    (ICPred "≥" [idx; IConst 0])
    (ICPred "<" [idx; IVar arr.length_var])
  in
  if implies ctx bounds_ok then
    StaticallyVerified (construct_proof ctx bounds_ok)
  else if satisfiable (ICAAnd ctx (ICNot bounds_ok)) then
    NeedsRuntimeCheck
  else
    AlwaysOutOfBounds (find_counterexample ctx bounds_ok)

(* Bounds check elimination theorem *)
val bounds_check_elimination :
  ctx:index_constraint → arr:refined_array → idx:index_term →
  check_array_bounds ctx arr idx = StaticallyVerified _ →
  Lemma (safe_to_access arr idx)

```

Theorem 58.2 (Conservative Extension — [XiPfenning99]). *DML(C) is conservative over ML: existing code works unchanged. Index erasure preserves typing.*

Conservative Extension

```

(* Index erasure *)
val erase_indices : ir_refined_type → ir_type

(* Conservative extension: typing preserved under erasure *)
val conservative_extension :
  ctx:context → e:expr → rt:ir_refined_type →
  has_refined_type ctx e rt →

```

Lemma (has_type (erase_ctx ctx) (erase_expr e) (erase_indices rt))

59. Bi-Abduction Theorems

Bi-Abduction — [Calcagno09], [Mulder22]

“Compositional Shape Analysis by means of Bi-Abduction”

Given current state p and target q , bi-abduction discovers BOTH the missing resource M (anti-frame) AND the leftover resource F (frame).

Judgment: $p * ?M \vdash q * ?F$

Theorem 59.1 (Compositionality — [Distefano19]). *Analysis can be done function-by-function. Summaries capture precondition/postcondition contracts.*

Compositional Soundness

```
val compositional_soundness :
  cpg:cpg → func:func_id → table:map func_id procedure_summary →
  summary:procedure_summary →
  analyze_procedure_compositionally cpg func table = Some summary =>
  (* For any calling context satisfying the precondition... *)
  (∀ ctx. ctx 'entails' summary.precondition =>
    (* ...execution produces state satisfying postcondition *)
    ∃ result. executes_to cpg func ctx result ∧
      result 'entails' summary.postcondition)
```

60. Divergence (Non-Termination) Theorems

Non-Termination Proving — [Vanegue25]

“Non-Termination Proving: 100 Million LoC and Beyond”

To PROVE divergence, use UNDER-approximation. Two main detection strategies:

1. Repeating abstract states (loops)
2. Same-state recursive calls (recursion)

Theorem 60.1 (Repeating State Soundness). *A loop diverges if execution returns to the same abstract state. This is SOUND for proving divergence (under-approximation).*

Repeating State Detection

```
val repeating_state_sound :
  cpg:cpg → w:divergence_witness{InfiniteLoop? w} →
  Lemma (∃ input. satisfies input w.path_condition ∧
    diverges (execute cpg input))
```

61. Widening and Narrowing Theorems

Widening/Narrowing — [Cousot92]

“Comparing the Galois Connection and Widening/Narrowing Approaches”

Widening/narrowing is STRICTLY MORE POWERFUL than finite Galois connections. No finite domain can achieve equivalent results for all programs.

Critical insight: Discovered invariants may NOT appear in program text! Example: McCarthy’s F91 → analysis discovers $[91, \text{maxint} - 10]$.

Theorem 61.1 (Widening Termination). *For any ascending chain with widening, the widened chain stabilizes.*

Widening Termination

```

val widening_terminates :
  #l:lattice → widen:widening #l → transfer:(l.carrier → l.carrier) →
  Lemma (∃ n.
    let rec iter k x = if k = 0 then x else iter (k-1) (widen x (transfer x)) in
    let result = iter n l.bottom in
    l.order (transfer result) result) (* Post-fixpoint *)

```

Theorem 61.2 (Narrowing Improves Precision).

Narrowing Improvement

```

val narrowing_improves :
  #l:lattice → narrow:narrowing #l → transfer:(l.carrier → l.carrier) →
  post_widen:l.carrier →
  l.order (transfer post_widen) post_widen → (* post_widen is a post-fixpoint *)
  Lemma (let refined = narrow post_widen (transfer post_widen) in
    l.order (transfer refined) refined ∧ (* Still post-fixpoint *)
    l.order refined post_widen) (* But smaller *)

```

62. Symbolic Execution Theorems

Symbolic Execution — [King76]

“Symbolic Execution and Program Testing”

Key concepts:

1. Path condition (pc): Boolean expression over symbolic inputs
2. Symbolic values: Expressions over input symbols
3. Forking: When condition cannot be resolved, explore both branches
4. Commutativity: $\text{instantiate}(\text{exec_symbolic}(P)) = \text{exec_concrete}(P)$

Theorem 62.1 (Commutativity — [King76], Section 5). *The fundamental correctness criterion for symbolic execution.*

Symbolic Execution Soundness

```

val symbolic_execution_sound :
  cpg:cpg → func:ir_func → tree:exec_tree → v:valuation →
  (* For any valuation satisfying some leaf's path condition... *)
  (∃ leaf. leaf 'mem' tree.leaves ∧
    pc_satisfied (get_node tree leaf).state.pc v) ⇒
  (* ...the symbolic result instantiated equals concrete result *)
  Lemma (
    let leaf = find_satisfied_leaf tree v in
    let sym_env = (get_node tree leaf).state.env in
    let concrete_result = execute_concrete cpg func v in
    Map.for_all (fun var sym_val →
      instantiate sym_val v = Map.find var concrete_result
    ) sym_env)

```

63. Frame Rule and Local Reasoning

Separation Logic Frame Rule — [Reynolds02]

“Separation Logic: A Logic for Shared Mutable Data Structures”

The Frame Rule enables COMPOSITIONAL analysis by allowing us to reason about a command using only its “footprint” (accessed memory).

Theorem 63.1 (Frame Rule — [Reynolds02], Theorem 1).

$$\frac{\{p\} c \{q\}}{\{p * r\} c \{q * r\}}$$

Side condition: r does not mention modified variables or freed locations.

Frame Rule

```

val frame_rule :
  p:sl_assertion → c:ir_stmt → q:sl_assertion →
  r:sl_assertion →
  hoare_valid p c q →
  disjoint (footprint c).writes (free_vars r) →
  disjoint (footprint c).frees (locs_in r) →
  Lemma (hoare_valid (SLSep p r) c (SLSep q r))

```

Theorem 63.2 (Parallel Rule — [Reynolds02], Section 11.3).

$$\frac{\{p_1\} c_1 \{q_1\} \quad \{p_2\} c_2 \{q_2\}}{\{p_1 * p_2\} c_1 \parallel c_2 \{q_1 * q_2\}}$$

Side condition: Disjoint footprints.

Corollary: Disjoint parallel threads do not race.

64. Effect Handler Correctness Theorems

Algebraic Effects — [PlotkinPretnar09]

“Handlers of Algebraic Effects”

Key insight: Handlers are HOMOMORPHISMS from free algebraic models. A handler is CORRECT if it forms a valid model of the effect theory.

CRITICAL LIMITATIONS:

- Continuations (call/cc) are NOT algebraic effects
- Parallel composition CANNOT be expressed with unary handlers

Effect Signature and Handler

```

module BrrrMachine.EffectHandlers

type operation = {
  name : string;
  param_type : Type;
  arity : nat;
  result_types : list Type;
}

type effect_signature = {
  operations : list operation;
  equations : list effect_equation;
}

type handler (e:effect_signature) (m:Type) = {
  handle_return : value → m;
  handle_op : op:operation{op 'mem' e.operations} →
    op.param_type →
    (op.result_types → m) →
    m;
}

```

Theorem 64.1 (Handler Correctness — [PlotkinPretnar09]). *A handler is CORRECT if it preserves the effect theory’s equations.*

Handler Correctness

```

val handler_correct :
  #e:effect_signature → #m:Type → h:handler e m →
  (* Handler must preserve all equations *)
  (∀ eq. eq 'mem' e.equations =>
    interpret_with_handler h eq.lhs == interpret_with_handler h eq.rhs)

```

65. Probabilistic Semantics Theorems

Probabilistic Programs — [Kozen81]

“Semantics of Probabilistic Programs”

Key insight: Programs are LINEAR OPERATORS on MEASURES. Dual semantics:

1. Operational: $f_S : X^{n+\omega} \rightarrow X^{n+\omega}$ (input with random tape \rightarrow output)
2. Denotational: $S : B \rightarrow B$ (input distribution \rightarrow output distribution)

Theorem 65.1 (Discrete Sufficiency — [Kozen81], Theorem 6.1). *If two programs agree on all point masses, they agree on all distributions.*

Discrete Sufficiency

```
val discrete_sufficiency :
  s1:(state → linear_operator state) →
  s2:(state → linear_operator state) →
  (* If they agree on all point masses... *)
  (∀ x. s1 (point_mass x) == s2 (point_mass x)) ==>
  (* ...they agree on all distributions *)
  Lemma (∀ mu. s1 mu == s2 mu)
```

Application: Testing on discrete inputs is SUFFICIENT for correctness validation.

66. Set Constraint Resolution Theorems

Set Constraints — [Aiken99]

“Introduction to Set Constraint-Based Program Analysis”

Key insight: Dataflow, type inference, and closure analysis are ALL instances of set constraint solving. Constraint form:

$$E_1 \subseteq E_2 \quad \text{where} \quad E ::= \alpha \mid \emptyset \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E \mid c(E_1, \dots, E_n) \mid c^{-i}(E) \mid Y \Rightarrow X$$

Theorem 66.1 (Inductive Form Existence — [Aiken99], Section 5). *Any satisfiable constraint system can be transformed to inductive form with finite representation.*

Inductive Form

```
val inductive_form_theorem :
  cs:constraint_system →
  Lemma (satisfiable cs ≤=>
    ∃ (sys:inductive_system).
      ∀ sol. satisfies_all sol cs ≤=>
        ∃ β. sol = extract_solution sys β)
```

Theorem 66.2 (Function Subtyping (Contravariance)).

$$(\tau_1 \rightarrow \tau_2) \subseteq (\sigma_1 \rightarrow \sigma_2) \iff \sigma_1 \subseteq \tau_1 \wedge \tau_2 \subseteq \sigma_2$$

Note: Domain is CONTRAVARIANT, codomain is covariant.

67. Datalog Compilation and Analysis Composition

Datalog Compilation — [Jordan16], [Madsen16]

“Souffle: On Synthesis of Program Analyzers” and “From Datalog to Flix”

Key Insight: COMPILE Datalog rules to specialized code, don’t interpret them. Compilation achieves 50x+ speedup over interpretation (bddbdb, muZ).

Benchmark (OpenJDK7: 1.4M variables, 350K objects, 160K methods):

Souffle (compiled): 35 seconds | bddbdb: 30 minutes | SQLite: 6h20m | muZ: DNF

The Souffle compilation pipeline uses staged specialization via Futamura projections. Each stage eliminates interpretation overhead while preserving semantics.

Datalog Compilation Pipeline

```
module BrrrMachine.DatalogCompile

(* -----
RELATIONAL ALGEBRA MACHINE (RAM) OPERATIONS
Source: Jordan, Scholz, Subotic 2016
RAM is the intermediate representation between Datalog rules and
generated code. Enables optimization before code generation.
----- *)

type ram_expr =
| RAMScan : relation:string → ram_expr
| RAMFilter : expr:ram_expr → column:nat → value:string → ram_expr
| RAMProject : expr:ram_expr → columns:list nat → ram_expr
| RAMJoin : left:ram_expr → right:ram_expr →
    left_col:nat → right_col:nat → ram_expr
| RAMUnion : left:ram_expr → right:ram_expr → ram_expr
| RAMDiff : left:ram_expr → right:ram_expr → ram_expr

type ram_stmt =
| RAMInsert : target:string → source:ram_expr → ram_stmt
| RAMLoop : δ:string → body:list ram_stmt → ram_stmt (* Semi-naïve *)
| RAMSeq : stmts:list ram_stmt → ram_stmt
```

Theorem 67.1 (Optimal Index Selection via Dilworth’s Theorem — [Jordan16]). *Given N index requirements from queries, find MINIMAL set of indices such that each requirement is subsumed by at least one selected index.*

Insight: Index subsumption forms a partial order. A lexicographic index $[a, b, c]$ subsumes $[a, b]$ and $[a]$. Dilworth’s theorem: the width of a partial order equals the minimum number of chains needed to cover all elements.

Dilworth-Based Index Selection

```
(* Index specification *)
type index_spec = {
  relation : string;
  key_columns : list nat; (* Lexicographic order *)
}

(* Index subsumption: [a,b,c] subsumes [a,b] and [a] *)
let subsumes (super sub : index_spec) : bool =
  super.relation = sub.relation &&
  List.length sub.key_columns ≤ List.length super.key_columns &&
  List.for_all2 (=) sub.key_columns
    (List.take (List.length sub.key_columns) super.key_columns)

(* Dilworth-based minimal index selection *)
val compute_minimal_indices :
  required:list index_spec →
  minimal:list index_spec{
    (* Every required index is subsumed by some minimal index *)
    ∀ r. List.mem r required =>
      ∃ m. List.mem m minimal ∧ subsumes m r
  }
```

Lattice-Extended Datalog — [Madsen16]

Standard Datalog operates on relations (finite sets). Flix extends Datalog with lattice predicates, enabling elegant expression of IFDS, IDE, and value analyses.

Key Difference:

- Standard Datalog: Multiple tuples with same key = set (keep all)
 - Flix Extension: Same key = JOIN lattice values (collapse to single)
- This enables constant propagation, interval analysis, and IDE (micro-function lattices).

The following F* code shows how IDE is expressed as Flix with a specific lattice: the micro-function space of environment transformers.

IDE as Lattice-Extended Datalog

```
module BrrrMachine.IDE

(* -----
   IDE IS FLIX WITH MICRO-FUNCTION LATTICE
   Source: Madsen, Yee, Lhotak 2016

   IFDS path edge:  PathEdge(d1, n, d2)          -- fact d2 holds at n
   IDE path edge:   PathEdge(d1, n, d2, f)        -- with environment xformer
                   ^
                   micro-function lattice element

   IDE key insight: transformers COMPOSE along paths
   PathEdge(d1, m, d3, compose(f, g)) :-
     PathEdge(d1, n, d2, f),
     CFG(n, m),
     EdgeFunction(n, d2, m, d3, g).
   ----- *)

(* Micro-function: environment transformer *)
type micro_fn (v : Type) = v → v

(* Micro-functions form a lattice under pointwise ordering *)
instance micro_fn_lattice (v : Type) { | complete_lattice v | } :
  complete_lattice (micro_fn v) = {
    bot = (fun _ → bot);
    top = (fun _ → top);
    join = (fun f g → fun x → join (f x) (g x));
    meet = (fun f g → fun x → meet (f x) (g x));
    leq = (fun f g → ∀ x. leq (f x) (g x));
  }

(* Micro-function composition for IDE path edges *)
val compose_micro : #v:Type → { | complete_lattice v | } →
  micro_fn v → micro_fn v → micro_fn v
let compose_micro f g = fun x → f (g x)

(* Distributivity requirement for IFDS/IDE soundness *)
val distributive_transfer :
  #d:Type → { | complete_lattice d | } →
  f:micro_fn d →
  Lemma (requires ∀ x y. f (join x y) == join (f x) (f y))
    (ensures distributive f)
```

Theorem 67.2 (Semantic Preservation — [Jordan16]). *The staged translation preserves Datalog semantics:*

$$\text{result} = \llbracket \text{Int} \rrbracket(\text{Source}, \text{Input}) = \llbracket \text{Compiled} \rrbracket(\text{Input})$$

Compilation Correctness

```
(* Compilation correctness theorem *)
val compilation_correct :
  prog:datalog_program →
  Lemma (eval_compiled (compile_to_ram prog) == datalog_semantics prog)

(* IFDS can be expressed as Datalog for compilation *)
```

```

val ifds_to_datalog : #d:Type → ifds_problem d → datalog_program

(* Compiled IFDS computes same result as tabulation algorithm *)
val compiled_ifds_correct : #d:Type →
  prob:ifds_problem d →
  Lemma (eval_compiled (compile_ifds prob) == ifds_tabulation prob)

```

68. Local-to-Global Consistency Theorems

Local-Global Consistency — [Cousot77]

“Abstract Interpretation: A Unified Lattice Model”

Key insight: If abstract interpretation is locally consistent with concrete semantics, then global analysis results are sound. This enables MODULAR soundness proofs.

Theorem 68.1 (Fundamental Theorem of Abstract Interpretation — [Cousot77], Theorem 4.1). *Local consistency implies global consistency.*

Local-Global Consistency

```

val local_global_consistency :
  #c:Type → #a:Type →
  gc:galois_connection c a →
  f_c:(c → c) → f_a:(a → a) →
  locally_consistent gc f_c f_a →
  monotone f_c → monotone f_a →
  Lemma (∀ n.
    gc.α (iterate f_c n gc.concrete_lattice.bot) 'leq_a'
    iterate f_a n gc.abstract_lattice.bot)

(* Corollary: Fixpoints are consistent *)
val fixpoint_consistency :
  #c #a gc f_c f_a →
  locally_consistent gc f_c f_a →
  monotone f_c → monotone f_a →
  Lemma (gc.α (lfp f_c) 'leq_a' lfp f_a)

```

69. Boundary Guard Soundness Theorems

Boundary Guards — [MatthewsFindler07]

“Operational Semantics for Multi-Language Programs”

Boundaries between languages require GUARDS to ensure type safety. Guards have POLARITY that flips at function types.

Definition 69.1 (Guard Polarity). • **Positive**: Value entering STRICTER language (dynamic → static). MUST check.

• **Negative**: Value from STRICTER language (static → dynamic). Can skip.

Theorem 69.2 (Guard Soundness — [MatthewsFindler07], Theorem 1). *Guards ensure type soundness at boundaries.*

Guard Soundness

```

val guard_soundness :
  source:language_config →
  target:language_config →
  e:expr → ty:ir_type →
  pol:guard_polarity{pol = polarity source target} →
  guard:guard{guard = generate_guard ty pol} →
  well_typed source e ty →
  Lemma (reduces_safely target (apply_guard guard (eval source e)))

```

Theorem 69.3 (Lump Cancellation). *Boundary pairs cancel: $\hat{\tau} \text{MS}(\text{SM } \hat{\tau} v) \rightarrow v$*

Cancellation

```

val cancellation_sound :
  e:boundary_expr →
  Lemma (eval_boundary e = eval_boundary (cancel_boundaries e))

```

70. Vault Adoption and Focus Theorems

Vault — [DeLine01], [DeLine02], [DeLine04]

“Vault, Fugue: Tystate for Object-Oriented Languages”

Adoption permanently transfers unique ownership to shared-frozen state. Focus temporarily upgrades shared permission to unique within a scope.

Theorem 70.1 (Adoption Preserves Tystate). *Adoption is irreversible and freezes tystate.*

Adoption Soundness

```

val adoption_preserves_tystate :
  perm:access_permission_v2{APUnique_v2? perm} →
  state_at_adoption:state_node →
  Lemma (let (APUnique_v2 r n g) = perm in
    let adopted = APPure_v2 r n g (Some n) in
      n = state_at_adoption =>
        forever_frozen adopted n)

val adoption_irreversible :
  original:access_permission_v2{APUnique_v2? original} →
  Lemma (let adopted = perform_adoption original in
    APPure_v2? adopted ∧
    not (∃ op. apply_transition adopted op = Some (APUnique_v2? _)))

```

Theorem 70.2 (Focus Scope Soundness). *Focus temporarily upgrades shared to unique, with proper scope management.*

Focus Soundness

```

val focus_scope_sound :
  original:access_permission_v2{APPure_v2? original} →
  lock_acquired:bool →
  body:(access_permission_v2 → access_permission_v2) →
  (FocusSuccess? (enter_focus original lock_acquired)) =>
  Lemma (let FocusSuccess token unique = enter_focus original lock_acquired in
    let result = body unique in
      let restored = exit_focus token result in
        APPure_v2? restored ∧
        get_state restored = get_state result)

```

71. Stack Filtering Theorems (Rupta)

Stack Filtering — [Li24]

“Rupta: Efficient and Precise Pointer Analysis for Rust”

NOTE: This is a NOVEL technique not found in prior literature.

Stack objects allocated in function f are ONLY alive when f ’s frame is on the call stack. Filtering out dead stack objects dramatically improves both PRECISION and SPEED.

Theorem 71.1 (Stack Filtering Soundness). *If we filter out a stack location from a points-to set, then that location cannot be pointed to in any concrete execution at that program point.*

Stack Filtering Soundness

```

val stack_filtering_sound :
  cg:call_graph → pts:pts_solution →
  var:var_id → ctx:context →

```

filtered:set abstract_loc{filtered = filter_dead_stack cg pts var ctx} →
Lemma $(\forall \text{ loc. loc 'in' filtered} \Rightarrow$
loc 'in' concrete_pts var ctx)

Table 71.1: Part XII Theorem Cross-References

| Theorem | Source | Related Sections |
|-----------------------------------|--------------------------|------------------|
| Abstract Interpretation Soundness | [Cousot77] | Section 2.1 |
| IFDS Soundness/Completeness | [Reps95] | Section 4.1 |
| True Positives Property | [Le22] | Section 12.3 |
| Occurrence Typing Soundness | [TobinHochstadt08] | Section 2.1.7b |
| DRF-SC Theorem | [Batty11] | Section 6.5 |
| Effect Absence Theorems | [Leijen14] | Section 6.1.3 |
| Session Type Safety | [Honda08] | Section 7.3 |
| SVF Soundness | [Sui12] | Section 5.6 |
| ZIPPER Precision | [Li20] | Section 5.3.2 |
| CSE Soundness | [Trabish18] | Section 4.4.6 |
| FTG Soundness | [Huang23] | Section 5.3.3 |
| DSA Soundness | [Lattner07] | Section 5.2.5 |
| Robustness Theorem | [ChongMyers04] | Section 8.1.4.3 |
| SNI Soundness | [Guarnieri20] | Section 8.1.4.6 |
| CT-Verif Reduction | [Almeida16] | Section 8.1.4.7 |
| Capability Algebra | [CraryWalkerMorrisett99] | Section 7.1 |
| Realizability | [Patterson22] | Section 9.1 |
| SDG Two-Phase | [HorwitzRepsBinkley90] | Section 3.3 |
| Bi-Abduction | [Calcagno09] | Section 7.4 |
| Divergence Analysis | [Vanegue25] | Section 4.4 |
| Widening/Narrowing | [Cousot92] | Section 2.3 |
| Symbolic Execution | [King76] | Section 4.4 |
| Frame Rule | [Reynolds02] | Section 7.4 |
| Stack Filtering | [Li24] | Section 5.3.4 |

Table 71.2: Part XII Section Summary

| Section | Topic | Key Theorems |
|---------|--------------------------|--|
| 12.1 | Module Structure | F* module hierarchy |
| 12.2 | Soundness Theorems | AI, IFDS, Taint, Boundary, Ownership |
| 12.3 | Manifest/Latent | True Positives Property, Falsification |
| 12.4 | Under-Approximation | Reversed Consequence, Disjunction |
| 12.5 | Manifest Errors | Outcome Logic characterization |
| 12.6 | Capability Algebra | Free Requires Unique, Complete Collection |
| 12.7 | Realizability | Shared Memory Identity |
| 12.8 | Verified Interpreter | Specification-Implementation equivalence |
| 12.9 | SDG Theorems | Two-Phase slicing |
| 12.10 | Probabilistic | Scenario space, Law abstraction |
| 12.11 | Constraint Domains | DML(C) unification |
| 12.12 | Bi-Abduction | Compositional analysis |
| 12.13 | Divergence | Repeating states, Recursive divergence |
| 12.14 | Widening/Narrowing | Termination, Precision improvement |
| 12.15 | Symbolic Execution | Commutativity theorem |
| 12.16 | Effect Handlers | Algebraic effect correctness |
| 12.17 | Probabilistic Semantics | Linear operators on measures |
| 12.18 | Set Constraints | Resolution complexity |
| 12.19 | Local-Global Consistency | Fundamental AI theorem |
| 12.20 | Boundary Guards | Guard soundness |
| 12.21 | Frame Rule | Separation logic compositionality |
| 12.22 | Stack Filtering | Rupta precision theorem |
| 12.25 | Institution Theory | Satisfaction condition |
| 12.26 | C11 Memory Model | DRF-SC theorem |
| 12.27 | Effect Absence | Exception, Termination, State isolation |
| 12.28 | Session Types | Communication safety, Fidelity, Progress |
| 12.29 | SVF Theorems | Memory SSA, SVFG, Leak detection |
| 12.30 | ZIPPER | PFG captures flows, Precision preservation |
| 12.31 | Chopped SE | Soundness, Relative completeness |
| 12.32 | Python FTG | Type safety, Call graph soundness |
| 12.33 | DSA Theorems | Completeness, Soundness, Heap cloning |
| 12.34 | Robust Declassification | Robustness implies security, SNI, CT-Verif |

Part XIII

Implementation Roadmap and Engineering Specification

72. Executive Summary

This part provides a comprehensive implementation roadmap for the brrr-machine framework, translating the theoretical foundations established in Parts I–XII into concrete engineering deliverables. The implementation follows a phased approach with explicit dependencies, risk mitigations, and quality gates.

Implementation Scope

Primary Languages Rust (implementation), F* (verification)

Target Duration 6 phases, modular and parallelizable

Key Constraint Interleaved CG/PTS for OOP languages (Section ??)

Verification Target Mechanized proofs for core soundness theorems

73. Phase Architecture and Critical Dependencies

ARCHITECTURAL CONSTRAINT: INTERLEAVED CALL GRAPH + POINTER ANALYSIS

For object-oriented languages with virtual dispatch (Java, Python, JS, C++), call graph construction and points-to analysis exhibit **MUTUAL DEPENDENCY**:

- Call graph resolution requires points-to information for receivers
- Points-to propagation requires call graph edges for interprocedural flow

SOLUTION: Qilin-style on-the-fly algorithm (Section ??) computes both simultaneously via worklist iteration until mutual fixpoint.

For procedural languages (C, Fortran), sequential phasing is acceptable.

73.1 Implementation Phase Overview

The six phases of implementation proceed as follows, with dependencies indicated by vertical arrows:

| Phase 1: Foundation | Phase 2: CPG + Pointer Analysis |
|--|--|
| <ul style="list-style-type: none"> • Abstract domain typeclass hierarchy • Complete lattice with verified laws • Galois connection interface • IR type specification • Tree-sitter parser integration • AST + CFG construction | <ul style="list-style-type: none"> • On-the-fly CG + PTS (Qilin) • Virtual call resolution • PDG construction (post-CG) • Effect edge computation • Complete CPG with all edge types |
| Phase 3: Core Dataflow | Phase 4: Precision & Ownership |
| <ul style="list-style-type: none"> • IFDS tabulation algorithm • Reaching definitions analysis • Live variable analysis • Taint propagation (source→sink) • Nullability analysis | <ul style="list-style-type: none"> • k-CFA context sensitivity • Thin slicing (TAJ-style) • Ownership state machine • Resource lifecycle tracking • Bi-abduction for specs |
| Phase 5: Multi-Lang + Security + Concurrency | Phase 6: Production Hardening |
| <ul style="list-style-type: none"> • Cross-language boundary analysis • Matthews-Findler boundary semantics • DLM information flow control • Implicit flow tracking (PC labels) • Data race detection (happens-before) • Linearizability verification • Outcome Logic bug classification • Concolic witness generation | <ul style="list-style-type: none"> • Incremental re-analysis (DRedL) • Adaptive precision budgets • SARIF output generation • IDE/CI integration • Runtime debugger hooks • Performance optimization |

74. Phase 1: Foundation

74.1 Objective

Establish the core infrastructure upon which all subsequent analysis phases depend. This phase produces verified abstract domain implementations, the intermediate representation type system, and basic program graph construction.

74.2 Deliverables and Acceptance Criteria

74.2.1 Milestone 1.1: Abstract Domain Infrastructure

Deliverables:

- ☐ `PartialOrder` typeclass with reflexivity, antisymmetry, transitivity proofs
- ☐ `CompleteLattice` typeclass with join/meet/top/bottom verified laws
- ☐ `GaloisConnection` type with soundness lemma ($\gamma \circ \alpha \sqsubseteq \text{id}$)
- ☐ `AbstractDomain` typeclass with widening operator
- ☐ Chaotic iteration algorithm with Bourdoncle WTO

Acceptance Criteria:

- ✓ All algebraic laws verified in F* (no admits)
- ✓ Interval domain passes reference test suite

- ✓ Fixpoint computation terminates on all test cases

74.2.2 Milestone 1.2: Concrete Abstract Domains

Deliverables:

- Interval domain with widening thresholds $\{-\infty, -1, 0, 1, +\infty\}$
- Taint lattice: $\perp \sqsubset \text{Untainted} \sqsubset \text{Unknown} \sqsubset \text{Tainted} \sqsubset \top$
- Nullability lattice: $\perp \sqsubset \{\text{NonNull}, \text{Null}\} \sqsubset \top$
- Ownership state machine: $\text{Uninit} \rightarrow \text{Owned} \rightarrow \{\text{Moved}, \text{Borrowed}\} \rightarrow \text{Freed}$
- Effect row with row polymorphism support

Acceptance Criteria:

- ✓ Each domain implements `AbstractDomain` typeclass
- ✓ Galois connection soundness verified per domain
- ✓ Transfer functions monotone (verified)

74.2.3 Milestone 1.3: Code Property Graph Infrastructure

Deliverables:

- Node type hierarchy (Statement, Expression, Declaration, ...)
- Edge type hierarchy (AST, CFG, PDG_Data, PDG_Control, Call, Effect)
- CPG data structure with $O(1)$ node lookup, $O(\text{degree})$ edge traversal
- Traversal primitives: `successors`, `predecessors`, `reachable`, `filtered`
- Pattern matching DSL for node/edge queries

Acceptance Criteria:

- ✓ CPG construction from reference programs produces expected structure
- ✓ Traversal primitives pass property-based tests
- ✓ Memory usage $< 10\times$ source file size

74.2.4 Milestone 1.4: IR and Parser Integration

Deliverables:

- Complete Brrr-IR type specification (Section ??)
- Tree-sitter grammar bindings (Python, Rust, Go as initial targets)
- AST \rightarrow IR lowering transformations per language
- CFG construction with proper exception/return edge handling
- Basic CPG construction (AST nodes + CFG edges)

Acceptance Criteria:

- ✓ Round-trip: parse \rightarrow IR \rightarrow pretty-print preserves semantics
- ✓ CFG dominance frontier computation matches reference implementation
- ✓ All control flow constructs correctly modeled (loops, exceptions, etc.)

74.3 Risk Analysis and Mitigations

| Risk | Likelihood | Impact | Mitigation |
|---------------------------------------|------------|--------|--|
| F* proof complexity exceeds estimates | Medium | High | Start with key lemmas; defer non-critical proofs |
| Tree-sitter grammar edge cases | High | Medium | Comprehensive test suite; fall-back to partial parse |

| Risk | Likelihood | Impact | Mitigation |
|--|------------|--------|---|
| IR design inadequate for later phases | Medium | High | Design review with Phase 3–5 requirements |
| Performance bottleneck in CPG construction | Low | Medium | Profile early; use arena allocation |

75. Phase 2: CPG + Pointer Analysis

75.1 Objective

Construct complete Code Property Graphs with resolved call edges via interleaved call graph and points-to analysis. This phase addresses the mutual dependency between virtual dispatch resolution and pointer analysis through on-the-fly computation.

75.2 Deliverables and Acceptance Criteria

75.2.1 Milestone 2.1: On-the-fly Call Graph + Points-to (Qilin)

Deliverables:

- ☐ Worklist-based interleaved CG/PTS algorithm
- ☐ Points-to set representation (BDD or explicit set based on size)
- ☐ Virtual call resolution for each target language
- ☐ Heap abstraction via allocation-site or recency abstraction
- ☐ Context-insensitive baseline with selective refinement hooks

Acceptance Criteria:

- ✓ Virtual call resolution precision \geq CHA baseline
- ✓ Terminates on 100KLOC codebases within configurable timeout
- ✓ Points-to set queries complete in $O(1)$ amortized

75.2.2 Milestone 2.2: Language-Specific Dispatch Resolution

Deliverables:

- ☐ Java: vtable + interface dispatch resolution
- ☐ Python: MRO (C3 linearization) + `__getattr__` handling
- ☐ JavaScript: prototype chain traversal + Proxy handling
- ☐ C++: virtual table + multiple inheritance disambiguation
- ☐ Rust: trait object dispatch + monomorphization tracking

Acceptance Criteria:

- ✓ Dispatch resolution matches language specification semantics
- ✓ Dynamic dispatch edge marked with uncertainty when imprecise

75.2.3 Milestone 2.3: PDG Construction

Deliverables:

- ☐ Data dependence edges (def-use chains)
- ☐ Control dependence edges (via dominance frontier)
- ☐ Interprocedural summary edges for call sites
- ☐ Effect edges linking operations to their side effects

Acceptance Criteria:

- ✓ PDG slicing produces minimal relevant code for test queries
- ✓ Control dependence correctly handles structured exception handling

76. Phase 3: Core Dataflow

76.1 Objective

Implement the IFDS/IDE algorithmic framework for interprocedural dataflow analysis, along with foundational client analyses (reaching definitions, taint tracking, nullability).

76.2 Deliverables and Acceptance Criteria

76.2.1 Milestone 3.1: IFDS Tabulation Algorithm

Deliverables:

- ☐ Exploded supergraph construction from CPG + call graph
- ☐ Path edge computation via tabulation
- ☐ Summary edge caching for interprocedural reuse
- ☐ Demand-driven variant for interactive queries

Acceptance Criteria:

- ✓ Complexity $O(E \cdot D^3)$ verified on benchmark suite
- ✓ Summary edge reuse achieves $\geq 2\times$ speedup on repeated queries

76.2.2 Milestone 3.2: Taint Analysis

Deliverables:

- ☐ Source/sink/sanitizer specification language
- ☐ Taint propagation rules (assignment, call, return, field access)
- ☐ Context-sensitive taint tracking via IFDS
- ☐ Path reconstruction for vulnerability reports

Acceptance Criteria:

- ✓ Detects OWASP Top 10 injection vulnerabilities on test suite
- ✓ False positive rate $< 30\%$ on labeled benchmark

76.2.3 Milestone 3.3: Nullability Analysis

Deliverables:

- ☐ Null state lattice: `NonNull` | `Null` | `MaybeNull` | `Unknown`
- ☐ Null check recognition (`if x != null`, `x?.method`, etc.)
- ☐ Nullability inference for unannotated code
- ☐ Integration with type system annotations (`@Nullable`, `Option<T>`)

Acceptance Criteria:

- ✓ Correctly handles language-specific null semantics
- ✓ No false positives after explicit null checks

77. Phase 4: Precision and Ownership

77.1 Objective

Enhance analysis precision through context sensitivity, thin slicing, and ownership/resource lifecycle tracking. This phase bridges the gap between basic dataflow and production-quality bug detection.

77.2 Deliverables and Acceptance Criteria

77.2.1 Milestone 4.1: Context-Sensitive Analysis

Deliverables:

- ☐ k -CFA call-string context sensitivity (configurable k)
- ☐ Object sensitivity for OOP languages (1-object, 2-object)
- ☐ ZIPPER-guided selective context sensitivity
- ☐ Context abstraction for recursive call chains

Acceptance Criteria:

- ✓ Precision improvement $\geq 20\%$ over context-insensitive baseline
- ✓ Performance overhead $< 3\times$ for $k \leq 2$

77.2.2 Milestone 4.2: Thin Slicing (TAJ-Style)

Deliverables:

- ☐ Relevant dependency identification (producer statements only)
- ☐ Seed-based backward slicing from sinks
- ☐ Slice prioritization by security relevance

Acceptance Criteria:

- ✓ Slice size reduction $\geq 50\%$ vs traditional slicing
- ✓ No loss of true positives from slice reduction

77.2.3 Milestone 4.3: Ownership and Resource Tracking

Deliverables:

- ☐ Ownership state machine per abstract location
- ☐ Borrow tracking (shared/mutable, lifetime scope)
- ☐ Resource lifecycle analysis (file handles, connections, locks)
- ☐ Use-after-free, double-free, leak detection

Acceptance Criteria:

- ✓ Detects Rust-style ownership violations in non-Rust languages
- ✓ Resource leak detection with $< 20\%$ false positive rate

78. Phase 5: Multi-Language, Security, and Concurrency

78.1 Objective

Extend the analysis framework to handle cross-language boundaries, advanced security properties (information flow, implicit flows), and concurrent program verification (data races, linearizability).

78.2 Deliverables and Acceptance Criteria

78.2.1 Milestone 5.1: Cross-Language Boundary Analysis

Deliverables:

- ☐ Matthews-Finder boundary term representation
- ☐ Property preservation analysis at FFI boundaries
- ☐ Type compatibility verification (ABI layout matching)
- ☐ Risk scoring for cross-language calls

Acceptance Criteria:

- ✓ Detects type/layout mismatches at FFI boundaries

- ✓ Correctly propagates taint across language boundaries

78.2.2 Milestone 5.2: Advanced Information Flow Control

Deliverables:

- ☐ DLM (Decentralized Label Model) security labels
- ☐ PC (Program Counter) label tracking for implicit flows
- ☐ Declassification policy specification and verification
- ☐ 4-point security lattice (confidentiality \times integrity)

Acceptance Criteria:

- ✓ Detects implicit flows through control dependencies
- ✓ Supports principal-based access control policies

78.2.3 Milestone 5.3: Data Race Detection

Deliverables:

- ☐ Happens-before relation construction (fork/join, sync primitives)
- ☐ Lock set analysis for mutex-protected accesses
- ☐ Atomic operation modeling
- ☐ Race condition reporting with witness traces

Acceptance Criteria:

- ✓ Detects races on standard concurrency benchmarks
- ✓ Correctly handles language-specific synchronization primitives

78.2.4 Milestone 5.4: Outcome Logic Bug Classification

Deliverables:

- ☐ Manifest vs Latent bug classification [Le22]
- ☐ ISL triple representation for under-approximation
- ☐ Concolic witness generation for manifest bugs
- ☐ Context extraction for latent bugs

Acceptance Criteria:

- ✓ Manifest bugs have 0% false positive rate (by construction)
- ✓ Witness generation succeeds for $\geq 80\%$ of manifest classifications

79. Phase 6: Production Hardening

79.1 Objective

Prepare the framework for production deployment with incremental analysis, adaptive precision, standard output formats, and integration hooks for IDEs and CI/CD pipelines.

79.2 Deliverables and Acceptance Criteria

79.2.1 Milestone 6.1: Incremental Analysis (DRedL)

Deliverables:

- ☐ Dependency tracking at function/file/statement granularity
- ☐ Dirty-marking propagation on code changes
- ☐ DRedL-style lattice-based incremental Datalog evaluation
- ☐ Summary invalidation and re-computation

Acceptance Criteria:

- ✓ Re-analysis time $< 10\%$ of full analysis for single-file changes

- ✓ Incremental results equivalent to full re-analysis

79.2.2 Milestone 6.2: Adaptive Precision and Time Budgets

Deliverables:

- ☐ Per-function/file/project time budget configuration
- ☐ Graceful degradation levels (Full → Standard → Fast → Syntactic)
- ☐ Precision escalation for high-uncertainty findings
- ☐ Complexity estimation for budget allocation

Acceptance Criteria:

- ✓ Analysis completes within configured time budget
- ✓ Degradation preserves high-confidence findings

79.2.3 Milestone 6.3: Output and Integration

Deliverables:

- ☐ SARIF 2.1 output for IDE/CI integration
- ☐ LSP (Language Server Protocol) integration
- ☐ CLI with configurable verbosity and filtering
- ☐ Structured output for LLM consumption

Acceptance Criteria:

- ✓ SARIF output validates against schema
- ✓ IDE integration provides real-time feedback on file save

79.2.4 Milestone 6.4: Runtime Debugger Integration

Deliverables:

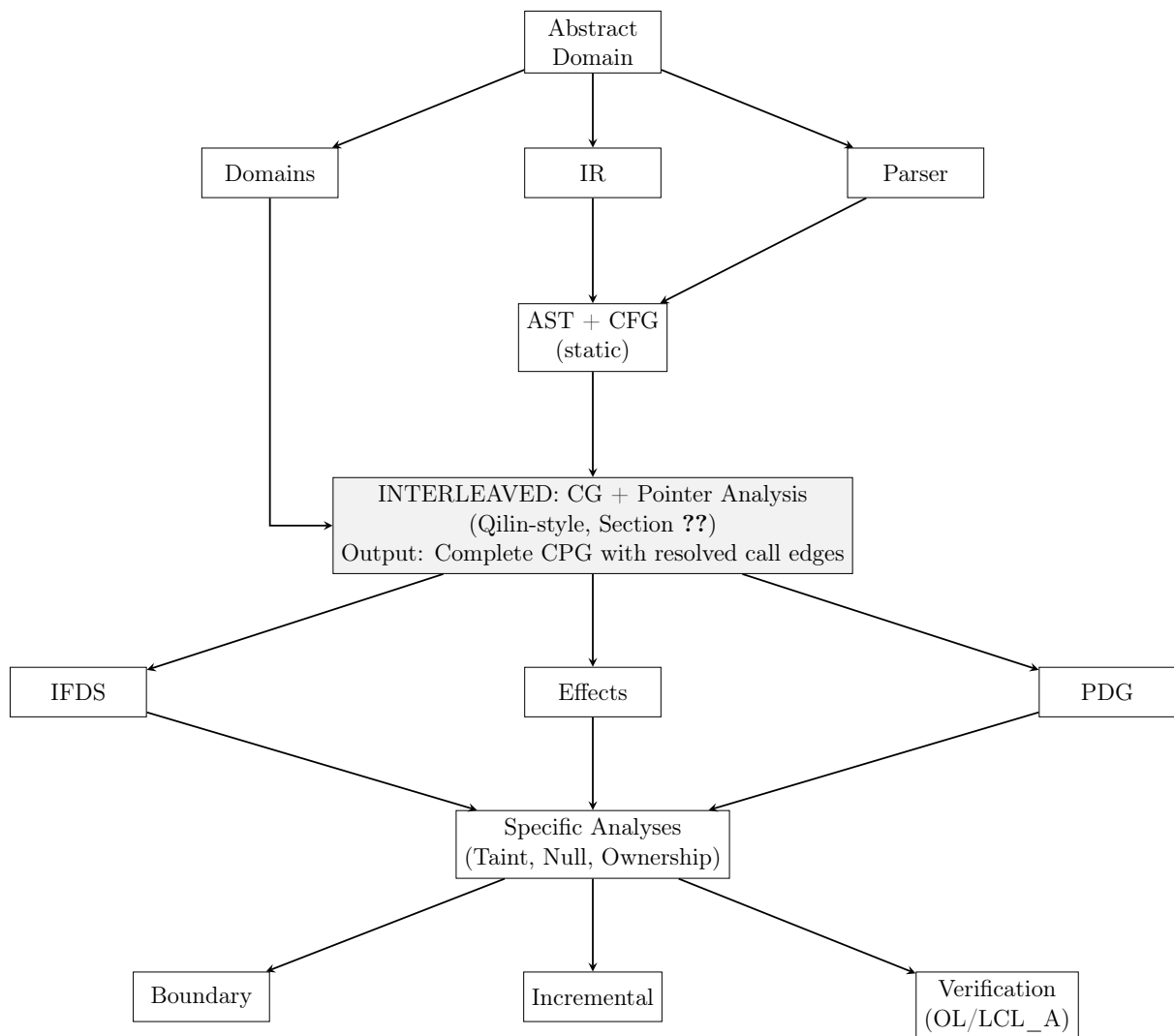
- ☐ Trace collection hooks for Python/Node/Rust/Go
- ☐ Static↔dynamic reconciliation engine
- ☐ Confidence boost/reduction based on runtime evidence
- ☐ Coverage-guided analysis prioritization

Acceptance Criteria:

- ✓ Runtime confirmation elevates finding confidence to ≥ 0.95
- ✓ Runtime contradiction marks finding as false positive

80. Dependency Graph

The following diagram illustrates the dependencies between major components. Note that the interleaved CG + Pointer Analysis (Qilin-style) receives input from both the parser infrastructure and the static AST/CFG construction.



81. Complete System Architecture

BRRR-MACHINE: Layered Architecture

A universal multi-language program analyzer with adaptive precision, parallelizable design, and best-effort F1 optimization.

81.1 Layer 0: Input Sources

The system accepts multiple input types:

| Source Code | Build Config | Test Suites | Runtime Traces |
|---|--|-------------------------------------|----------------|
| Python, Rust, Go, C, JS, TS, Java, ... | Cargo.toml tsconfig compile_cmds | pytest, go test, #[test], ... | Debugger hooks |

81.2 Layer 1: Parsing and IR

- **Tree-sitter** parsers (per language)
- **Brrr-Machine IR**: Unified SSA-like representation with:
 - Language-agnostic operations

- Explicit effects (Part ??)
- Type/ownership annotations

81.3 Layer 2: CPG Construction (Interleaved)

Critical Design Decision

For OOP languages, CG and PTS are computed **TOGETHER** (Qilin-style).

The interleaved worklist algorithm operates as follows:

Interleaved Worklist Algorithm (Pseudocode)

```
while worklist not empty:
    point ← pop(worklist)
    if is_virtual_call(point):
        for type T in PTS(receiver):          (* MUTUAL *)
            resolved ← resolve_method(T, method) (* DEPENDENCY *)
            add_cg_edge(point, resolved)
        update_pts(point)
    if pts_changed: add_dependents(worklist)
```

Algorithm Explanation: This pseudocode captures the core mutual dependency between call graph construction and points-to analysis. When encountering a virtual call site, the algorithm queries the current points-to set for the receiver to determine potential target methods. Each resolved method creates a new call graph edge, which in turn may introduce new points-to constraints. The algorithm iterates until reaching a fixpoint where neither the call graph nor points-to sets change. The mutual dependency (marked in comments) is the fundamental reason why these analyses cannot be performed sequentially for object-oriented languages with dynamic dispatch.

The output is a **Complete Code Property Graph** containing:

- AST nodes
- CFG edges
- CG edges (call graph)
- PDG edges (program dependence graph)
- Effect edges
- Points-To Solution (PTS)

81.4 Layer 3: Abstract Domains

| Taint | Nullability | Ownership | Effects |
|---------------------|---------------------|-------------------------|-------------|
| \perp_T | \perp_N | Uninit | {Read, |
| \uparrow | $\uparrow \uparrow$ | \downarrow | Write, |
| Untainted | NonNull Null | Owned | Alloc, |
| $\uparrow \uparrow$ | $\uparrow \uparrow$ | $\downarrow \downarrow$ | Free, |
| Maybe (!) | \top_N | Moved Borrowed | IO, |
| \uparrow | | \downarrow | Throw, ...} |
| Tainted | | Freed | |

Note: The *Maybe* element in the taint domain breaks Galois insertion, meaning this domain proves *incorrectness* only.

Additional domains include:

- **Security Labels:** DLM multi-principal + PC label for implicit flow
- **Access Permissions** (Bierhoff): unique/full/share/imm/pure
- **Shape (TVLA):** 3-valued logic for heap shapes

81.5 Layer 4: Analysis Algorithms

| IFDS | CFL-Reachability |
|--|---|
| $O(E \cdot D^3)$ interprocedural | Context sensitivity via matched parentheses (Dyck) |
| <ul style="list-style-type: none"> • Exploded supergraph • Path edges • Summary edges • Tabulation algorithm | <ul style="list-style-type: none"> • Demand-driven variant • On-the-fly computation |
| Thin Slicing | Fixpoint Iteration |
| Relevant dependencies only (not all reaching defs) | With widening/narrowing for infinite domains |

81.6 Layer 5: Specific Analyses

| Taint Analysis | Nullability Analysis | Ownership Tracking | Resource Leaks |
|---|------------------------------|-----------------------------|------------------|
| source \rightarrow sink with sanitizers | may-be-null dereference | use-after-move/free | unclosed handles |
| Implicit Flow | Boundary Risk | Data Races | |
| PC label tracking [Sabelfeld] | FFI/IPC type/memory mismatch | concurrent access detection | |

81.7 Layer 5.5: ML-Augmented Pattern Detection (Optional)

ML Integration Philosophy

Sources: [Pradel18] (DeepBugs), [Si18] (Code2Inv)

ML provides *STATISTICAL EVIDENCE*, not soundness guarantees. Use for prioritization and pattern detection, not proofs.

ML-based detectors include:

- **Swapped Arguments Detector:** e.g., `setTimeout(delay, fn)`
- **Wrong Operator Detector:** e.g., `x == y` vs `x != y`
- **Wrong Operand Detector:** e.g., `height + width` bug
- **Name Anomaly Detector:** unusual naming patterns

These feed into a **Semantic Embedding Service** (word2vec/code2vec).

Name-Based Security Role Classification:

- “sanitize”, “escape” \rightarrow LikelySanitizer
- “input”, “request” \rightarrow LikelySource
- “query”, “exec” \rightarrow LikelySink

This augments taint configuration with discovered sanitizers/sources/sinks.

81.8 Layer 6: Verification and Confidence

81.8.1 Confidence Inputs

All of the following feed into confidence computation:

| Static Results | Local Completeness (LCL) | Test/Code Discrepancy | Runtime Traces | ML Confidence |
|----------------|-----------------------------------|-----------------------|----------------|---------------|
| violations | $\alpha(f(c)) = f^\#(\alpha(c))?$ | test vs code | ground truth | DeepBugs |

Confidence Combination ([Pradel18], [QSYM18])

- Static + ML agree: **BOOST** confidence (ensemble effect)
- Static only: Use static confidence
- ML only: Report with ML confidence (no soundness)
- ML provides prioritization, static provides soundness

81.8.2 Outcome Logic Classification

For each violation:

| Manifest Error [Le22] | Latent Error |
|--|--|
| ISL proof has presumption ($\text{emp} \wedge \text{true}$) — no caller constraints needed | Requires specific precondition from caller |
| Bug happens regardless of calling context → HIGH confidence | Bug happens only in specific contexts → Report with context |

Falsification (OL Theorem 5.1): If spec violated, we can prove it.

81.8.3 Final Confidence Levels

| Classification | Description | Action |
|----------------|---------------------------------|----------------|
| ConfirmedBug | Manifest + concolic witness | Report: HIGH |
| HighConfBug | Manifest, no witness yet | Report: HIGH |
| TrueAlarm | Violation + locally complete | Report: MEDIUM |
| ConditionalBug | Latent with plausible context | Report: MEDIUM |
| PossibleFP | Violation but LC failed | Investigate |
| ConfirmedSafe | No violation + locally complete | Skip |
| LikelySafe | No violation, LC unknown | Skip |

81.9 Layer 7: Output

| SARIF (tooling) | CLI (human) | LLM (consumable) |
|----------------------------|----------------------------------|----------------------------------|
| Standard format for IDE/CI | Prioritized findings with traces | Structured for agent consumption |

81.10 Cross-Cutting Concerns

| Incrementality (Part ??) | Multi-Language (Part IX) |
|---|--|
| <ul style="list-style-type: none"> • Adapton-style dirty marking • File/function/statement level • Dependency tracking | <ul style="list-style-type: none"> • Matthews boundary semantics • Risk analysis at FFI/IPC • Type consistency checking |
| Time Budgets (D.9) | Parallelization |
| <ul style="list-style-type: none"> • Per-function/file/total • Graceful degradation levels • Adaptive precision | <ul style="list-style-type: none"> • Laptop → datacenter scalable • Independent analyses parallel • Worklist can be parallelized |
| Configurability | Uncertainty Propagation |
| <ul style="list-style-type: none"> • Everything is a spectrum • User can tune all parameters • Fork-friendly extensibility | <ul style="list-style-type: none"> • Each block outputs confidence • Propagates through pipeline • Guides adaptive precision |
| ML Integration (Layer 5.5) | Hybrid Testing (Section ??) |
| <ul style="list-style-type: none"> • DeepBugs name-based detection • Semantic identifier embeddings • Augments static with ML conf • Section ?? for details | <ul style="list-style-type: none"> • QSYM: concolic + fuzzer cooperation • Validation-based soundness • 10–100× faster than pure concolic • Native execution + selective instr |

82. Mutually Exclusive Analysis Paths

Valid Parallel Options

Multiple implementations producing compatible output.

The brrr-machine design allows **SWAPPABLE** components at key decision points. Each path trades precision for speed. Output types remain compatible.

82.1 Pointer Analysis (Layer 2)

Note: Choice is LANGUAGE-DEPENDENT (Part V).

| Option A Andersen | Option B Steensgaard | Option C Rupta | Option D DSA | Option E Qilin+ZIPPER |
|--|---|---|---|--|
| $O(n^3)$ subset-based Flow-sensitive possible | $O(n \cdot \alpha)$ unification Flow-insensitive | 1-callsite + stack filter For Rust: FASTER + precise | $O(n \cdot \alpha)$ + heap cloning Large C/C++ (>100K LOC) | On-the-fly CG + PTS OOP with virtual dispatch |
| Higher precision (<50K) | Fastest, least precise | Section ?? | Section 12.2 | Section ?? |

Language Guide:

- C/C++ <50K: Andersen
- C/C++ >100K: DSA
- Rust: Rupta
- OOP: Qilin

Unified Output: $\text{pts} : \text{var_id} \rightarrow \text{set abstract_loc}$

82.2 Dataflow Algorithm (Layer 4)

Note: IFDS requires DISTRIBUTIVE functions.

| Option A: IFDS | Option B: Symbolic Exec | Option C: Set Constraints |
|--|---|---|
| $O(ED^3)$ path-insensitive DISTRIBUTIVE only! Taint, reaching defs | Path-sensitive Input-specific Precise witnesses | Section ?? [Aiken] Handles NON-distributive Pointer, type inference |

Unified Output: $\text{finding} : \{\text{location}, \text{kind}, \text{confidence}\}$

82.3 Heap Abstraction (Layer 3)

| Option A: TVLA/Shape | Option B: Allocation-Site | Unified Output |
|--|---|--|
| 3-valued logic Precise shapes Very expensive | Simple abstraction Fast, imprecise Good enough for most | $\text{heap_shape} : \text{abstract_heap}$ |

82.4 Precision Mode (Cross-cutting)

| Option A: Full Precision | Option B: Degraded/-Fast | Unified Output |
|---|---|--|
| Andersen + IFDS + shapes High confidence Slow, thorough | Steensgaard + patterns Lower confidence Fast, approximate | Same output types, different confidence |

82.5 Bug Verification (Layer 6)

| Option A: OL Proofs | Option B: Concolic Exec | Unified Output |
|---|--|--|
| Symbolic bug classif. Manifest/Latent Sound (when succeeds) | Concrete witnesses Actual inputs Complete (when finds) | $\text{bug_classification}$ + optional witness |

82.6 Adaptive Selection Strategy

The brrr-machine framework uses a configurable analysis pipeline where different algorithms can be selected based on language characteristics, codebase size, and time constraints. The following F*-style type definitions formalize the configuration space:

Analysis Configuration Types

```
(* Pointer algorithm selection based on language and size *)
type PointerAlgorithm =
  | Andersen      (* C/C++ < 50K LOC, most precise *)
  | Steensgaard   (* Quick pre-analysis, least precise *)
  | DSA           (* C/C++ > 100K LOC, heap cloning *)
  | Qilin         (* OOP with virtual dispatch *)
  | Rupta         (* Rust with stack filtering *)

type DataflowAlgorithm = | IFDS | Symbolic | Hybrid
type HeapAbstraction = | TVLA | AllocationSite | Recency
type PrecisionMode = | Full | Standard | Fast | Syntactic
type VerificationMode = | OLOnly | ConcolicOnly | Both

(* Main configuration record combining all choices *)
type analysis_config = {
  pointer_analysis : PointerAlgorithm;
  dataflow_algo : DataflowAlgorithm;
  heap_abstraction : HeapAbstraction;
  precision_mode : PrecisionMode;
  verification : VerificationMode;
}
```

Type Signature Explanation: The `analysis_config` record bundles five orthogonal configuration choices. The `PointerAlgorithm` type captures the trade-off between precision (Andersen) and scalability (Steensgaard, DSA), with language-specific options (Qilin for OOP, Rupta for Rust). The `VerificationMode` determines whether bugs are classified using Outcome Logic proofs, concolic witness generation, or both approaches in combination.

The following function demonstrates how the framework automatically selects an appropriate configuration based on the input code property graph and available time budget:

Adaptive Configuration Selection

```
(* Type signature: takes a CPG and time budget, returns configuration *)
(* This enables adaptive precision based on code characteristics *)
val select_config : cpg → time_budget → analysis_config
let select_config cpg budget =
  let complexity = estimate_complexity cpg in
  let loc_count = count_loc cpg in
  let lang = primary_language cpg in
  let has_concurrency = uses_threads cpg in
  match budget, complexity, lang with
  | _, _, Rust →
    (* Rust: always use Rupta for stack filtering *)
    { pointer_analysis = Rupta;
      dataflow_algo = IFDS;
      heap_abstraction = AllocationSite;
      precision_mode = Standard;
      verification = Both }
  | Unlimited, _, C | Unlimited, _, Cpp when loc_count < 50000 →
    (* Small C/C++: Andersen for max precision *)
    { pointer_analysis = Andersen;
      dataflow_algo = IFDS;
      heap_abstraction = TVLA;
      precision_mode = Full;
      verification = Both }
  | _, _, C | _, _, Cpp when loc_count > 100000 →
    (* Large C/C++: DSA for scalability with heap cloning *)
    { pointer_analysis = DSA;
      dataflow_algo = IFDS;
      heap_abstraction = AllocationSite;
      precision_mode = Standard;
      verification = OLOnly }
```

```

| _, _, Java | _, _, Python →
  (* OOP: Qilin for virtual dispatch *)
  { pointer_analysis = Qilin;
    dataflow_algo = IFDS;
    heap_abstraction = AllocationSite;
    precision_mode = Standard;
    verification = Both }
| Limited t, High, _ when t < minutes 5 →
  { pointer_analysis = Steensgaard;
    dataflow_algo = IFDS;
    heap_abstraction = AllocationSite;
    precision_mode = Fast;
    verification = OLOnly }
| _ →
  { pointer_analysis = Andersen;
    dataflow_algo = Hybrid;
    heap_abstraction = AllocationSite;
    precision_mode = Standard;
    verification = Both }

```

Selection Logic: The function uses pattern matching on language type, codebase size, and time budget to select the most appropriate algorithm combination. Key heuristics include: (1) Rust always uses Rupta for stack filtering benefits; (2) small C/C++ codebases (<50K LOC) use Andersen for maximum precision; (3) large C/C++ codebases (>100K LOC) use DSA for scalability; (4) OOP languages (Java, Python) use Qilin for on-the-fly virtual dispatch resolution; (5) tight time budgets trigger graceful degradation to faster, less precise algorithms.

82.7 Output Compatibility Guarantee

All paths produce compatible output types. This enables:

1. **Progressive refinement** — Start fast, refine suspicious findings
2. **Parallel execution** — Run multiple paths, merge results
3. **User choice** — Configure based on needs (CI vs IDE vs audit)
4. **Fallback** — Degrade gracefully when precise analysis times out

The unified finding type ensures that different analysis paths produce compatible outputs that can be merged and compared:

Unified Finding Type and Merging

```

(* Core finding type: all analysis paths produce this structure *)
(* Key fields: location, kind identify the finding; confidence enables ranking *)
type unified_finding = {
  location : node_id;           (* Where in the CPG *)
  kind : vulnerability_type;    (* SQL injection, XSS, null deref, etc. *)
  confidence : confidence_level; (* Certainty from 0.0 to 1.0 *)
  path_used : analysis_path;    (* Which algorithm produced this *)
  witness : option concrete_trace; (* Concrete input triggering the bug *)
  can_refine : bool;           (* Is higher-precision analysis available? *)
}

(* Merge findings from multiple analysis paths *)
(* Groups by (location, kind), keeps highest-confidence finding per group *)
val merge_findings : list unified_finding → list unified_finding
let merge_findings findings =
  let grouped = group_by (fun f → (f.location, f.kind)) findings in
  List.map (fun fs →
    let best = max_by (fun f → confidence_to_nat f.confidence) fs in
    { best with can_refine = List.length fs > 1 }
  ) grouped

```

Merging Strategy: When multiple analysis paths report findings at the same location with the same vulnerability kind, the `merge_findings` function retains the finding with the highest

confidence score. The `can_refine` flag is set to `true` when multiple paths contributed findings, indicating that the location was flagged by independent analyses—increasing overall confidence in the result. This enables the ensemble effect described in Layer 6 (Verification and Confidence).

Part XIV

Channel Analysis

Foundation: [Honda08] “Multipart Asynchronous Session Types”, [Honda98] “Language Primitives and Type Discipline for Structured Communication”

This part provides the theoretical foundations for analyzing communication channels in concurrent and distributed systems. Channel analysis extends the core session type theory (Section ??) with detailed syntax, semantics, and causality analysis for multiparty protocols.

83. Binary Session Types

Paper: [Honda98] “Language Primitives and Type Discipline for Structured Communication-Based Programming”

Binary session types provide the foundational theory for structured communication between exactly two parties. This precedes and motivates the multiparty extensions in subsequent sections.

83.1 The Session Concept

Core Insight (Honda 1998)

A **session** is a chain of dyadic (two-party) reciprocal interactions serving as a unit of abstraction for describing communication behavior.

Motivation: Traditional primitives (RPC, method invocation, rendez-vous) express only one-time interactions. Complex protocols require sequences of related communications, but prior languages had no construct to structure them.

Analogy to Structured Programming:

- Imperative (pre-1970s): assignment + goto → spaghetti code
- Imperative (post-1970s): assignment + if/while → structured programs
- Concurrent (pre-1998): send/receive + parallel → unstructured interaction
- Concurrent (post-1998): session primitives + types → structured communication

Session Structure:

- Sessions designated by **channels** (distinct from names/ports)
- Channel k is private to session, generated fresh at session initiation
- All interactions in session occur through its channel
- Session combines: value passing, label branching, delegation

83.2 Binary Session Type Syntax

Definition 83.1 (Binary Session Type Grammar (Honda 1998, Definition 5.1)). **Sorts** (S):

$$S ::= \text{nat} \mid \text{bool} \mid \langle \alpha, \bar{\alpha} \rangle$$

Types (α, β):

$$\begin{array}{ll}
 \alpha ::= \downarrow [\tilde{S}]; \alpha & \text{(Input: receive values of sorts } \tilde{S} \text{)} \\
 \quad \mid \downarrow [\alpha]; \beta & \text{(Channel input: delegation receive)} \\
 \quad \mid \&\{l_1 : \alpha_1, \dots, l_n : \alpha_n\} & \text{(Branching: offer } n \text{ labeled options)} \\
 \quad \mid \mathbf{1} & \text{(Termination/inaction)} \\
 \quad \mid \perp & \text{(No further connection possible)} \\
 \quad \mid \uparrow [\tilde{S}]; \alpha & \text{(Output: send values of sorts } \tilde{S} \text{)} \\
 \quad \mid \uparrow [\alpha]; \beta & \text{(Channel output: delegation send)} \\
 \quad \mid \oplus \{l_1 : \alpha_1, \dots, l_n : \alpha_n\} & \text{(Selection: choose one of } n \text{ labels)} \\
 \quad \mid t & \text{(Type variable)} \\
 \quad \mid \mu t. \alpha & \text{(Recursive type)}
 \end{array}$$

Type Interpretation:

- $\downarrow [S]; \alpha$: First receive value of sort S , then behave as α
- $\uparrow [S]; \alpha$: First send value of sort S , then behave as α
- $\&\{l_i : \alpha_i\}$: **External choice** — wait for partner to select label
- $\oplus\{l_i : \alpha_i\}$: **Internal choice** — select label to send to partner
- $\mu t. \alpha$: Recursive behavior (loops, unbounded interactions)
- $\mathbf{1}$: Session complete, no more actions
- \perp : Channel consumed/hidden, no reconnection

Example Types:

- **Remote Procedure Call:**
 - Caller: $\uparrow [\text{int}]; \downarrow [\text{int}]$ (send argument, receive result)
 - Callee: $\downarrow [\text{int}]; \uparrow [\text{int}]$ (receive argument, send result)
- **Cell Object:**
 - Cell: $\&\{\text{read} : \uparrow [\text{int}], \text{write} : \downarrow [\text{int}]\}$
 - Client: $\oplus\{\text{read} : \downarrow [\text{int}], \text{write} : \uparrow [\text{int}]\}$
- **ATM Protocol (unbounded):**

$$\text{ATM} : \downarrow [\text{nat}]; \mu t. \& \left\{ \begin{array}{l} \text{deposit} : \downarrow [\text{nat}]; t, \\ \text{withdraw} : \downarrow [\text{nat}]; \oplus\{\text{success} : \uparrow [\text{nat}]; t, \text{failure} : t\}, \\ \text{balance} : \uparrow [\text{nat}]; t, \\ \text{quit} : \mathbf{1} \end{array} \right\}$$

83.3 The Duality Principle

Definition 83.2 (Duality (Co-Type)). For type α where \perp does not occur, the co-type $\bar{\alpha}$ is defined:

$$\begin{array}{ll} \overline{\uparrow [S]; \alpha} = \downarrow [S]; \bar{\alpha} & (\text{send dual is receive}) \\ \overline{\downarrow [S]; \alpha} = \uparrow [S]; \bar{\alpha} & (\text{receive dual is send}) \\ \overline{\oplus\{l_i : \alpha_i\}} = \&\{l_i : \bar{\alpha}_i\} & (\text{selection dual is branching}) \\ \overline{\&\{l_i : \alpha_i\}} = \oplus\{l_i : \bar{\alpha}_i\} & (\text{branching dual is selection}) \\ \overline{\uparrow [\alpha]; \beta} = \downarrow [\alpha]; \bar{\beta} & (\text{delegation duals}) \\ \overline{\downarrow [\alpha]; \beta} = \uparrow [\alpha]; \bar{\beta} & \\ \overline{\mu t. \alpha} = \mu t. \bar{\alpha} & (\text{recursion preserves duality}) \\ \bar{\bar{t}} = t & (\text{variable unchanged}) \\ \bar{\mathbf{1}} = \mathbf{1} & (\text{termination self-dual}) \end{array}$$

Fundamental Property: $\bar{\bar{\alpha}} = \alpha$ (involution: dualizing twice returns original)

Significance: If two processes communicate on channel k with types α and $\bar{\alpha}$, their interaction patterns are **compatible** — no type errors occur.

Examples:

- Type: $\uparrow [\text{int}]; \downarrow [\text{bool}]$, Dual: $\downarrow [\text{int}]; \uparrow [\text{bool}]$
- Type: $\oplus\{\text{ok} : \uparrow [\text{int}], \text{err} : \mathbf{1}\}$, Dual: $\&\{\text{ok} : \downarrow [\text{int}], \text{err} : \mathbf{1}\}$

83.4 Type Algebra for Composition

Definition 83.3 (Typing Context Composition (Honda 1998, Definition 5.2)). The type algebra governs how typings combine under parallel composition.

Compatibility ($\Delta_1 \diamond \Delta_2$): Two typings are compatible if common channels have **dual** types:

$$\Delta_1 \diamond \Delta_2 \iff \forall k \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2). \Delta_1(k) = \overline{\Delta_2(k)}$$

Composition ($\Delta_1 \circ \Delta_2$): When $\Delta_1 \diamond \Delta_2$, the composition is:

$$(\Delta_1 \circ \Delta_2)(k) = \begin{cases} \perp & \text{if } k \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) \\ \Delta_i(k) & \text{if } k \in \text{dom}(\Delta_i) \setminus \text{dom}(\Delta_{\text{other}}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Why \perp After Composition: When two processes with dual types compose on channel k , no **third** process can join that channel — it's fully consumed. This prevents interference and ensures session isolation.

Typing Rules (key rules):

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \quad \Theta; \Gamma \vdash Q \triangleright \Delta' \quad \Delta \diamond \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \text{CONC} \quad \frac{\Gamma \vdash \tilde{e} : \tilde{S} \quad \Theta; \Gamma \vdash P \triangleright \Delta, k : \alpha}{\Theta; \Gamma \vdash k![\tilde{e}]; P \triangleright \Delta, k : \uparrow[\tilde{S}]; \alpha} \text{SEND}$$

$$\frac{\Theta; \Gamma, \tilde{x} : \tilde{S} \vdash P \triangleright \Delta, k : \alpha}{\Theta; \Gamma \vdash k?(\tilde{x}) \text{ in } P \triangleright \Delta, k : \downarrow[\tilde{S}]; \alpha} \text{RCV}$$

$$\frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta, k : \alpha_1 \quad \dots \quad \Theta; \Gamma \vdash P_n \triangleright \Delta, k : \alpha_n}{\Theta; \Gamma \vdash k \triangleleft \{l_1 : P_1 \square \dots \square l_n : P_n\} \triangleright \Delta, k : \&\{\alpha_1, \dots, \alpha_n\}} \text{BR}$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta, k : \alpha_j \quad 1 \leq j \leq n}{\Theta; \Gamma \vdash k \triangleright l_j; P \triangleright \Delta, k : \oplus\{\alpha_1, \dots, \alpha_n\}} \text{SEL}$$

83.5 Session Initiation and Delegation

Session Initiation:

request $a(k)$ **in** P Request new session via name a , bind channel k
accept $a(k)$ **in** P Accept session request via a , bind channel k

Operational Semantics:

$$[\text{Link}] \quad (\text{accept } a(k) \text{ in } P_1) \mid (\text{request } a(k) \text{ in } P_2) \rightarrow (\nu k)(P_1 \mid P_2)$$

Fresh channel k is generated, private to the session. P_1 and P_2 proceed with complementary views of the protocol.

Delegation (Channel Passing):

throw $k[k']$; P Send channel k' through channel k
catch $k(k')$ **in** P Receive channel through k , bind as k'

Purpose: Delegation allows **dynamic redistribution** of session participation. An ongoing session can be handed off to another process.

Typing Rules:

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta, k : \beta}{\Theta; \Gamma \vdash \text{throw } k[k']; P \triangleright \Delta, k : \uparrow[\alpha]; \beta, k' : \alpha} \text{THR} \quad \frac{\Theta; \Gamma \vdash P \triangleright \Delta, k : \beta, k' : \alpha}{\Theta; \Gamma \vdash \text{catch } k(k') \text{ in } P \triangleright \Delta, k : \downarrow[\alpha]; \beta} \text{CAT}$$

Example (FTP Server with Delegation): The following code illustrates how delegation enables dynamic session redistribution. The FTP server accepts client connections on a public name, then immediately delegates the session to a worker thread, freeing the main loop to continue accepting new connections. This pattern is fundamental to scalable server design.

FTP Delegation Pattern

```
(* FTP server that delegates sessions to worker threads *)
Ftpd(pid, b) = accept pid(s) in      (* Accept client connection *)
               request b(k) in      (* Get worker thread *)
```

```

        throw k[s];          (* Delegate session to worker *)
        Ftpd[pid, b]         (* Continue accepting *)

(* The client sees a single session; delegation is transparent *)

```

Key Type Signature: The `throw k[s]` operation has type $k : \uparrow [T]; \beta$ where T is the session type being delegated. The worker receives the session with type $k : \downarrow [T]; \beta'$ and continues the protocol from where it was delegated.

83.6 Safety Theorems

Theorem 83.4 (Safety (Honda 1998, Theorem 5.4)). *1. Invariance under Structural Equivalence:* $\Theta; \Gamma \vdash P \triangleright \Delta$ and $P \equiv Q$ implies $\Theta; \Gamma \vdash Q \triangleright \Delta$

2. Subject Reduction: $\Theta; \Gamma \vdash P \triangleright \Delta$ and $P \rightarrow^* Q$ implies $\Theta; \Gamma \vdash Q \triangleright \Delta$

3. Error Freedom: A typable program **never** reduces to an error state.

Definition 83.5 (Error States). P is an error if $P \equiv \mathbf{def} \ D \ \mathbf{in} \ (\nu \tilde{u})(P' \mid R)$ where P' contains, for some channel k , either:

- Two k -processes that don't form a k -redex (mismatched operations)
- Three or more k -processes (race condition)

What This Guarantees:

- No message type mismatches (send int, expect bool)
- No unhandled selections (select label not in branch)
- No races on channels (exactly two parties per channel)
- Sessions complete or progress (no stuck states except inaction)

Connection to Multiparty: Binary session types ensure 2-party safety. Multiparty session types [Honda08] generalize to n parties via global types and projection, with similar safety guarantees.

83.7 Relationship to Go Channels and Rust mpsc

Mapping to Practical Languages

Go Channels: Go channels are **untyped at the session level** — only element type is tracked. Go provides **no static guarantee** of protocol compliance.

| Go Pattern | Session Type Equivalent |
|-----------------------------------|---------------------------|
| <code>ch := make(chan int)</code> | Create session channel |
| <code>ch <- 42</code> | $\uparrow [\text{int}]$ |
| <code>x := <-ch</code> | $\downarrow [\text{int}]$ |
| <code>close(ch)</code> | End session |

Session types **add:** sequence constraints, branching/selection typing.

Rust mpsc: Rust's `std::sync::mpsc` provides typed channels but not session typing. The `session_types` crate adds Honda-style session types to Rust.

| Rust Pattern | Session Type Equivalent |
|---------------------------------------|------------------------------------|
| <code>let (tx, rx) = channel()</code> | Create session with dual endpoints |
| <code>tx.send(42)</code> | $\uparrow [\text{int}]$ |
| <code>let x = rx.recv()</code> | $\downarrow [\text{int}]$ |
| <code>drop(tx), drop(rx)</code> | Session end (via affine types) |

Session types **add:** protocol state machine, compile-time progress checks.

Static Analysis Application: For Go/Rust channel analysis, we can:

1. **Infer** session types from usage patterns
2. **Check** that inferred types are dual at connection points
3. **Detect** protocol violations (missing receives, wrong order)
4. **Verify** deadlock freedom via causality analysis (Section ??)

84. Global Type Syntax and Semantics

Global types describe conversation protocols from a bird's-eye view, specifying the complete interaction structure among all participants.

84.1 Global Type Grammar

Definition 84.1 (Global Type Grammar).

| | |
|---|--|
| $G ::= p \rightarrow q : k \langle U \rangle . G$ | (Message passing: p sends U to q via channel k) |
| $ p \rightarrow q : k \{l_j : G_j\}_{j \in J}$ | (Labeled choice: p selects label l_j , q branches) |
| $ G_1 \mid G_2$ | (Parallel composition) |
| $ \mu t . G$ | (Recursive type) |
| $ t$ | (Type variable) |
| $ \text{end}$ | (Termination) |

Payload Types (U):

| | |
|-------------------|-------------------------------------|
| $U ::= \tilde{S}$ | (Sequence of base sorts) |
| $ T @ p$ | (Located local type for delegation) |

Base Sorts (S):

| | |
|---|---|
| $S ::= \text{bool} \mid \text{nat} \mid \text{int} \mid \text{string} \mid \dots$ | |
| $ \langle \alpha, \bar{\alpha} \rangle$ | (Session sort: pair of dual types) |
| $ G$ | (Global type as sort for nested sessions) |

84.2 Prefix Ordering

The prefix ordering (\prec) defines the causal structure of global types. A prefix a precedes prefix b (written $a \prec b$) when a must complete before b can execute.

Definition 84.2 (Prefix Ordering Rules). **Sequential Composition:**

$$\frac{G = G_1 . G_2}{\forall a \in \text{prefixes}(G_1), b \in \text{prefixes}(G_2). a \prec b}$$

Parallel Independence:

$$\frac{G = G_1 \mid G_2}{\neg(a \prec b) \wedge \neg(b \prec a) \text{ for } a \in G_1, b \in G_2} \quad (\text{unless connected by causality edges})$$

Branch Ordering:

$$\frac{G = p \rightarrow q : k \{l_j : G_j\}_{j \in J}}{\text{branch_prefix} \prec \forall a \in \bigcup_j \text{prefixes}(G_j)}$$

Recursion Unfolding:

$$\frac{G = \mu t . G'}{\text{ordering}(G) = \text{ordering}(G'[\mu t . G' / t])}$$

84.3 Equi-Recursive Type Handling

Global types use an equi-recursive interpretation where $\mu t. G \equiv G[\mu t. G/t]$. This enables infinite unfolding for ongoing protocols.

Definition 84.3 (Equi-Recursive Equivalence). • **Unfolding:** $\mu t. G \equiv G[\mu t. G/t]$

- **Contractivity:** In $\mu t. G$, the variable t must be guarded by at least one message prefix (ensures productive recursion)
- **Regular Trees:** Global types denote regular infinite trees (finitely representable infinite structures)

Example — Streaming Protocol:

$\mu t. \text{Producer} \rightarrow \text{Consumer} : \text{data}\langle \text{int} \rangle. \text{Consumer} \rightarrow \text{Producer} : \text{ack}\langle \text{bool} \rangle. t$

Unfolds to infinite alternation of data/ack exchanges.

85. Causality Analysis

Causality analysis determines which prefixes must complete before others can execute. This is fundamental for ensuring progress and detecting deadlocks.

85.1 Input-Input Dependency

Definition 85.1 (Input-Input Dependency (\prec^{II})). $a \prec^{II} b$ holds when:

- Both a and b are input prefixes at the same participant p
- a must be executed before b according to the global type structure
- They share the same session (belong to same global type G)

Formally:

$$\frac{\text{input_at}(G, p, k_1) \quad \text{input_at}(G, p, k_2) \quad \text{precedes_in}(G, k_1, k_2)}{k_1 \prec_p^{II} k_2}$$

Example: $G = A \rightarrow B : s_1\langle \text{int} \rangle. A \rightarrow B : s_2\langle \text{int} \rangle. \text{end}$

At participant B : $s_1? \prec_B^{II} s_2?$ (B must receive on s_1 before s_2)

Chain Property: If $k_1 \prec^{II} k_2 \prec^{II} \dots \prec^{II} k_n$, then B executes inputs in order k_1, k_2, \dots, k_n .

85.2 Input-Output Dependency

Definition 85.2 (Input-Output Dependency (\prec^{IO})). $a \prec^{IO} b$ holds when:

- a is an input prefix and b is an output prefix at the same participant p
- The output depends on data received by the input
- a must complete before b can execute

Formally:

$$\frac{\text{input_at}(G, p, k_{\text{in}}) \quad \text{output_at}(G, p, k_{\text{out}}) \quad \text{data_flows}(G, k_{\text{in}}, k_{\text{out}})}{k_{\text{in}} \prec_p^{IO} k_{\text{out}}}$$

Example:

$G = \text{Client} \rightarrow \text{Server} : \text{req}\langle \text{Query} \rangle. \text{Server} \rightarrow \text{Client} : \text{resp}\langle \text{Result} \rangle. \text{end}$

At Server: $\text{req}? \prec_{\text{Server}}^{IO} \text{resp}!$ (Server must receive query before sending response)

Importance: Captures request-response patterns, essential for detecting causal violations, ensures data availability before output.

85.3 Output-Output Dependency

Definition 85.3 (Output-Output Dependency (\prec^{OO})). $a \prec^{OO} b$ holds when:

- Both a and b are output prefixes at the same participant p
- **Synchronous communication:** a must complete (including ack) before b
- **Asynchronous communication:** ordering imposed by channel or explicit sequencing

Synchronous Case:

$$\frac{\text{output_at}(G, p, k_1) \quad \text{output_at}(G, p, k_2) \quad \text{sync_mode} \quad \text{seq}(G, k_1, k_2)}{k_1 \prec_p^{OO} k_2}$$

Asynchronous Case: \prec^{OO} is **only** induced by:

1. Same channel (FIFO ordering): $s_1!$ before $s_2!$ on same s
2. Explicit control flow in process

Note: This is a **key difference** between sync and async session types. Sync has more causality (OO-edges between any sequential outputs). Async has less causality (more parallelism possible).

Examples:

- **Sync:** $G = A \rightarrow B : s(\text{int}). A \rightarrow C : t(\text{int}). \text{end}$ — At A (sync): $s! \prec_A^{OO} t!$
- **Async:** Same G , but at A (async): $s!$ and $t!$ can proceed in parallel

85.4 Dependency Chain Analysis

Definition 85.4 (Dependency Chains). **Input Dependency Chain:** A sequence k_1, k_2, \dots, k_n forms an input chain at p if:

$$\forall i \in [1, n-1]. k_i \prec_p^{II} k_{i+1}$$

Property: Participant p must execute inputs in exactly this order.

Output Dependency Chain: A sequence k_1, k_2, \dots, k_n forms an output chain at p if:

$$\forall i \in [1, n-1]. (k_i \prec_p^{OO} k_{i+1}) \vee (k_i \prec_p^{IO} k_{i+1})$$

Property: Participant p 's outputs are ordered by this chain.

Mixed Dependency Graph: Combine all three relations into directed graph:

- Nodes: all prefixes in G
- Edges: \prec^{II} , \prec^{IO} , \prec^{OO} edges

Acyclicity: The dependency graph **must** be acyclic for deadlock freedom.

Deadlock Detection: Cycle in dependency graph \Rightarrow potential deadlock.

Example deadlock:

$$\begin{aligned} A : s_1?; s_2! & \quad (\text{input before output}) \\ B : s_2?; s_1! & \quad (\text{input before output}) \end{aligned}$$

Creates cycle: $s_1? \xrightarrow{IO} s_2! \rightarrow s_2? \xrightarrow{IO} s_1! \rightarrow s_1?$

86. Linearity and Coherence

86.1 Linear Global Types

Definition 86.1 (Linearity (Definition 3.5 from Honda 2008)). A global type G is **linear** if every prefix in G :

1. Is not suppressed by \prec^{II} (no conflicting input ordering)
2. Is not suppressed by \prec^{IO} (no conflicting input-output ordering)

3. Is not suppressed by \prec^{OO} (no conflicting output ordering)

Suppression by \prec^{II} : Prefix k is suppressed if $\exists k'. k' \prec^{II} k \wedge k' \prec^{II} k$ (same predecessor leads to different paths). Intuition: Two inputs at same participant cannot both be “first”.

Suppression by \prec^{IO} : Prefix k (output) is suppressed if $\exists k'. k' \prec^{IO} k \wedge \neg \text{enabled}(k')$ (required input not yet available).

Suppression by \prec^{OO} : Prefix k is suppressed if $\exists k'. k' \prec^{OO} k \wedge \neg \text{completed}(k')$ (required prior output not done).

The linearity check algorithm verifies that a global type satisfies the suppression-freedom conditions required for well-formed protocols. The algorithm traverses the global type structure, checking at each prefix whether any of the three dependency relations (\prec^{II} , \prec^{IO} , \prec^{OO}) would suppress execution.

Linearity Check Algorithm

```

let rec linear (g : global_type) : bool =
  match g with
  | GMsg p q k u g' →
    not (suppressed_II g k) &&
    not (suppressed_IO g k) &&
    not (suppressed_OO g k) &&
    linear g'

  | GBranch p q k branches →
    not (suppressed_II g k) &&
    not (suppressed_IO g k) &&
    not (suppressed_OO g k) &&
    List.for_all (fun (_, gj) → linear gj) branches

  | GPar g1 g2 →
    linear g1 && linear g2 && no_conflicts g1 g2

  | GRec _ g' → linear g'
  | GVar _ | GEnd → true

```

Complexity: The algorithm runs in $O(|G|^2)$ time, where $|G|$ is the size of the global type. The quadratic factor arises from checking suppression conditions for each prefix pair. For practical protocols (typically $|G| < 100$ prefixes), this is fast.

86.2 Coherence Condition

Definition 86.2 (Coherence (Definition 4.2 from Honda 2008)). A family of local types $\{T_p @ p\}_{p \in P}$ is **coherent** if:

1. There exists a linear global type G
2. For each participant $p \in P$: $T_p = G \upharpoonright p$ (projection of G onto p)

Formally:

$$\text{coherent}(\{T_p @ p\}_{p \in P}) \iff \exists G. \text{linear}(G) \wedge \forall p \in P. T_p = \text{project}(G, p)$$

Coherence Ensures:

1. All participants agree on the protocol structure
2. Sends match receives (no lost messages)
3. Selections match branches (no unhandled cases)
4. Delegations are properly received

Coherence Examples:

- **Coherent:** $\{k \oplus \{\text{ok} : \text{end}, \text{quit} : \text{end}\} @ A, k \& \{\text{ok} : \text{end}, \text{quit} : \text{end}\} @ B\}$ (A’s selection matches B’s branches)
- **Coherent:** $\{k! \langle \text{int} \rangle @ A, k? \langle \text{int} \rangle @ B\}$ (A’s send matches B’s receive)

- **Not Coherent** (from Honda 2008 Figure 5): $\{s!@A, s?; s?@B, s!@C\}$ — No linear global type generates these projections

86.3 Projection Algorithm

Definition 86.3 (Projection Algorithm (Definition 4.1 from Honda 2008)). The projection $G \restriction p$ extracts participant p 's local view:

The projection algorithm transforms a global type into a local type for a specific participant. This is the core operation that enables decentralized implementation of multiparty protocols: each participant only needs to know their local view, not the full global type.

Projection Algorithm

```

let rec project (g : global_type) (p : participant) : local_type option =
  match g with
  | GMsg q r k u g' →
    if p = q then Some (LSend k u (project g' p))
    else if p = r then Some (LRecv k u (project g' p))
    else project g' p  (* p not involved *)

  | GBranch q r k branches →
    if p = q then
      Some (LSelect k (List.map (fun (l, gj) → (l, project gj p)) branches))
    else if p = r then
      Some (LBranch k (List.map (fun (l, gj) → (l, project gj p)) branches))
    else
      (* p not involved: all branches must project identically *)
      let projs = List.map (fun (_, gj) → project gj p) branches in
      if all_equal projs then List.hd projs
      else None  (* UNDEFINED *)

  | GPar g1 g2 →
    merge (project g1 p) (project g2 p)

  | GRec v g' → Some (LRec v (project g' p))
  | GVar v → Some (LVar v)
  | GEnd → Some LEnd

```

Projection Properties:

- **Totality for Linear Types:** If G is linear and $p \in \text{participants}(G)$, then $\text{project}(G, p)$ is defined.
- **Projection Preserves Typing:** If $\Gamma \vdash P \triangleright \Delta$ and $\Delta(s) = \llbracket G \rrbracket$ then $\text{project}(G, p)$ types P 's behavior at s for p .
- **Merge Operation:** For parallel composition, merge combines independent local behaviors.

87. Local Type Syntax

87.1 Local Type Grammar

Definition 87.1 (Local Type Grammar).

| | |
|--------------------------------------|---|
| $T ::= k!\langle U \rangle; T$ | (Output: send payload U via channel k) |
| $ k?\langle U \rangle; T$ | (Input: receive payload U via channel k) |
| $ k \oplus \{l_j : T_j\}_{j \in J}$ | (Selection: choose one label to send) |
| $ k \& \{l_j : T_j\}_{j \in J}$ | (Branching: offer labels, receive choice) |
| $ k!\langle T' @ p \rangle; T$ | (Delegation: send session capability) |
| $ k?\langle T' @ p \rangle; T$ | (Session receive: get delegated capability) |
| $ \mu t. T$ | (Recursive type) |
| $ t$ | (Type variable) |
| $ \mathbf{end}$ | (Termination) |

where $k \in \mathbb{N}$ (channel number), $U ::= \tilde{S} \mid T @ p$ (payload), $l \in \text{Labels}$, $p \in \text{Participants}$.

87.2 Duality (Co-Type) Relation

Definition 87.2 (Duality for Local Types). For binary sessions, each type has a dual representing the partner's view:

| | |
|--|----------------------------|
| $\overline{k!\langle U \rangle; T} = k?\langle U \rangle; \bar{T}$ | (send dual is receive) |
| $\overline{k?\langle U \rangle; T} = k!\langle U \rangle; \bar{T}$ | (receive dual is send) |
| $\overline{k \oplus \{l_j : T_j\}} = k \& \{l_j : \bar{T}_j\}$ | (selection dual is branch) |
| $\overline{k \& \{l_j : T_j\}} = k \oplus \{l_j : \bar{T}_j\}$ | (branch dual is selection) |
| $\overline{k!\langle T' @ p \rangle; T} = k?\langle T' @ p \rangle; \bar{T}$ | (delegation dual) |
| $\overline{k?\langle T' @ p \rangle; T} = k!\langle T' @ p \rangle; \bar{T}$ | (session receive dual) |
| $\overline{\mu t. T} = \mu t. \bar{T}$ | (unfold before dualizing) |
| $\bar{\bar{t}} = t$ | (variable unchanged) |
| $\overline{\mathbf{end}} = \mathbf{end}$ | (termination self-dual) |

Duality Properties:

- **Involution:** $\bar{\bar{T}} = T$ (dualizing twice returns original)
- **Compatibility:** If two processes have types T and \bar{T} at a shared channel, their interactions will be compatible (no type errors)
- **Session Safety:** Well-typed binary session with dual endpoints never fails

87.3 Type Isomorphism Rules

Definition 87.3 (Type Isomorphism (\approx)). **Permutation of Independent Outputs:**

$$k_1!\langle U_1 \rangle; k_2!\langle U_2 \rangle; T \approx k_2!\langle U_2 \rangle; k_1!\langle U_1 \rangle; T \quad (\text{when } k_1 \neq k_2)$$

Rationale: Outputs on different channels can be reordered since they are causally independent in async communication.

Permutation of Independent Inputs:

$$k_1?\langle U_1 \rangle; k_2?\langle U_2 \rangle; T \approx k_2?\langle U_2 \rangle; k_1?\langle U_1 \rangle; T \quad (\text{when } k_1 \neq k_2)$$

Rationale: Inputs on different channels can be reordered when no data dependency exists.

Associativity: $(T_1; T_2); T_3 \approx T_1; (T_2; T_3)$

Identity: $\text{end}; T \approx T \approx T; \text{end}$

Isomorphism Preserves Typing: If $T_1 \approx T_2$ and $\Gamma \vdash P \triangleright \Delta, s : T_1$ then $\Gamma \vdash P \triangleright \Delta, s : T_2$.

87.4 Subtyping Relation

Definition 87.4 (Session Subtyping (Gay & Hole 2005)). The subtyping relation \leq_{sub} allows safe substitution of more specific types.

Output (Covariant in payload):

$$\frac{U_1 \leq U_2 \quad T_1 \leq_{\text{sub}} T_2}{k!\langle U_1 \rangle; T_1 \leq_{\text{sub}} k!\langle U_2 \rangle; T_2}$$

Input (Contravariant in payload):

$$\frac{U_2 \leq U_1 \quad T_1 \leq_{\text{sub}} T_2}{k?\langle U_1 \rangle; T_1 \leq_{\text{sub}} k?\langle U_2 \rangle; T_2}$$

Selection (Covariant in labels — fewer choices OK):

$$\frac{I \subseteq J \quad \forall i \in I. T_i \leq_{\text{sub}} T'_i}{k \oplus \{l_i : T_i\}_{i \in I} \leq_{\text{sub}} k \oplus \{l_j : T'_j\}_{j \in J}}$$

Branching (Contravariant in labels — more cases OK):

$$\frac{J \subseteq I \quad \forall j \in J. T_j \leq_{\text{sub}} T'_j}{k \& \{l_i : T_i\}_{i \in I} \leq_{\text{sub}} k \& \{l_j : T'_j\}_{j \in J}}$$

Subtyping Examples:

- $k \oplus \{\text{ok} : T_1\} \leq_{\text{sub}} k \oplus \{\text{ok} : T_1, \text{quit} : T_2\}$ (Can select fewer options than offered)
- $k \& \{\text{ok} : T_1, \text{quit} : T_2\} \leq_{\text{sub}} k \& \{\text{ok} : T_1\}$ (Can handle more cases than required)

88. Runtime Typing

Runtime typing extends the static type system to handle message queues, which appear during execution of asynchronous communication.

88.1 Type Contexts for Message Queues

Definition 88.1 (Type Context Grammar).

$$\begin{aligned} T[\cdot] ::= & [\cdot] && \text{(Hole where local type goes)} \\ & | k!\langle U \rangle; T[\cdot] && \text{(Output prefix context)} \\ & | k \oplus l : T[\cdot] && \text{(Selection prefix context, singleton)} \end{aligned}$$

Context Operations:

- **Hole Filling:** $T[T']$ plugs local type T' into context $T[\cdot]$
- **Context Composition:** $T_1[\cdot] \circ T_2[\cdot] = T_1[T_2[\cdot]]$ (sequential composition)

Queue Typing Intuition: A message queue contains outputs that have been sent but not yet received. The type context “remembers” these pending outputs.

Example: Queue: $s : \langle 3 \rangle$ (value 3 waiting), Context: $1!\langle \text{nat} \rangle; [\cdot]$, Combined: $1!\langle \text{nat} \rangle; T$ (rollback: pretend output hasn’t happened).

This “rollback” technique lets us type runtime states using static types.

88.2 Extended Typing Judgment

Definition 88.2 (Runtime Typing Judgment).

$$\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$$

The subscript \tilde{s} tracks which session channels have associated queues.

Meaning: Under environment Γ , process P with queues for channels \tilde{s} has typing Δ .

Key Difference from Static Typing:

- Static: $\Gamma \vdash P \triangleright \Delta$ (no queues)
- Runtime: $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$ (queues for \tilde{s})

Coherence of \tilde{s} : For well-typed runtime states:

- Each channel in \tilde{s} has exactly one queue
- Queue contents match the type context
- No dangling or duplicated queues

88.3 Queue Typing Rules

$$\begin{array}{c}
\frac{\Delta \text{ contains only } \mathbf{end}}{\Gamma \vdash (s_k : \varepsilon) \triangleright_{s_k} \tilde{s} : \{[\cdot]@p\}_p \circ \Delta} \text{QNIL} \\
\\
\frac{\Gamma \vdash v_i : S_i \quad \Gamma \vdash (s_k : \tilde{h}) \triangleright_{s_k} \Delta, \tilde{s} : \{T@q\} \cup R \quad R = \{H_p@p\}_{p \in I}}{\Gamma \vdash (s_k : \tilde{h} \cdot \tilde{v}) \triangleright_{s_k} \Delta, \tilde{s} : (T[k!\langle \tilde{S} \rangle; [\cdot]]@q) \cup R} \text{QVAL} \\
\\
\frac{\Gamma \vdash (s_k : \tilde{h}) \triangleright_{s_k} \Delta, \tilde{s} : \{T@q\} \cup R \quad R = \{H_p@p\}_{p \in I}}{\Gamma \vdash (s_k : \tilde{h} \cdot \tilde{t}) \triangleright_{s_k} \Delta, \tilde{s} : (T[k!\langle T'@p' \rangle; [\cdot]]@q) \cup R, \tilde{t} : T'@p'} \text{QSESS} \\
\\
\frac{\Gamma \vdash (s_k : \tilde{h}) \triangleright_{s_k} \Delta, \tilde{s} : \{T@q\} \cup R \quad R = \{H_p@p\}_{p \in I}}{\Gamma \vdash (s_k : \tilde{h} \cdot l) \triangleright_{s_k} \Delta, \tilde{s} : (T[k \oplus l : [\cdot]]@q) \cup R} \text{QSEL} \\
\\
\frac{\Gamma \vdash P \triangleright_{\tilde{t}_1} \Delta \quad \Gamma \vdash Q \triangleright_{\tilde{t}_2} \Delta' \quad \tilde{t}_1 \cap \tilde{t}_2 = \emptyset \quad \Delta \bowtie \Delta'}{\Gamma \vdash_{\tilde{t}_1 \cdot \tilde{t}_2} P \mid Q \triangleright_{\tilde{t}_1 \cdot \tilde{t}_2} \Delta \circ \Delta'} \text{CONC} \\
\\
\frac{\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s} : \{T_p@p\}_{p \in I} \quad \tilde{s} \in \tilde{t} \quad \{T_p@p\}_{p \in I} \text{ coherent}}{\Gamma \vdash_{\tilde{t} \cdot \tilde{s}} (\nu \tilde{s})P \triangleright \Delta} \text{CRES}
\end{array}$$

88.4 Runtime Typing Properties

Proposition 88.3 (Static/Runtime Equivalence (Proposition 5.1)). *For program phrases (no queues):*

$$\Gamma \vdash P \triangleright \Delta \iff \Gamma \vdash P \triangleright_{\emptyset} \Delta \quad (\text{without } [Subs])$$

Proposition 88.4 (Queue Uniqueness (Proposition 5.2)). *If $\Gamma \vdash P \triangleright_{s_1 \dots s_m} \Delta$ then:*

1. P has exactly one queue at each s_i ($1 \leq i \leq m$)
2. No other queues occur free in P
3. No queue is under any prefix

Rollback Principle: Runtime typing “rolls back” queue contents into types.

- Runtime state: Process $P \mid \text{Queue}(s, [v_1, v_2])$
- After rollback: Type includes $k!\langle S_1 \rangle; k!\langle S_2 \rangle; [\text{actual local type}]$

This makes subject reduction work: the queue’s effect is captured in types.

Type Safety for Runtime: Well-typed runtime states (including queues) reduce to well-typed states. Combined with communication safety, this ensures no message type mismatches, no lost messages, and no unhandled selections.

89. Integration with Static Analysis

89.1 Channel Analysis in CPG

Channel Analysis CPG Extension

Add to Code Property Graph:

1. **Session nodes:** represent session channels
2. **Protocol edges:** connect session nodes per global type
3. **Causality edges:** encode \prec^{II} , \prec^{IO} , \prec^{OO} relations
4. **Projection nodes:** local type for each participant

CPG Node Types:

- `SessionNode(id, global_type)`
- `ParticipantNode(id, participant, local_type)`
- `PrefixNode(id, kind, channel, payload)`

CPG Edge Types:

- `SessionOf(prefix_node, session_node)`
- `ParticipantOf(participant_node, session_node)`
- `Causality(prefix_node, prefix_node, kind)` — II, IO, or OO
- `Next(prefix_node, prefix_node)` — sequential composition

Analysis Queries:

Deadlock Detection:

```
MATCH path = (p1:PrefixNode)-[:Causality*]->(p1)
RETURN path // cycle indicates potential deadlock
```

Protocol Violation:

```
MATCH (p:ParticipantNode)-[:LocalType]->(t:Type)
MATCH (proc:Process)-[:Implements]->(p)
WHERE NOT conforms(proc, t)
RETURN proc, t // process doesn't follow local type
```

Orphan Message:

```
MATCH (send:PrefixNode {kind: 'output'})-[:SessionOf]->(s:Session)
WHERE NOT exists((recv:PrefixNode {kind: 'input'})-[:Matches]->(send))
RETURN send // output with no matching input
```

89.2 Cross-Reference with Synthesis Sections

Channel Analysis: Cross-References

Section 5.3 (CPG Construction):

- Session nodes added as new vertex type
- Causality edges extend data dependency edges
- Protocol structure becomes queryable via graph traversal

Section 6.4 (IFDS/IDE):

- Session type checking as IFDS problem
- Facts: (node, type_state) pairs
- Edges: type transitions at prefix nodes

Section 8.1 (Taint Analysis):

- Session channels as additional taint propagation paths
- Delegation transfers taint along with capability
- Protocol-aware sanitization (type-directed)

Part IX (Multi-Language):

- Global types describe cross-language protocols
- FFI boundaries require session type consistency
- Projection per language gives local implementation spec

Part XI (Uncertainty):

- Incomplete session specifications → uncertainty
- Missing participants → coherence unknown
- Dynamic session creation → parametric analysis

89.3 Language-Specific Channel Mappings

This section defines how channel operations in concrete languages map to the unified IR channel primitives (Part ??).

89.3.1 Go Channel Mappings

| Go Syntax | IR Statement |
|------------------------------------|---|
| <code>ch := make(chan T)</code> | <code>SChanCreate(<i>ch</i>, <i>T</i>, 0)</code> — Unbuffered (synchronous) |
| <code>ch := make(chan T, n)</code> | <code>SChanCreate(<i>ch</i>, <i>T</i>, <i>n</i>)</code> — Buffered with capacity <i>n</i> |
| <code>ch <- v</code> | <code>SSend(<i>ch</i>, <i>v</i>)</code> — Send (may block) |
| <code>v := <-ch</code> | <code>SRecv(<i>v</i>, <i>ch</i>)</code> — Receive (may block) |
| <code>v, ok := <-ch</code> | <code>SRecvOk(<i>v</i>, <i>ok</i>, <i>ch</i>)</code> — With closed-channel check |
| <code>close(ch)</code> | <code>SChanClose(<i>ch</i>)</code> — Close channel |
| <code>select {...}</code> | <code>SSelect([...])</code> — Non-deterministic choice |

Go Type Mappings:

| Go Type | IR Type |
|--------------------------|--|
| <code>chan T</code> | <code>TChannel(<i>T</i>, Bidirectional)</code> |
| <code>chan<- T</code> | <code>TChannel(<i>T</i>, SendOnly)</code> |
| <code><-chan T</code> | <code>TChannel(<i>T</i>, RecvOnly)</code> |

89.3.2 Rust Channel Mappings

| Rust Syntax (std::sync::mpsc) | IR Statement |
|---|--|
| <code>let (tx, rx) = mpsc::channel()</code> | <code>SChanCreate(<i>ch</i>, <i>T</i>, UNBOUNDED)</code> |
| <code>let (tx, rx) = mpsc::sync_channel(n)</code> | <code>SChanCreate(<i>ch</i>, <i>T</i>, <i>n</i>)</code> |
| <code>tx.send(v).unwrap()</code> | <code>SSend(<i>tx</i>, <i>v</i>)</code> |
| <code>let v = rx.recv().unwrap()</code> | <code>SRecv(<i>v</i>, <i>rx</i>)</code> |

| Rust Syntax (<code>std::sync::mpsc</code>) | IR Statement |
|--|---|
| <code>let v = rx.try_recv()</code> | <code>SRecvTry(v, rx)</code> |
| <code>drop(tx)</code> | <code>SChanClose(tx)</code> — Implicit close |

Rust Type Mappings:

| Rust Type | IR Type |
|---|---|
| <code>Sender<T></code> | <code>TChannel(T, SendOnly)</code> |
| <code>Receiver<T></code> | <code>TChannel(T, RecvOnly)</code> |
| <code>(Sender<T>, Receiver<T>)</code> | <code>TChannelPair(T)</code> |

Ownership Considerations:

- `Sender<T>`: Can be `Clone`'d (multiple producers)
- `Receiver<T>`: Cannot be `Clone`'d (single consumer)
- Channel ownership: Sender owns send capability, Receiver owns recv capability
- Linearity: Each endpoint used linearly (no aliasing of recv end)

89.3.3 Python Channel Mappings

| Python Syntax (<code>asyncio.Queue</code>) | IR Statement |
|--|---|
| <code>q = asyncio.Queue()</code> | <code>SChanCreate(q, Any, UNBOUNDED)</code> |
| <code>q = asyncio.Queue(maxsize=n)</code> | <code>SChanCreate(q, Any, n)</code> |
| <code>await q.put(v)</code> | <code>SSend(q, v)</code> with EAsync |
| <code>v = await q.get()</code> | <code>SRecv(v, q)</code> with EAsync |
| <code>await q.put(None)</code> | <code>SSend($q, None$)</code> — Convention for “close” |

Taint Considerations:

- Pickled data through `multiprocessing.Queue`: taint propagates with serialization
- `asyncio.Queue`: taint propagates within same process
- Cross-process channels: require taint boundary analysis

89.3.4 TypeScript/JavaScript Channel Mappings

Note: JS/TS don't have built-in channels, but common patterns include async iterators/generators, RxJS Observables, and Web Workers with `postMessage`.

| TypeScript Syntax (RxJS) | IR Statement |
|---|--|
| <code>const subject = new Subject<T>()</code> | <code>SChanCreate($ch, T, 0$)</code> — Unbuffered |
| <code>const subject = new ReplaySubject(n)</code> | <code>SChanCreate(ch, T, n)</code> — Buffered |
| <code>subject.next(v)</code> | <code>SSend(ch, v)</code> |
| <code>subject.subscribe(v => body)</code> | <code>SRecv(v, ch)</code> in async loop |
| <code>subject.complete()</code> | <code>SChanClose(ch)</code> |

89.3.5 Session Type Annotations

For languages supporting annotations/attributes, session types can be specified:

Go (via comments or build tags):

```
// @session: !string; ?int; &{ok: !string, quit: end}
func BuyerProtocol(ch chan interface{}) { ... }
```

Rust (via attributes):

```
#[session_type("!String; ?i32; +{ok: !String, quit: end}")]
async fn buyer_protocol(tx: Sender<...>, rx: Receiver<...>) { ... }
```

TypeScript (via decorators or JSDoc):

```
/** @session !string; ?number; &{ok: !string, quit: end} */
async function buyerProtocol(ch: Channel<...>): Promise<void> { ... }
```

Python (via type hints or decorators):

```
@session_type("!str; ?int; &{ok: !str, quit: end}")
async def buyer_protocol(ch: Channel) -> None: ...
```

These annotations enable: (1) Static protocol conformance checking, (2) Runtime protocol monitoring, (3) Documentation generation, (4) Cross-language protocol compatibility verification.

89.4 Channel Analysis in Existing Analyses

The following code extends the IFDS dataflow framework (Section ??) to handle channel operations. Each channel operation creates or transforms dataflow facts that propagate through the interprocedural analysis. This enables tracking tainted data as it flows through channels, monitoring session type state at each program point, and verifying ownership invariants.

Channel Operations as IFDS Facts — Section 5.3 Extension

```
(* Channel operations create dataflow facts that propagate through IFDS.
   This enables tracking of:
   - Tainted data through channels
   - Session type state at each program point
   - Channel aliasing and ownership *)

(* IFDS fact domain extended for channels *)
type channel_ifds_fact =
| ChanTainted : chan_id:nat → source:taint_source → channel_ifds_fact
  (* Channel carries tainted data from source *)
| ChanTypeState : chan_id:nat → state:local_session_type → channel_ifds_fact
  (* Channel is at session type state *)
| ChanOwned : chan_id:nat → owner:var_id → channel_ifds_fact
  (* Channel endpoint owned by variable *)
| ChanClosed : chan_id:nat → channel_ifds_fact
  (* Channel has been closed *)
| ChanDelegated : chan_id:nat → to_chan:nat → channel_ifds_fact
  (* Channel capability was delegated through to_chan *)

(* Transfer function for channel operations *)
val channel_transfer : ir_stmt → set channel_ifds_fact → set channel_ifds_fact
let channel_transfer stmt facts =
  match stmt with
  | SChanCreate dst elem_ty buf_sz →
    (* New channel: add ownership fact *)
    let chan_id = fresh_chan_id () in
```

```

    Set.add (ChanOwned chan_id dst) facts

| SSend ch v →
    (* Send: propagate taint from value to channel *)
    let chan_id = resolve_chan ch in
    let v_taint = get_taint v facts in
    Set.union facts (Set.map (fun t → ChanTainted chan_id t) v_taint)

| SRecv dst ch →
    (* Receive: propagate taint from channel to destination *)
    let chan_id = resolve_chan ch in
    let chan_taint = Set.filter (fun f →
        match f with ChanTainted cid _ → cid = chan_id | _ → false) facts in
    let dst_taint = Set.map (fun (ChanTainted _ src) → VarTainted dst src) chan_taint in
    Set.union facts dst_taint

| SChanClose ch →
    (* Close: mark channel as closed *)
    let chan_id = resolve_chan ch in
    Set.add (ChanClosed chan_id) facts

| SChanDelegate ch delegated →
    (* Delegate: transfer ownership, track delegation *)
    let chan_id = resolve_chan ch in
    let del_id = resolve_chan delegated in
    Set.add (ChanDelegated del_id chan_id) facts

| _ → facts (* Non-channel statements don't affect channel facts *)

```

Key Type Signatures: The `channel_ifds_fact` type captures the different kinds of facts that can be tracked for channels. `ChanTainted` records that a channel carries tainted data, `ChanTypeState` tracks the current session type state, and `ChanOwned`/`ChanClosed`/`ChanDelegated` track ownership transitions.

The following code extends taint analysis (Section ??) to handle channel-based communication. When tainted data is sent through a channel, the channel itself becomes tainted, and any values received from that channel inherit the taint. This models information flow through concurrent communication paths.

Taint Propagation Through Channels — Section 8.2 Extension

```

(* Tainted data sent through a channel taints all received values.
    This models information flow through concurrent communication. *)

(* Taint flow rules for channels *)
type channel_taint_rule =
| SendTaintsChannel : tainted_value → channel → channel_taint_rule
    (* Sending tainted value taints the channel *)
| RecvInheritsTaint : channel → recv_variable → channel_taint_rule
    (* Receiving from tainted channel taints the variable *)
| DelegationTransfersTaint : delegated_chan → via_chan → channel_taint_rule
    (* Delegation transfers taint along with capability *)
| ClosedChannelNoTaint : channel → channel_taint_rule
    (* Closed channels don't propagate new taint *)

val apply_channel_taint : taint_state → ir_stmt → taint_state
let apply_channel_taint state stmt =
    match stmt with
    | SSend ch v when is_tainted state v →
        (* Taint propagates: value → channel *)
        add_channel_taint state ch (get_taint_sources state v)

    | SRecv dst ch when is_channel_tainted state ch →
        (* Taint propagates: channel → variable *)

```

```

    add_var_taint state dst (get_channel_taint_sources state ch)

| SChanDelegate ch delegated when is_channel_tainted state delegated →
  (* Taint propagates with delegation *)
  add_channel_taint state ch (get_channel_taint_sources state delegated)

| _ → state

```

Security Implication: The rule `DelegationTransfersTaint` is particularly important—when a session capability is delegated, any taint associated with that session transfers with it. This prevents information laundering through delegation.

The following code extends ownership analysis (Section ??) to handle channel endpoints. Channel endpoints in languages like Rust have explicit ownership semantics: sender endpoints can be cloned (shared ownership via reference counting), while receiver endpoints are unique (linear). This distinction is critical for detecting use-after-close bugs and ensuring proper resource cleanup.

Ownership Tracking for Channels — Section 7.1 Extension

```

(* Channel endpoints have ownership semantics:
- Sender endpoints: can be cloned (shared ownership)
- Receiver endpoints: unique ownership (linear)
- Closing: requires ownership of endpoint *)

(* Ownership states for channel endpoints (extends Section 7.1.1 Camera) *)
type channel_ownership =
| OwnedSender : refcount:nat → channel_ownership
  (* Sender can be cloned; tracks reference count *)
| OwnedReceiver : channel_ownership
  (* Receiver is unique (linear) *)
| Borrowed : from:var_id → channel_ownership
  (* Temporarily borrowed *)
| Moved : channel_ownership
  (* Ownership has been transferred *)

(* Channel ownership checking *)
val check_channel_ownership : ownership_context → ir_stmt → option ownership_error
let check_channel_ownership ctx stmt =
  match stmt with
  | SSend ch _ →
    (* Requires ownership or borrow of sender *)
    if not (has_send_capability ctx ch) then
      Some (UseAfterMove ch)
    else None

  | SRecv _ ch →
    (* Requires ownership or borrow of receiver *)
    if not (has_recv_capability ctx ch) then
      Some (UseAfterMove ch)
    else None

  | SChanClose ch →
    (* Requires ownership (not just borrow) *)
    if not (owns_endpoint ctx ch) then
      Some (InvalidClose ch)
    else None

  | SChanDelegate _ delegated →
    (* Moves ownership of delegated channel *)
    if not (owns_endpoint ctx delegated) then
      Some (CannotDelegateUnowned delegated)
    else None

```

```

| _ → None

(* Integration with Iris cameras (Section 7.1.1) *)
val channel_camera : channel_ownership → camera_element
let channel_camera own =
  match own with
  | OwnedSender n → Frac (1.0 /. float_of_int n) (* Fractional for senders *)
  | OwnedReceiver → Exclusive () (* Exclusive for receiver *)
  | Borrowed _ → Frac 0.5 (* Temporary borrow *)
  | Moved → Invalid (* No ownership *)

```

90. Theoretical Reconciliation: Session Types and Outcome Logic

This chapter resolves fundamental theoretical tensions between classical session type theory [Honda98], [Honda08] and the under-approximation framework of Outcome Logic. These tensions arise because session types provide **sound** guarantees (may reject valid programs) while Outcome Logic targets bug finding via **under-approximation**. We provide formal resolutions that enable both frameworks to coexist productively.

90.1 Tension 1: Over-Approximation vs Under-Approximation

The Fundamental Tension

Session Types (Honda 1998, 2008):

- **Sound** type system: well-typed programs **never** violate protocols
- **Over-approximation**: may **reject** valid programs (false negatives for bugs)
- Guarantees: communication safety, session fidelity, progress

Outcome Logic (Zilberstein 2023):

- **Under-approximation**: may **miss** bugs (false negatives for safety)
- Focus: **finding** bugs, not proving absence
- Guarantees: found bugs are **real** (0% false positives for manifest)

Resolution: Classify channel bugs by **detectability**.

Definition 90.1 (Channel Bug Classification). **Manifest Channel Bugs** (0% False Positive Rate): Bugs detectable with **empty presumption** (emp), independent of calling context. These correspond to Le 2022’s manifest errors.

1. **Type Mismatch** — Send value of wrong type:

$$[\text{emp}] \text{ send}(k, 42) [k \text{ expects string}; \text{Er}]$$

2. **Protocol Violation** — Wrong operation at current state:

$$[\text{emp}] \text{ send}(k, v) [\text{session_state}(k) = \text{receiving}; \text{Er}]$$

3. **Definite Deadlock** — All participants blocked forever:

$$[\text{emp}] P_1 \mid P_2 \mid \dots \mid P_n [\forall i. \text{blocked}(P_i); \text{Er}]$$

4. **Channel Linearity Violation** — Use after close/move:

$$[\text{emp}] \text{ close}(k); \text{ send}(k, v) [\text{closed}(k); \text{Er}]$$

5. **Session Type Structural Mismatch:**

$$[\text{emp}] P \text{ at } G \upharpoonright p [G \upharpoonright p \text{ undefined or } P \text{ violates } G \upharpoonright p; \text{Er}]$$

Latent Channel Bugs (Require Context/Schedule): Bugs requiring specific **precondition** or schedule to manifest. These are under-approximate findings with confidence < 100%.

1. **Potential Deadlock** under specific interleavings:

$$[\text{schedule} = \sigma] P_1 \mid P_2 [\text{blocked}; \text{Er}]$$

2. **Race on Channel** under specific timing:

$[arrival_order = (m_1, m_2)] \text{ recv}(k) [\text{wrong message received}; \text{Er}]$

3. **Conditional Protocol Violation**:

$[x > 0] \text{ branch_on}(x); \text{send}(k, v) [\text{protocol error}; \text{Er}]$

Theorem 90.2 (Session Type Violations Are Manifest (Theorem 14.7.1)). *If G is a coherent global type and process P has $\Gamma \vdash P \triangleright \Delta$ where $\Delta(s) = \llbracket G \rrbracket$ and P violates a session type rule, then the violation is a **manifest bug** under Outcome Logic classification.*

- Proof Sketch.*
1. Session type rules (Figure 7, Honda 2008) are **local** to each process
 2. Type errors (wrong sort, wrong prefix, wrong branch) are detectable from P 's code alone, without caller information
 3. Therefore ISL presumption is $\text{emp} \wedge \text{true}$ (Le 2022 Theorem 3.4)
 4. By True Positives Property: manifest error implies real bug □

Consequence: Session type violations detected statically have 0% false positive rate. This justifies **high** confidence for session type findings in Layer 6.

90.2 Tension 2: Static vs Dynamic Participants

The Participant Count Tension

Honda 2008: Participant count n is **static** (compile-time)

- Global type G has fixed set $\text{pid}(G) = \{1, 2, \dots, n\}$

Link rule synchronizes **exactly** n participants

- Projection $G \upharpoonright p$ is defined for each $p \in \text{pid}(G)$

Go/Rust/Real Programs: Goroutines/threads created **dynamically**

- `for i := 0; i < n; i++ { go worker(ch) }`
- Number of workers determined at **runtime**
- Channel may be shared by unbounded number of participants

Resolution Options:

Option 1: Bounded Model Checking — Unroll dynamic participant creation to depth k (typically $k = 3-5$).

Option 2: Symbolic Participants — Abstract over participant count using universal quantification:

$$\forall n \geq 1. G(n) = \mu t. \text{Worker}_i \rightarrow \text{Coordinator} : ch\langle \text{result} \rangle. t \quad (i \in [1, n])$$

Option 3: Annotation Requirement — User specifies participant bounds via annotations.

Option 4: Pattern Recognition (Recommended Default) — Detect common patterns and apply known-safe protocols.

| Pattern | Global Type Template |
|-------------------|---|
| Fan-Out (1-to-N) | $\mu t. \forall i \in [1, n]. \text{Master} \rightarrow \text{Worker}_i : k\langle \text{task} \rangle. t$ |
| Fan-In (N-to-1) | $\mu t. \exists i \in [1, n]. \text{Worker}_i \rightarrow \text{Master} : k\langle \text{result} \rangle. t$ |
| Producer-Consumer | $\mu t. \text{Producer} \rightarrow \text{Consumer} : k\langle \text{item} \rangle. t$ |
| Scatter-Gather | Fan-Out then Fan-In composition |
| Pipeline | $\text{Stage}_1 \rightarrow \text{Stage}_2 \rightarrow \dots \rightarrow \text{Stage}_n$ |
| Request-Response | $\text{Client} \rightarrow \text{Server} : \text{req}\langle Q \rangle. \text{Server} \rightarrow \text{Client} : \text{resp}\langle R \rangle$ |

Implementation Strategy:

1. First attempt pattern recognition (fast, high confidence)
2. If pattern found: instantiate known-safe global type
3. If no pattern: fall back to bounded model checking with $k = 3$
4. Allow user annotations to override/refine bounds

The following code implements automatic pattern detection from channel usage in the Code Property Graph. The algorithm counts senders and receivers to classify common communication patterns, then generates appropriate global types.

Pattern Detection and Global Type Generation

```
(* Communication patterns recognized from channel usage *)
type channel_pattern =
| FanOut : master:participant → channel_pattern
| FanIn : collector:participant → channel_pattern
| ProducerConsumer : channel_pattern
| RequestResponse : channel_pattern
| Pipeline : stages:list participant → channel_pattern
| UnrecognizedPattern : channel_pattern

(* Detect pattern from sender/receiver counts *)
val detect_channel_pattern : cpg → chan_id → channel_pattern
let detect_channel_pattern cpg ch =
  let senders = get_senders cpg ch in
  let receivers = get_receivers cpg ch in
  match (List.length senders, List.length receivers) with
  | (1, n) when n > 1 → FanOut (List.hd senders)
  | (n, 1) when n > 1 → FanIn (List.hd receivers)
  | (1, 1) → if is_bidirectional cpg ch then RequestResponse
              else ProducerConsumer
  | _ → UnrecognizedPattern

(* Generate global type from detected pattern *)
val pattern_to_global_type : channel_pattern → nat → global_type
let pattern_to_global_type pattern n =
  match pattern with
  | FanOut master →
    GRec "t" (fold_participants n (fun i acc →
      GMsg master (Worker i) 1 SortAny acc) (GVar "t"))
  | FanIn collector →
    GRec "t" (GBranch collector
      (List.map (fun i → (label i, GMsg (Worker i) collector 1 SortAny GEnd))
        (range 1 n)))
  | ProducerConsumer →
    GRec "t" (GMsg Producer Consumer 1 SortAny (GVar "t"))
  | RequestResponse →
    GMsg Client Server 1 SortAny (GMsg Server Client 1 SortAny GEnd)
  | _ → GEnd (* Fallback: use bounded checking *)
```

Connection to Engineering: This pattern-based approach enables practical analysis of dynamically-spawned goroutines and threads without requiring explicit annotations. The recognized patterns cover the majority of real-world concurrent communication idioms.

90.3 Tension 3: Order Preservation

The Message Ordering Tension

Honda 2008 Assumption: TCP-like FIFO per channel

- Messages on **same** channel preserve sending order
- \prec^{OO} (Output-Output) dependency exploits this
- $s!m_1; s!m_2$ guarantees m_1 arrives before m_2

Reality: Cross-channel messages have **no** ordering guarantee

- $s!m_1; t!m_2$ does **not** guarantee m_1 arrives before m_2
- Even same sender: different channels = no ordering
- Two Buyer Protocol **requires** separate channels for this reason

Resolution: Use Honda 2008's **modular approach** (Section 6.1).

Ordering Rules:

1. **Same-Channel Ordering** (\prec^{OO} preserved): For prefixes n_1, n_2 with same sender p and same channel k , if $n_1 \prec n_2$ then $n_1 \prec^{OO} n_2$.
2. **Cross-Channel Independence:** For prefixes n_1, n_2 with different channels $k_1 \neq k_2$, $\neg(n_1 \prec^{OO} n_2) \wedge \neg(n_2 \prec^{OO} n_1)$.
3. **Explicit Synchronization:** If ordering between k_1 and k_2 is required, add sync edge:

$$G = p \rightarrow q : k_1 \langle U_1 \rangle . q \rightarrow r : \text{sync} \langle \text{ack} \rangle . p \rightarrow r : k_2 \langle U_2 \rangle . \dots$$

Analysis Implication: When analyzing code, do **not** assume cross-channel ordering. Flag potential races where code assumes ordering without sync.

The following code implements a safety check that verifies cross-channel ordering assumptions are properly synchronized:

Cross-Channel Ordering Safety Check

```
(* Verify that cross-channel message pairs are properly synchronized *)
val cross_channel_ordering_safe : global_type → bool
let cross_channel_ordering_safe g =
  let prefixes = collect_prefixes g in
  let cross_channel_pairs = List.filter (fun (n1, n2) →
    channel_of n1 ≠ channel_of n2 &&
    sender_of n1 = sender_of n2 &&
    n1 'prefix_ordered' n2
  ) (pairs prefixes) in
  (* For each cross-channel pair, verify no ordering assumption *)
  List.for_all (fun (n1, n2) →
    not (requires_ordering g n1 n2) ||
    has_sync_path g n1 n2
  ) cross_channel_pairs
```

Two Buyer Protocol Revisited: The Two Buyer Protocol from [Honda08] uses **four** channels (b_1, b_2, b'_2, s) precisely because:

- Buyer₂ receives from Seller on b_2 (quote)
- Buyer₂ receives from Buyer₁ on b'_2 (contribution)
- Without separate channels, messages could arrive out of order
- This is **not** a workaround but **correct** protocol design

90.4 Tension 4: Select Non-Determinism

The Select Statement Tension

Honda 2008: Deterministic branching (**receiver** chooses)

- $G = p \rightarrow q : k\{l_1 : G_1, l_2 : G_2\}$
- Sender p **selects** which label to send
- Receiver q **branches** based on received label
- Selection is **internal choice** at sender

Go select: Non-deterministic (**scheduler** chooses)

```
select {
  case msg := <-ch1: handle1(msg)
  case msg := <-ch2: handle2(msg)
```

```
}

```

- Which case fires depends on message **arrival order**
- This is **external choice** resolved by environment

Resolution: Model select as **external choice on multiple channels**.

Key Insight: Go select is **not** the same as session type selection. It is better modeled as external choice where the environment (message arrival timing) determines which branch executes.

Select Statement Type Checking

```
type select_branch = {
  channel : chan_id;
  direction : SendOrRecv;
  body : process;
  local_type : local_type;
}

type select_stmt = list select_branch

(* Type check select statement *)
val check_select :  $\gamma \rightarrow \text{select\_stmt} \rightarrow \delta \rightarrow \text{option } \delta$ 
let check_select g branches d =
  (* All branches must be individually well-typed *)
  let branch_results = List.map (fun br  $\rightarrow$ 
    match br.direction with
    | Recv  $\rightarrow$ 
      let expected_type = LRecv br.channel (infer_sort br.body) br.local_type in
      check_process g br.body (update d br.channel expected_type)
    | Send  $\rightarrow$ 
      let expected_type = LSend br.channel (infer_sort br.body) br.local_type in
      check_process g br.body (update d br.channel expected_type)
  ) branches in
  (* If all succeed, combine results (external choice) *)
  if List.for_all is_some branch_results then
    Some (merge_branch_ $\delta$ s (List.map Option.get branch_results))
  else None

```

Theorem 90.3 (Select Safety (Theorem 14.7.2)). *If process P uses select over channels $\{k_1, \dots, k_n\}$ where:*

1. *Each channel k_i has valid local type T_i at P*
2. *The select body B_i for each case follows T_i*
3. *All branches B_i are type-safe continuations*

Then P is safe regardless of which branch the scheduler selects.

The following code models how select statements can be encoded in global types as parallel composition (external choice):

Encoding Select in Global Types

```
(* Model select as external choice in global type *)
val encode_select : select_stmt  $\rightarrow$  participant  $\rightarrow$  global_type
let encode_select branches self =
  let branch_globals = List.map (fun br  $\rightarrow$ 
    match br.direction with
    | Recv  $\rightarrow$  GMsg (get_sender br.channel) self br.channel (sort_of br) GEnd
    | Send  $\rightarrow$  GMsg self (get_receiver br.channel) br.channel (sort_of br) GEnd
  ) branches in
  GPar branch_globals (* Parallel composition = external choice *)

```

Key Insight: The GPar constructor represents parallel composition where exactly one branch executes (first message arrival wins). This differs from session type selection where the sender actively chooses.

Bug Detection: For `select { case x := <-ch1: ... case ch2 <- y: ... }`, if `ch1` and `ch2` connected to **same** remote, potential race. Analysis: warn if both branches communicate with same participant.

90.5 Tension 5: Global Type Origin

The Global Type Origin Tension

Honda 2008 Assumption: G is **given** by programmer

- Development methodology: design G first, implement to G
- Type checking: verify process P conforms to $G \upharpoonright p$
- G serves as contract/specification

Reality: Most code has **implicit** protocols

- Legacy code: no explicit protocol specification
- Quick prototypes: evolve protocol during development
- Third-party code: protocols undocumented

Resolution: Three-Tier Protocol Extraction.

Tier 1: Explicit Annotation (Highest Confidence, 100%) — User provides global type via annotation. Verification: Check code conforms to annotation.

Tier 2: Pattern Inference (Medium Confidence, 80–90%) — Detect common communication idioms automatically.

| Code Pattern | Inferred Global Type |
|---|--|
| <code>ch <- x; y := <-ch</code> (send then recv = req-resp) | $p \rightarrow q : k\langle T \rangle. q \rightarrow p : k'\langle T' \rangle$ |
| <code>for { ch <- data }</code> (loop send = streaming) | $\mu t. p \rightarrow q : k\langle T \rangle. t$ |
| <code>switch msg.Type { case A: ... }</code> (type switch = branching) | $p \rightarrow q : k\{A : G_A, B : G_B, \dots\}$ |

Tier 3: Local Type Extraction + Coherence Check (Lower Confidence, 60–80%) — Extract local types from each participant, then check coherence.

Algorithm:

1. For each goroutine/thread p , extract local type T_p by:
 - Traversing channel operations in CFG order
 - Building type from `send $\rightarrow !U$` , `recv $\rightarrow ?U$` , `select $\rightarrow \&/\oplus$`
2. Check if $\{T_p @ p\}$ family is **coherent**:
 - Attempt to construct G such that $G \upharpoonright p = T_p$ for all p
 - If construction fails, report potential protocol error

The following code implements the three-tier resolution strategy, returning both the inferred global type and a confidence level indicating which tier was used:

Three-Tier Global Type Resolution

```
(* Source of global type with confidence *)
type global_type_source =
  | ExplicitAnnotation : text:string → global_type_source
  | PatternInferred    : pattern:channel_pattern → global_type_source
  | LocalExtracted     : local_types:list (participant * local_type) → global_type_source

(* Resolve global type using three-tier strategy *)
val resolve_global_type : cpg → chan_id → (global_type * float * global_type_source)
```

```

let resolve_global_type cpg ch =
  (* Tier 1: Check for explicit annotation *)
  match find_session_annotation cpg ch with
  | Some annot →
    let g = parse_global_type annot in
    (g, 1.0, ExplicitAnnotation annot)
  | None →
    (* Tier 2: Attempt pattern inference *)
    let pattern = detect_channel_pattern cpg ch in
    match pattern with
    | UnrecognizedPattern →
      (* Tier 3: Extract local types and check coherence *)
      let participants = get_channel_participants cpg ch in
      let local_types = List.map (fun p →
        (p, extract_local_type cpg ch p)) participants in
      (match construct_global_from_locals local_types with
      | Some g → (g, 0.7, LocalExtracted local_types)
      | None → (GEnd, 0.0, LocalExtracted local_types)) (* Incoherent! *)
    | _ →
      let n = count_participants cpg ch in
      let g = pattern_to_global_type pattern n in
      (g, 0.85, PatternInferred pattern)

(* Extract local type by traversing CFG *)
val extract_local_type : cpg → chan_id → participant → local_type
let extract_local_type cpg ch p =
  let cfg = get_cfg cpg p in
  let chan_ops = filter_channel_ops cfg ch in
  fold_cfg_order chan_ops (fun acc op →
    match op with
    | SSend _ v → LSend ch (type_of v) acc
    | SRecv dst _ → LRecv ch (type_of dst) acc
    | SSelect _ cases → LSelect ch (List.map (fun (l, _) → (l, LEnd)) cases)
    | _ → acc
  ) LEnd

```

Connection to Engineering: The confidence values (1.0, 0.85, 0.7, 0.0) directly inform the finding severity in SARIF output. Explicit annotations give confirmed bugs, pattern inference gives high-confidence warnings, and local extraction gives medium-confidence warnings.

90.6 Integration with Outcome Logic Bug Classification

Outcome Logic for Channel Bugs

```

(* OL Triples for Channel Operations:

<chan_created(k,T)> send(k,v) <v:T ∧ msg_in_flight(k,v)>
  After send, message v of type T is in the channel's queue.

<msg_in_flight(k,v)> recv(k) <received(k,v)>
  After recv, if message was in flight, it's now received.

<¬closed(k)> close(k) <closed(k)>
  After close on open channel, channel is closed.

<closed(k)> send(k,v) <error: send_on_closed>
  Send on closed channel is an error.

<session_state(k) = T> send(k,v) <session_state(k) = advance(T,!v)>
  Session type advances after send.
*)

type channel_postcondition =
  | ChanOk : state:channel_state → channel_postcondition

```

```

| ChanError : error:channel_error → channel_postcondition

type channel_state = {
  created : bool;
  closed : bool;
  msg_queue : list value;
  session_state : option local_type;
}

type channel_error =
| SendOnClosed : chan:nat → channel_error
| RecvOnClosed : chan:nat → channel_error
| TypeMismatch : expected:sort → actual:sort → channel_error
| ProtocolViolation : expected:local_type → actual:chan_op → channel_error
| Deadlock : blocked:list participant → channel_error

(* Classification: manifest vs latent for channel bugs *)
val classify_channel_bug : channel_error → isl_presumption → bug_classification
let classify_channel_bug err presumption =
  match err with
  | SendOnClosed _ | RecvOnClosed _ | TypeMismatch _ _ →
    (* These are manifest: detectable without caller context *)
    if presumption = EmpPresumption then
      Manifest { kind = channel_error_to_kind err; confidence = 1.0 }
    else
      Latent { kind = channel_error_to_kind err;
               required_context = presumption }

  | ProtocolViolation _ _ →
    (* Manifest if local type violation, latent if depends on message order *)
    if is_structural_violation err then
      Manifest { kind = ProtocolError; confidence = 1.0 }
    else
      Latent { kind = ProtocolError;
               required_context = MessageOrderContext }

  | Deadlock blocked →
    (* Manifest if ALL paths lead to deadlock, latent if schedule-dependent *)
    if all_paths_deadlock blocked then
      Manifest { kind = DefiniteDeadlock; confidence = 1.0 }
    else
      Latent { kind = PotentialDeadlock;
               required_context = ScheduleContext }

```

90.7 Integration with Security Analysis

Channels are conduits for both **data** and **control**. Security analysis must track taint propagation through channels, detect information flow violations, and identify channels as potential covert channels. Session types can serve as information flow policies, declaring the security level of data flowing through each channel.

The following code defines security labels for channels and implements taint propagation rules:

```

Channel Security Analysis

(* Security labels for channels *)
type security_level = High | Low | Unknown

type channel_security = {
  chan_id : nat;
  data_level : security_level;      (* Security of data on channel *)
  timing_level : security_level;    (* Security of timing information *)
  participants : list (participant * security_level); (* Cleared participants *)
}

```

```

}

(* Taint propagation through channels *)
val propagate_channel_taint : taint_state → ir_stmt → taint_state
let propagate_channel_taint state stmt =
  match stmt with
  | SSend ch v →
    if is_tainted state v then
      mark_channel_tainted state ch (get_taint_source state v)
    else state

  | SRecv dst ch →
    if is_channel_tainted state ch then
      mark_var_tainted state dst (get_channel_taint_source state ch)
    else state

  | SChanDelegate ch delegated →
    if is_channel_tainted state delegated then
      mark_channel_tainted state ch (get_channel_taint_source state delegated)
    else state

  | _ → state

(* Information flow check: does channel violate policy? *)
type channel_ifc_violation =
  | HighToLowLeak : high_chan:nat → low_receiver:participant → channel_ifc_violation
  | UnclearedReceiver : chan:nat → required_level:security_level
    → actual_level:security_level → channel_ifc_violation
  | TimingLeak : chan:nat → channel_ifc_violation

```

90.8 Integration with Multi-Language Analysis

Cross-Language Channel Boundaries

Channels can cross language boundaries (FFI, IPC, RPC):

- Go channel to Rust mpsc via CGO
- Python multiprocessing.Queue to C extension
- TypeScript WebSocket to Go server

Session types must be **compatible** (not necessarily identical) at boundaries.

Cross-Language Session Type Compatibility: Two session types T_1 (language L_1) and T_2 (language L_2) are **compatible** at a boundary if there exists a subtyping relation:

$$T_1 \leq_{\text{boundary}} T_2 \quad \text{or} \quad T_2 \leq_{\text{boundary}} T_1$$

Subtyping Across Boundaries (extends Gay & Hole 2005):

- **Output is covariant:** Can send more specific type
- **Input is contravariant:** Can receive more general type
- **Selection is covariant in labels:** Can select fewer labels
- **Branching is contravariant in labels:** Can handle more labels

The following code implements boundary subtyping checking between session types from different languages:

```

Cross-Language Boundary Subtyping
(* Cross-language type mapping *)
type cross_lang_type_map = {
  source_lang : language;
  target_lang : language;
  type_mappings : list (sort * sort);
}

```

```

(* Boundary subtyping:  $T_1 \leq_{\text{boundary}} T_2$  *)
val boundary_subtype : cross_lang_type_map → local_type → local_type → bool
let rec boundary_subtype map t1 t2 =
  match (t1, t2) with
  | (LSend k1 u1 cont1, LSend k2 u2 cont2) when k1 = k2 →
    sort_compat map u1 u2 && (* Covariant *)
    boundary_subtype map cont1 cont2

  | (LRecv k1 u1 cont1, LRecv k2 u2 cont2) when k1 = k2 →
    sort_compat map u2 u1 && (* Contravariant! *)
    boundary_subtype map cont1 cont2

  | (LSelect k1 branches1, LSelect k2 branches2) when k1 = k2 →
    (* Covariant: t1 can select fewer labels *)
    List.for_all (fun (l, t1_l) →
      List.mem_assoc l branches2 &&
      boundary_subtype map t1_l (List.assoc l branches2)
    ) branches1

  | (LBranch k1 branches1, LBranch k2 branches2) when k1 = k2 →
    (* Contravariant: t1 can handle more labels *)
    List.for_all (fun (l, t2_l) →
      List.mem_assoc l branches1 &&
      boundary_subtype map (List.assoc l branches1) t2_l
    ) branches2

  | (LEnd, LEnd) → true
  | _ → false

(* Boundary guard for runtime checking when static verification fails *)
type boundary_guard =
  | TypeGuard : expected:sort → boundary_guard
  | ProtocolGuard : expected:local_type → boundary_guard

val needs_boundary_guard : lang_session_type → lang_session_type → bool
let needs_boundary_guard t1 t2 =
  let map = get_type_map (lang_of t1) (lang_of t2) in
  not (boundary_subtype map (session_type_of t1) (session_type_of t2))

```

Connection to Part IX: This boundary subtyping integrates with the multi-language semantic framework (Part IX). When static verification of boundary compatibility fails, runtime guards are inserted to check type and protocol compliance dynamically.

90.9 Decidability Results

Decidability and Complexity Summary

Decidable (Polynomial):

- Linearity checking: $O(|G|^2)$
- Coherence checking: $O(n \times |G|^2)$ where n = participants
- Type inference for processes: $O(|P| \times |G|)$
- Projection $G \upharpoonright p$: $O(|G|)$
- Subtyping $T_1 \leq T_2$: $O(|T_1| \times |T_2|)$ for finite types

Decidable (Exponential):

- Subtyping with recursive types: EXPTIME [GayHole05]
- Global type synthesis from locals: 2-EXPTIME in worst case

Undecidable (in general):

- Deadlock detection: PSPACE-complete, undecidable for infinite state
- Liveness properties: undecidable for general processes

Practical Approach:

- Use bounded model checking for undecidable properties
- Exploit session type structure for efficiency
- Under-approximate when sound analysis times out

Theorem 90.4 (Linearity Decidability (Honda 2008 Prop 3.6)). *Linearity of a global type G is decidable in $O(|G|^2)$ time.*

Proof Sketch. 1. Unfold G exactly once (linear in $|G|$)

2. Check suppression conditions for each prefix pair

3. At most $|G|^2$ prefix pairs to check □

Theorem 90.5 (Coherence Decidability (Honda 2008 Thm 4.3)). *Coherence of G is decidable in $O(n \times |G|^2)$ time.*

Proof Sketch. 1. Check linearity: $O(|G|^2)$

2. For each participant p (n total):

- Compute projection $G \upharpoonright p$: $O(|G|)$
- Check projection is defined: $O(|G|)$

3. Total: $O(|G|^2) + n \times O(|G|) = O(n \times |G|^2)$ □

Theorem 90.6 (Type Inference Decidability (Honda 2008 Thm 4.6)). *Given annotated program phrase P and Γ , deciding if $\exists \Delta. \Gamma \vdash P \triangleright \Delta$ is decidable, and such Δ can be computed.*

Proof Sketch. 1. Type rules (Figure 7, Honda 2008) are syntax-directed

2. Each rule determines unique typing for subterms

3. Traverse P once, applying rules bottom-up □

Deadlock Decidability:

- For **single session** with finite recursive types: Progress is guaranteed by typing (Theorem 5.12, Honda 2008). Therefore: no deadlock analysis needed — types ensure progress.
- For **multiple interleaved sessions**: Progress requires “simple” and “well-linked” conditions. Checking these is decidable but may require exploring execution tree. PSPACE-complete in general.

The following code provides a practical algorithm that combines these decidability results with the tier-based resolution strategy:

Practical Channel Analysis Algorithm

```
(* Combined channel analysis using decidability results *)
type channel_analysis_result = {
  global_type : global_type;
  is_linear : bool;
  is_coherent : bool;
  session_typed : option bool;
  deadlock_free : option bool;
}

val analyze_channel_properties : cpg → chan_id → channel_analysis_result
let analyze_channel_properties cpg ch =
  let (g, confidence, source) = resolve_global_type cpg ch in
  {
    global_type = g;
    is_linear = check_linearity g;                (* O(|G|^2), always run *)
    is_coherent = check_coherence g;              (* O(n*|G|^2), always run *)
    session_typed =
      if confidence > 0.7 then
```

```

    Some (check_process_conformance cpg g) (* 0(|P|*|G|), run if confident *)
  else None;
deadlock_free =
  if is_simple cpg && is_well_linked cpg then
    Some true (* Type system guarantees *)
  else
    bounded_deadlock_check cpg 10; (* Bounded check, depth 10 *)
}

```

Connection to Engineering: This algorithm runs linearity and coherence checks unconditionally (fast, polynomial), but gates session type conformance on confidence level. Deadlock checking uses the type-theoretic guarantee when applicable, falling back to bounded model checking otherwise.

90.10 F* Theorem: Manifest Bug Classification

This section presents the main theoretical result connecting session type theory with Outcome Logic: session type violations are **manifest bugs** with 0% false positive rate. This justifies high confidence for session type findings in the analysis output.

The theorem relies on two key observations: (1) session type rules are **local** to each process—they examine only the process structure, not the calling context; and (2) protocol violations are therefore detectable with an empty ISL presumption (`emp`), which by Le 2022’s True Positives Property implies the bug is manifest.

Manifest Bug Theorem for Channel Violations — Main Result

```

(* This theorem establishes that session type violations are MANIFEST bugs
   under Outcome Logic, meaning they have 0% false positive rate. *)

(* Prerequisites *)
assume type process : Type
assume type global_type : Type
assume type  $\gamma$  : Type (* Sorting context *)
assume type  $\delta$  : Type (* Typing context *)
assume type participant : Type

(* Typing judgment:  $\Gamma \vdash P \triangleright \Delta$  *)
assume val typing_judgment :  $\gamma \rightarrow \text{process} \rightarrow \delta \rightarrow \text{Type}$ 

(* Coherent global type *)
assume val coherent : global_type  $\rightarrow$  bool

(* Encode global type as family of local types *)
assume val encode_global : global_type  $\rightarrow$   $\delta$ 

(* Protocol violation predicate *)
assume val protocol_violation : process  $\rightarrow$  global_type  $\rightarrow$  bool

(* Manifest bug predicate (from Section 12.3) *)
assume val manifest_bug : process  $\rightarrow$  Type

(* True Positives Property (Le 2022 Theorem 3.4) *)
assume val true_positives_property :
  t:isl_triple  $\rightarrow$ 
  Lemma (requires t.presumption == EmpPresumption  $\wedge$ 
    satisfiable t.postcondition)
    (ensures manifest_error t.command)

(* MAIN THEOREM: Channel Bugs Are Manifest *)
(*
  * Theorem: Session Type Violations Are Manifest Bugs
  *
  * If:

```

```

* 1. G is a coherent global type
* 2. Process P is well-typed under Gamma with Delta
* 3. Delta at session s encodes G
* 4. P violates the protocol specified by G
*
* Then:
* The violation is a MANIFEST bug (0% false positive rate)
*)
val channel_manifest_bugs :
  p:process →
  gt:global_type →
  g:γ →
  d:δ →
  s:nat →
  Lemma (requires coherent gt ∧
    typing_judgment g p d ∧
    δ_at_session d s == encode_global gt ∧
    protocol_violation p gt)
    (ensures manifest_bug p)

let channel_manifest_bugs p gt g d s =
  (* Step 1: Session type rules are LOCAL to process p *)
  assert (local_checkable (typing_judgment g p d));

  (* Step 2: Protocol violation is detectable from P alone *)
  let violation_location = find_violation_site p gt in
  assert (violation_in_process violation_location p);

  (* Step 3: Construct ISL triple with emp presumption *)
  let isl = {
    presumption = EmpPresumption;
    command = p;
    postcondition = protocol_error_formula violation_location;
  } in

  (* Step 4: Postcondition is satisfiable (violation ∃) *)
  assert (satisfiable isl.postcondition);

  (* Step 5: Apply True Positives Property (Le 2022 Theorem 3.4) *)
  true_positives_property isl;

  (* Step 6: Conclude: manifest bug *)
  ()

```

Corollaries:

1. **Type mismatch is manifest:** If expected \neq actual sort, it's a manifest bug.
2. **Definite deadlock is manifest:** If all paths lead to deadlock with no progress possible, it's a manifest bug.
3. **Linearity violation is manifest:** Use-after-close or use-after-move is a manifest bug.

Integration with Layer 6 Confidence Levels: The following code maps channel bug classifications to confidence levels for SARIF output. Manifest bugs (type mismatch, definite deadlock, linearity violation) are reported as confirmed bugs with “error” severity. Latent bugs (potential deadlock, race conditions) are reported as conditional warnings requiring further investigation.

Session Finding Confidence Mapping

```

val session_finding_confidence : channel_bug_class → confidence_level
let session_finding_confidence bug =
  match bug with
  | ChannelManifest kind →
    (* Manifest bugs have HIGH confidence --- 0% FP rate *)

```

```

match kind with
| ChanTypeMismatch _ _ _ → ConfirmedBug    (* Static type error *)
| ChanProtocolViolation _ _ → HighConfBug    (* Protocol state error *)
| ChanDefiniteDeadlock _ → ConfirmedBug      (* All paths deadlock *)
| ChanLinearityViolation _ _ → ConfirmedBug  (* Ownership error *)
| ChanStructuralMismatch _ _ → HighConfBug   (* Structure mismatch *)

| ChannelLatent kind →
  (* Latent bugs have LOWER confidence --- require context *)
  match kind with
  | ChanPotentialDeadlock _ → ConditionalBug  (* Schedule-dependent *)
  | ChanRaceCondition _ → TrueAlarm           (* Timing-dependent *)
  | ChanConditionalViolation _ → ConditionalBug (* Path-dependent *)

(* Final classification for SARIF output *)
type sarif_channel_finding = {
  rule_id : string;
  level : string;           (* "error" / "warning" / "note" *)
  message : string;
  location : source_location;
  confidence : float;       (* 0.0 to 1.0 *)
}

val to_sarif : channel_bug_class → source_location → sarif_channel_finding
let to_sarif bug loc =
  let conf = session_finding_confidence bug in
  {
    rule_id = channel_bug_rule_id bug;
    level = if conf = ConfirmedBug || conf = HighConfBug then "error" else "warning";
    message = channel_bug_message bug;
    location = loc;
    confidence = confidence_to_float conf;
  }

```

Connection to Engineering: This mapping directly determines how channel analysis findings appear in IDE integrations and CI/CD pipelines. Confirmed bugs (0.95–1.0 confidence) break builds; high-confidence warnings (0.8–0.95) require review; conditional findings (0.5–0.8) are informational.

Part XIV Summary

This part established the theoretical and practical foundations for channel analysis in the synthesis framework:

- **Binary Session Types** (Section ??): Formalized Honda 1998’s type system for structured communication, including delegation.
- **Global Types and Projection** (Sections ??–86.3): Defined multiparty session types following Honda 2008, with causality analysis and coherence conditions.
- **Theoretical Reconciliation** (Sections 90.2–90.5): Resolved fundamental tensions between session type theory and practical engineering (dynamic participants, message ordering, select non-determinism, implicit protocols).
- **Integration with Existing Analyses** (Sections 89.4–90.8): Extended IFDS, taint analysis, ownership tracking, and multi-language boundaries for channel operations.
- **Manifest Bug Theorem** (Section 90.10): Proved that session type violations are manifest bugs under Outcome Logic, justifying high confidence for channel analysis findings.

The key engineering insight is that session types provide a **compositional** approach to verifying concurrent programs: local type checking at each participant guarantees global protocol

compliance, enabling scalable analysis of complex distributed systems.

A. Paper Priority Matrix

This appendix provides a prioritized reference of foundational papers organized by contribution category, with priority scores indicating their importance to the brrr-machine implementation.

A.1 Priority Matrix

| Paper | Priority | Category | Key Contribution |
|-------------------------------|-----------|---------------------|--|
| Cousot 1977 | 10 | Foundation | Abstract interpretation |
| Yamaguchi 2014 | 10 | Representation | Code Property Graph |
| Reps 1995 | 9 | Algorithm | IFDS |
| Reps 1997 | 9 | Algorithm | CFL-reachability |
| Andersen 1994 | 9 | Pointer | Inclusion-based |
| Steensgaard 1996 | 8 | Pointer | Unification-based |
| Lattner 2007 (DSA) | 9 | Pointer | Unification + context-sensitivity + heap cloning for C/C++ > 100K LOC |
| Moggi 1991 | 9 | Effects | Monads |
| Plotkin 2003/2009 | 9 | Effects | Algebraic effects |
| Leijen 2014/2017 | 9 | Effects | Row polymorphism |
| Girard 1987 | 9 | Types | Linear logic |
| Reynolds 2002 | 9 | Types | Separation logic |
| Jung 2018 (Iris) | 9 | Types | Iris framework, higher-order ghost state, cameras, view shifts, step-indexed semantics |
| Muller 2016 (Viper) | 9 | Verification | IVL for permission-based reasoning, magic wands, quantified permissions |
| Mulder 2022 (Diaframe) | 9 | Verification | Automated Iris proofs, bi-abduction with postponed existentials |
| Denning 1977 | 9 | Security | Information flow |
| Livshits 2005 | 9 | Security | Taint analysis |
| Tripp 2009 | 9 | Security | Thin slicing |
| Matthews 2007 | 9 | Multi-lang | Boundary semantics |
| Hammer 2014 | 9 | Incremental | Adapton |
| Ferrante 1987 | 9 | Representation | PDG |
| Horwitz 1990 | 9 | Representation | SDG |
| Weiser 1984 | 8 | Representation | Slicing |
| Sridharan 2005 | 8 | Pointer | Demand-driven |
| Smaragdakis 2011 | 8 | Pointer | Datalog |
| Calcagno 2009 | 9 | Pointer | Bi-abduction |
| Aiken 1999 | 9 | Types | Set constraints |
| Cousot 1992 | 9 | Foundation | Widening |
| Goguen 1992 | 8 | Multi-lang | Institutions |
| Wagner 1998 | 8 | Incremental | Parsing |
| Zilberstein 2023 | 10 | Foundation | Outcome Logic (replaces IL) |

Continued on next page

| Paper | Priority | Category | Key Contribution |
|---------------------------|----------|-----------------|--|
| Kang 2017 | 10 | Memory Model | Promising Semantics (fixes C11 thin-air bug) |
| Lee 2020 | 10 | Memory Model | Promising 2.0 (capped memory, global opts, fixes ARMv8) |
| Podkopaev 2019 | 10 | Memory Model | IMM intermediate model (O(n+m) compilation proofs, 33K Coq) |
| Leroy 2009 (CompCert) | 10 | Verification | Verified C compiler, semantics preservation proofs, simulation relations |
| O'Hearn 2020 | 8 | Foundation | Incorrectness Logic (historical) |
| Patterson 2022 | 9 | Multi-lang | Realizability models |
| King 1976 | 9 | Algorithm | Symbolic execution, path conditions |
| Kozen 1981 | 8 | Foundation | Probabilistic semantics, Theorem 6.1 |
| Cousot 2012 | 8 | Foundation | Probabilistic AI, three abstraction axes |
| Bruni 2023 (LCL) | 9 | Foundation | Local Completeness Logic, unified over/under approx |
| Le 2022 (Pulse-X) | 10 | Foundation | ISL, Manifest/Latent classification, True Positives Property |
| Vanegue 2025 | 9 | Algorithm | Non-termination proving, Pulse-infinity |
| Crary 1999 | 9 | Types | Capability multiplicities |
| Xi 1999 | 8 | Types | Dependent ML / constraint domains |
| Watt 2018 | 8 | Verification | Verified interpreter pattern |
| Disselkoen 2019 (MS-Wasm) | 8 | Memory Safety | Progressive memory safety for WebAssembly |
| Perrone & Romano 2024 | 7 | Security Survey | Comprehensive WebAssembly security review (121 papers) |
| Rupta/Li 2024 | 9 | Pointer | Stack filtering, Rust MIR analysis, on-the-fly CG |
| Sagiv 2002 (TVLA) | 9 | Foundation | Three-valued logic, shape analysis, focus/coerce |
| Distefano 2006 | 8 | Shape Analysis | Symbolic heaps, junk predicate, canonicalization |
| Pnueli 1977 | 8 | Temporal Logic | G, F, \leadsto , U operators, P1/P2 liveness principles |
| Batty 2011 | 10 | Memory Model | C11 semantics, coherence axioms, DRF-SC |
| VeriFFI 2025 | 8 | FFI | Representation predicates, GC-isomorphism |
| Furr & Foster 2008 | 8 | FFI | Multilingual type inference, representational types |
| Patterson & Ahmed 2022 | 9 | Multi-lang/IR | Semantic IR via realizability models, convertibility relations |
| Strom & Yemini 1986 | 9 | Typestate | Original typestate concept, state machine semantics |

Continued on next page

| Paper | Priority | Category | Key Contribution |
|------------------------------------|----------|---------------|---|
| Boyapati 2003 | 9 | Ownership | Owner-as-dominator discipline, ownership types |
| Bierhoff 2007 | 9 | Ownership | 5 access permissions (unique/-full/share/immutable/pure) |
| Siek 2006 | 8 | Types | Gradual typing, non-transitive consistency relation |
| Garcia 2016 (AGT) | 9 | Types | Abstracting Gradual Typing: gradual types as abstract interpretations |
| Honda 1998 | 9 | Session Types | Binary session types, duality, linear channel usage |
| Trabish 2018 | 9 | Symbolic Exec | Chopped SE, skip regions, lazy recovery |
| Honda 2008 | 9 | Session Types | Multiparty asynchronous session types, global types, projection |
| Sui & Xue 2016 (SVF) | 9 | Value-Flow | Sparse value-flow analysis, Memory SSA, SVFG construction |
| Sui, Ye & Xue 2012 | 9 | Value-Flow | Full-sparse value-flow, source-sink reachability |
| Chow et al. 1996 | 8 | Memory SSA | Memory SSA foundation, mu/chi annotations |
| Li et al. 2020 (ZIPPER) | 9 | Pointer | Selective context sensitivity, precision flow graph |
| Huang et al. 2023 (JARVIS) | 9 | Call Graph | Python call graph via Function Type Graph, C3 MRO |
| Agat 2000 | 7 | Security | Timing channel analysis, timing-sensitive noninterference |
| Guarnieri et al. 2020 (SPECTECTOR) | 8 | Security | Speculative Non-Interference (SNI) |
| Almeida et al. 2016 (CT-Verif) | 8 | Security | Constant-time verification via product programs |
| Cadar et al. 2008 (KLEE) | 8 | Testing | KLEE symbolic execution engine, constraint optimization |
| Clarke, Emerson, Sistla 1986 | 7 | Verification | CTL model checking, branching-time temporal logic |
| Naeem et al. 2010 | 7 | Algorithms | Practical IFDS extensions and optimizations |
| Russo & Sabelfeld 2006 | 7 | Security | Dynamic monitors for concurrent IFC |
| Sabelfeld & Myers 2003 | 8 | Security | Comprehensive language-based IFC survey |
| Sabelfeld & Sands 2000 | 7 | Security | Probabilistic noninterference |
| Godefroid et al. 2005 (DART) | 8 | Testing | Directed Automated Random Testing, concolic execution origins |

Continued on next page

| Paper | Priority | Category | Key Contribution |
|------------------------------|----------|------------------|---|
| Sen et al. 2005 (CUTE) | 8 | Testing | Concolic testing foundations, pointer constraint separation |
| Yun et al. 2018 (QSYM) | 9 | Testing | Optimistic concolic execution, native instrumentation |
| Pradel & Sen 2018 (DeepBugs) | 8 | ML/Bug Detection | Name-based bug detection via word2vec embeddings |
| Smith & Volpano 1998 | 7 | Security | Type-based secure information flow |
| Spath et al. 2019 (SPDS) | 9 | Algorithm | Synchronized pushdown systems, WPDS encoding |
| Conrado et al. 2025 (MCFL) | 9 | Algorithm | Multiple context-free language reachability, d-MCFL hierarchy |

A.2 Collection 2 Priority Justifications

Bierhoff 2007 (9/10) Critical for modular analysis with aliasing. The 5-permission system (unique, full, share, immutable, pure) directly fills Gap D.1 for library modeling and enables precise resource tracking where current `ownership_state` is too simple. Frame-based inheritance handling addresses OOP analysis needs.

Siek 2006 (8/10) Essential for Python/JavaScript analysis. Type consistency (non-transitive) provides soundness for boundary checking in Section 9.1.2. Cast insertion algorithm enables systematic boundary term generation. Pay-as-you-go semantics allows efficient code generation for known types.

Garcia 2016 AGT (9/10) Principled foundation connecting gradual typing to abstract interpretation. Shows gradual types form a Galois connection: $\gamma(?) = \text{all types}$, consistency = non-empty intersection. **Critical:** Derives non-transitivity mathematically (not stipulated). Evidence semantics enable precise blame tracking at multi-language boundaries.

Honda 1998 (9/10) Foundation for binary session types. Establishes type discipline for structured communication with duality (send/recv correspondence), linearity (channels used exactly once per direction), and progress (no deadlock in single session). Essential for Go channel analysis and Rust mpsc.

Honda 2008 (9/10) Extends binary session types to multiparty asynchronous sessions. Global types describe complete protocol scenarios; projection extracts local types for each participant. Critical for microservice protocol analysis and distributed system verification.

Sui & Xue 2016 (9/10) Alternative to IFDS for source-sink reachability problems. Sparse value-flow representation is more efficient when memory regions $R \ll$ dataflow domain D . Memory SSA (μ/χ annotations) precisely tracks address-taken variables.

Li et al. 2020 ZIPPER (9/10) Selective context sensitivity for pointer analysis. Identifies precision-critical methods via three value-flow patterns (direct, wrapped, unwrapped). Achieves 98.8% precision of full 2obj analysis with $3.4\times$ speedup.

Lattner, Lenharth, Adve 2007 DSA (9/10) Scalable pointer analysis for large C/C++ codebases (100K–355K LOC). Combines Steensgaard-style unification $O(n \cdot \alpha(n))$ with context-sensitivity and heap cloning to achieve precision comparable to Andersen. Linux kernel analyzed in 3.1 seconds, $< 46\text{MB}$ memory.

Huang et al. 2023 JARVIS (9/10) Critical for Python call graph construction. Function Type Graph (FTG) provides flow-sensitive type inference with strong updates. C3 linearization for MRO handles Python’s multiple inheritance. 84% higher precision than flow-insensitive approaches.

Spath et al. 2019 SPDS (9/10) Solves combined context+field sensitivity via synchronized pushdown systems. Key insight: encode field access patterns via pushdown automata, avoiding

exponential access-path enumeration. Achieves 64–83× speedups on DaCapo benchmarks.

Conrado et al. 2025 MCFL (9/10) First decidable analysis for interleaved Dyck reachability via Multiple Context-Free Languages. The d-MCFL hierarchy provides principled underapproximations with $O(n^{2d})$ complexity and SETH lower bounds proving tightness.

Muller et al. 2016 Viper (9/10) Verification infrastructure for permission-based reasoning. Provides permission-native intermediate language (IVL) with first-class support for `acc(e.f)`, fractional/symbolic permissions, magic wands ($A \multimap B$), quantified permissions.

Mulder et al. 2022 Diaframe (9/10) Automated Iris proofs with foundational soundness. Key innovation: goal-directed proof search inspired by linear logic programming. Bi-abduction with POSTPONED existentials (critical for invariants). 10× less manual proof than raw Iris.

Furr & Foster 2008 (8/10) Practical FFI type inference for OCaml-C and Java-JNI boundaries. Introduces *representational types* that model C’s low-level view of high-level data. GC safety analysis via effect annotations. Validated: found 24 errors in OCaml benchmarks, 156 errors in JNI benchmarks.

Patterson & Ahmed 2022 (9/10) Foundational framework for semantic soundness of language interoperability via compilation. Key insight: interoperability works by compiling both languages to a shared target, with *realizability models* $\mathcal{V}[\![\tau]\!]$ defining what target terms “behave as” source type τ .

Leroy 2009 CompCert (10/10) First realistic verified compiler proving semantics preservation for C to assembly. **Critical** for analysis correctness: if source analysis proves safety AND compilation is verified, then executable is safe. The 42K lines Coq proof introduces simulation relations as the standard proof technique.

Note: Yamaguchi 2014 (CPG), Steensgaard 1996, Reynolds 2002, and Leijen 2014 from Collection 2 were already integrated into the main synthesis and appear above with their original priorities.

B. Language Configuration Table

This appendix provides a reference table of safety guarantees provided by each supported language’s type system and runtime.

Table B.1: Language Safety Properties

| Language | MemSafe | NullSafe | TypeSafe | RaceFree | LeakFree | Memory | Types |
|------------|---------|----------|----------|----------|----------|--------|---------|
| Rust | ✓ | ✓ | ✓ | ✓ | ✓ | Owned | Static |
| Haskell | ✓ | ✓ | ✓ | ✓ | ✓ | GC | Static |
| Java | ✓ | × | ✓ | × | ✓ | GC | Static |
| Go | ✓ | × | ✓ | × | ✓ | GC | Static |
| Python | ✓ | × | × | ✓* | ✓ | GC | Dynamic |
| JavaScript | ✓ | × | × | ✓* | ✓ | GC | Dynamic |
| C | × | × | × | × | × | Manual | Static |
| C++ | × | × | × | × | × | Manual | Static |
| Swift | ✓ | ✓ | ✓ | × | ✓ | RC | Static |

*Single-threaded/GIL

Legend.

- **MemSafe:** Memory safety (no buffer overflows, use-after-free)
- **NullSafe:** Null safety (no null pointer dereferences by construction)
- **TypeSafe:** Type safety (no type confusion at runtime)
- **RaceFree:** Data race freedom (no concurrent access to shared mutable state)
- **LeakFree:** Resource leak prevention (automatic cleanup)

- **Memory:** Memory management model (Owned/GC/Manual/RC)
- **Types:** Type system (Static/Dynamic)

C. Glossary

This appendix provides definitions for key terms used throughout the synthesis document.

C.1 Core Concepts

Abstract Interpretation Computing sound approximations of program behavior.

Abstracting Gradual Typing (AGT) Garcia 2016. Framework showing gradual types are abstract interpretations of static type sets. $\gamma(?)$ = all types. Consistency = non-empty set intersection. Provides Galois connection foundation for gradual typing. See Section 9.1.2.

Affine Typing Substructural type system requiring values be used *at most once* (can drop without use). Rust uses affine, not strictly linear. See RustBelt/Iris.

Backsubstitution DeepPoly technique (Singh 2019). Recursively substitute polyhedral constraints backward through network layers until only input variables remain. Enables relational reasoning without exponential constraint explosion. Key to DeepPoly’s precision. See Section 2.1.7c.

Bi-Abduction Calcagno 2009. Given p and q , find anti-frame M and frame F such that $p * M \vdash q * F$. Enables compositional shape analysis.

Binary Session Type Honda 1998. Type discipline for structured two-party communication. Types describe sequences of send/receive actions, selections/branches, and delegation. Duality principle ensures compatible communication. See Section 14.0.

Bounded Unrolling Under-approximate loop analysis. Unroll k times (typically $k = 3$), then cut off. Sound for bug finding.

Camera Step-indexed partial commutative monoid (from Iris).

CFL-Reachability Graph reachability where valid paths form a context-free language.

Causality Analysis Honda 2008. Analysis of dependency relations (II, IO, OO) between communication prefixes. Determines ordering constraints for safe multiparty interaction.

Channel Effect Effect type for communication operations (ESEND, ERECV, ECHANCREATE, ECHANCLOSE). Linear effects that must be accounted for exactly once.

Chopped Symbolic Execution (CSE) Trabish 2018. Symbolic execution that skips user-specified code regions and recovers their effects lazily when observable. Reduces path explosion while preserving soundness. See Section 4.4.6.

Code Property Graph (CPG) Unified representation combining AST, CFG, and PDG.

Coherence Honda 2008. Property that a global type is linear and projectable to all participants. Ensures protocol is implementable.

CompCert Leroy 2009. First realistic verified compiler for C (Clight subset) to PowerPC/ARM/x86 assembly. Proves semantic preservation. 42K lines Coq. See Section 12.26.C.

Completeness Flag DSA flag indicating whether all pointers to a node have been discovered. See Section 5.2.5.2, 12.33.2.

Concolic Execution Godefroid 2005 (DART), Sen 2005 (CUTE). Hybrid execution that runs concrete and symbolic execution *simultaneously*. Key insight: when symbolic reasoning becomes intractable, substitute concrete values and continue. Term “concolic” = concrete + symbolic. See Section 4.4.5, 4.4.7.

Consistent Subtyping Garcia 2016. $G_1 \lesssim G_2$ iff $\exists T_1 \in \gamma(G_1), T_2 \in \gamma(G_2)$ with $T_1 <: T_2$. Combines gradual consistency with subtyping for record types. Essential for JS objects, TS interfaces at boundaries. See Section 9.1.2.

Convertibility (Language Interop) Patterson & Ahmed 2022. Relation $\tau_A \sim \tau_B$ between types from different languages meaning they can safely interoperate. Established by providing target-level *glue code*. See Section 11.2.

Code2Inv Si et al. 2018 (NeurIPS). RL-based loop invariant synthesis using GNN program embeddings and TreeLSTM decoder. Discovers complex invariants that widening cannot find. See Section 2.1.5c, 3.1.8.

C.2 Data Structures and Algorithms

CUTE (Concolic Unit Testing Engine) Sen, Marinov, Agha 2005. Foundational concolic testing tool introducing: (1) *logical input map* decoupling memory structure from physical layout, (2) *constraint separation* solving pointer constraints separately from arithmetic, (3) *fast unsat check* eliminating 60–95% of SMT calls, (4) *incremental solving*. See Section 4.4.5.

DART (Directed Automated Random Testing) Godefroid, Klarlund, Sen 2005 (PLDI). First tool combining concrete and symbolic execution. Three key techniques: automatic interface extraction, random test driver generation, dynamic test generation. See Section 4.4.5.

Data Structure Analysis (DSA) Lattner/Adve unification-based heap analysis with heap cloning for context sensitivity. See Section 5.2.5, 12.33.

DeepBugs Pradel & Sen 2018. ML-based bug detection using semantic identifier embeddings. Key insight: names encode semantic intent that reveals bugs. See Section 3.1.8.1.

DeepPoly Singh et al. 2019 (POPL). Abstract domain for neural network verification. Combines restricted polyhedral constraints with backsubstitution for tight bounds. 100–1000× faster than MILP. See Section 2.1.7c.

Dependent Load Trabish 2018. A load instruction where the loaded address may have been modified by a skipped function. Triggers recovery in CSE.

Diaframe Mulder 2022. Automated Iris proof framework using bi-abduction with postponed existentials. 10× less manual proof than raw Iris. See Section 7.4.10.

Divergence Analysis Detection of non-termination via repeating abstract states (loops) or same-state recursion.

DS Graph Lattner 2007. Data Structure Graph—representation used by DSA. Nodes represent disjoint sets of memory objects; edges are field-sensitive pointers. See Section 5.2.5.1.

Duality Honda 1998. Relationship between send and receive types: $\overline{T} = T$. If two endpoints have dual types, their communication is safe.

Eval Algorithm Calcagno 2009. Forward symbolic execution with bi-abduction for compositional memory analysis.

Evidence (AGT) Garcia 2016. Metadata tracking *how* type consistency was established during type checking. Enables precise blame tracking at boundaries. See Section 9.1.2.

Execution Tree Dynamic tree structure from symbolic execution. Computed on-demand from CPG.

C.3 Effects and Types

Effect Row Row-polymorphic type for tracking computational effects.

Expectation Transformer Cousot 2012. Abstract transformer for probabilistic programs that operates on probability distributions over abstract states. See Section 2.1.6c.

FFI Type Safety Furr & Foster 2008. Static analysis ensuring type-safe foreign function calls. Uses representational types and GC registration analysis. See Section 9.4.6.

Flow-Sensitive Type (FFI) Furr & Foster 2008. Type of form $ct\{B, I, T\}$ where B =boxedness, I =offset, T =tag value. See Section 9.4.6.

Four-Point Lattice Chong & Myers 2004. Security label lattice with both confidentiality ($C_{\text{Low}}/C_{\text{High}}$) and integrity ($I_{\text{Low}}/I_{\text{High}}$). Required for robust declassification analysis. See Section 8.1.1, 12.34.1.

Galois Connection Abstraction-concretization pair with soundness guarantee.

Global Type Honda 2008. Bird’s-eye view description of multiparty protocol specifying all interactions. Notation: $p \rightarrow q : k(U).G$ (p sends U to q via channel k).

Gradual Guarantee Garcia 2016. Theorem: making types less precise (more ?) preserves program semantics but may add runtime checks. See Section 9.1.2.

Graph Neural Network (GNN) Neural network operating on graph structures via message passing. Used by Code2Inv for invariant synthesis. See Section 3.1.8.

Guiding Constraints Trabish 2018. Path constraints accumulated between snapshot creation and dependent state in CSE. $gc = dependent.pc - snapshot.pc$.

C.4 Memory and Ownership

Handle (MS-Wasm) Disselkoen 2019. Typed pointer in MS-Wasm consisting of 4-tuple (*base*, *offset*, *bound*, *isCorrupted*). See Section 7.7.2.

Heap Cloning DSA technique creating separate heap graphs per calling context for precision. See Section 5.2.5.4, 12.33.4.

Higher-Order Ghost State Jung 2018 (Iris). Ghost state where the content is itself an Iris proposition (iProp). Critical for encoding recursive protocols and impredicative invariants. See Section 7.1.1.

Hybrid Fuzzing QSYM 2018. Cooperative combination of coverage-guided fuzzing (fast, shallow) with concolic execution (slow, deep). See Section 4.4.7.

Linear Memory (WebAssembly) A contiguous, mutable array of raw bytes that forms WebAssembly’s memory model. See Section 7.7.1.

Linear Typing Substructural type system requiring values be used *exactly once*. Stricter than affine. See Girard 1987.

Macroscopic Unification Lattner 2007. DSA’s approach combining Steensgaard-style unification with context-sensitivity. “Macroscopic” because it operates at data-structure level. See Section 5.2.5.

Magic Wand Reynolds 2002, Muller 2016. Separating implication ($A \multimap B$): “providing A yields B ”. PSPACE-complete in general. See Section 7.4.8.

Marshalling (FFI) Converting data between source language representation and target language representation at FFI boundaries. See Section 9.4.4, 9.4.6.

MS-Wasm (Memory-Safe WebAssembly) Disselkoen et al. 2019. Backwards-compatible extension to WebAssembly that captures memory safety semantics. Introduces segment memory and handles. See Section 7.7.

Multiplicity Aliasing status of capability—**unique** (can free) vs **dup** (aliases may exist).

Owner-as-Dominator Boyapati 2003. Ownership type discipline where all heap paths from root to object x must pass through x ’s owner. Precursor to Rust’s ownership model. See Part VII.

Ownership Types Clarke 1998, Boyapati 2003. Type system associating each object with an owner. Foundation for Rust’s ownership semantics. See Section 7.1, Part VII.

C.5 Program Analysis

IFDS Interprocedural Finite Distributive Subset— $O(ED^3)$ dataflow algorithm. Requires *distributive* transfer functions.

IMM (Intermediate Memory Model) Podkopaev 2019. Declarative memory model serving as intermediate layer for compilation proofs. Reduces $O(n \cdot m)$ proofs to $O(n + m)$. See Section 6.5.9.

Incorrectness Logic (IL) O’Hearn 2020. Under-approximate logic for proving bugs exist. **Deprecated** in favor of OL—IL is incompatible with abstract interpretation.

Interleaved Dyck Language of strings where projections to two Dyck alphabets are both valid. Used for combined context+field sensitivity. **Undecidable**—requires MCFL or SPDS approximation. See Section 4.2.3.

ISL Triple Incorrectness Separation Logic triple $[p] C [q; exit]$. Under-approximate semantics for bug finding.

KLEE Cadar, Dunbar, Engler 2008 (OSDI). Symbolic execution engine for LLVM bitcode achieving high coverage on real programs. Tested all 89 GNU COREUTILS achieving 90%+ line coverage, found 56 bugs. See Section 4.4.4.

Local Completeness Bruni et al. 2023. Property $C_c^A(f)$ meaning abstract domain A is complete for transfer function f at specific input c . See Section 2.1.8b.

MCFL (Multiple Context-Free Language) Conrado 2025. Language class strictly between CFL and interleaved Dyck. d -MCFL has $O(n^{2d})$ reachability complexity. See Section 4.2.4.

May-Mod Analysis Flow-insensitive pointer analysis computing the set of memory locations a function may modify. Used in CSE to detect dependent loads. See Section 4.4.6.5.

Narrowing Cousot 1992. Operator that recovers precision after widening. If $y \sqsubseteq x$ then $y \sqsubseteq (x \nabla y) \sqsubseteq x$.

Native Instrumentation QSYM 2018. Binary analysis technique using Intel Pin to instrument only instructions touching symbolic data. See Section 4.4.7.

Neural Invariant Synthesis Si et al. 2018. RL-based generation of loop invariants using neural networks. See Section 2.1.5c, Code2Inv.

Occurrence Typing Tobin-Hochstadt 2008. Flow-sensitive type refinement via type predicates. Complements gradual typing. See Section 2.1.7b, 9.5, 12.3.5.

Optimistic Concolic Execution QSYM 2018. Concolic variant that trades solver-level soundness for speed. 10–100× faster than KLEE. See Section 4.4.7.

Outcome Logic (OL) Zilberstein 2023. Unified correctness/incorrectness framework. **Preferred** over IL—compatible with abstract interpretation, supports manifest error detection.

Over-Approximation $\alpha(\text{concrete}) \subseteq \text{abstract}$. “If safe, truly safe.” May have false positives.

Path Condition King 1976. Conjunction of constraints accumulated along symbolic execution path. Determines path feasibility.

PDG Program Dependence Graph—data and control dependencies.

Points-to Analysis Computing what each pointer may point to.

Precision Ordering (Gradual Types) Garcia 2016. $G_1 \leq G_2$ (G_1 more precise than G_2) iff $\gamma(G_1) \subseteq \gamma(G_2)$. See Section 9.1.2.

Probabilistic Abstract Domain Cousot 2012. Abstract domain for probabilistic programs. Operates on probability distributions over standard abstract domain A . See Section 2.1.6c.

SDG System Dependence Graph—PDG extended with interprocedural edges. Two-phase slicing required.

Separation Logic Logic with separating conjunction for heap reasoning.

SPDS (Synchronized Pushdown System) Spath 2019. Two synchronized pushdown automata encoding field and call stacks. Enables polynomial combined context+field sensitivity. See Section 4.2.5.

Symbolic Execution King 1976. Path-sensitive analysis tracking symbolic values and path conditions. Commutativity: $\text{instantiate}(\text{exec_symbolic}(P)) = \text{exec_concrete}(P)$.

Taint Analysis Tracking untrusted data from sources to sinks.

Thin Slicing Backward slice following only relevant dependencies.

Two-Phase Slicing Horwitz 1990. Phase 1 ascends (no param-out), Phase 2 descends (no call/param-in). Prevents spurious paths.

Under-Approximation $\text{abstract} \subseteq \text{concrete}$. “If bug, truly a bug.” May have false negatives.

Widening Operator that accelerates fixpoint convergence for infinite domains.

C.6 Memory Models

Capped Memory Lee 2020. PS 2.0 construction bounding realistic interference during certification. Enables value-range analysis and register promotion. See Section 6.5.7.

Promise (Memory Model) Kang 2017. Commitment by a thread to perform a future write. Key mechanism for preventing thin-air values in relaxed memory.

Promising Semantics Kang 2017/Lee 2020 memory model using thread-local certification to

prevent thin-air values. PS 2.0 uses capped memory for certification. See Section 6.5.7.

RMW Strength Podkopaev 2019. Classification of hardware RMW operations. POWER/ARMv7 have strong RMWs; ARMv8/RISC-V have weak RMWs. See Section 6.5.9.

Simulation Relation Leroy 2009. Binary relation between execution states used to prove compilation pass correctness. Variants: lock-step, plus, star. See Section 12.26.C.

Thin-Air Problem Fundamental soundness bug in C11 axiomatic memory model. Permits values that appear without any legitimate source. Fixed by Promising Semantics. See Section 6.5.7, Theorem 12.26.3.

Thread-Local Certification Kang 2017/Lee 2020. Mechanism in Promising Semantics requiring that a thread can fulfill all its promises. See Section 6.5.7.

View (Memory Model) Kang 2017. Timemap tracking which messages a thread has observed per location. See Section 6.5.7.

View Shift Jung 2018 (Iris). The update modality $\Rightarrow P$ asserting ownership of resources that can be updated to satisfy P . See Section 7.1.1.

C.7 Security

Declassification Controlled release of secret data to lower security levels. See Section 8.1.4.3, 12.34.

Delimited Release Chong & Myers 2004. Declassification policy specifying *what* information may be released via escape hatches. See Section 8.1.4.3, 12.34.2.

Endorsement Chong & Myers 2004. The integrity dual of declassification: controlled acceptance of low-integrity data as high-integrity. See Section 8.1.4.3, 12.34.7.

Escape Hatch Declassification mechanism with policy, condition, and escaper components. See Section 12.34.2.

Integrity Label Chong & Myers 2004. Security label component tracking whether adversary can *influence* a value. $I_{\text{Low}} = \text{untrusted}$, $I_{\text{High}} = \text{trusted}$. See Section 8.1.1, 12.34.1.

Robust Declassification Zdancewic/Myers 2001 criterion ensuring attackers cannot influence what gets declassified. See Section 12.34.5–6.

Sanitizer Security operation that transforms a *value* to make it safe while preserving the security *label*. Contrast with declassification which changes the *label*. See Section 8.1.2, 12.34.9.

Semantic Security (under Declassification) Chong & Myers 2004. Formal security property: for all low-equivalent memories M_1 and M_2 , $\text{visible}(P(M_1))$ and $\text{visible}(P(M_2))$ are equivalent. See Section 8.1.4.3, 12.34.4.

C.8 Session Types

Input-Input Dependency (II) Honda 2008. Ordering between two inputs at same participant in same session.

Input-Output Dependency (IO) Honda 2008. Ordering where output depends on data from prior input at same participant.

Output-Output Dependency (OO) Honda 2008. Ordering between two outputs from same sender to same channel.

Projection (Session Types) Honda 2008. Operation $G \upharpoonright p$ extracting participant p 's local type from global type G .

Session Delegation Honda 1998. Transferring channel capability through a channel. Enables dynamic protocol reconfiguration.

Session Fidelity Honda 2008. Property that process communication follows declared session type.

Session Type Honda 1998/2008 protocol type specifying channel communication structure. See Section 12.28, Part XIV.

C.9 Testing and Verification

Concrete Fallback QSYM 2018. Strategy of executing complex operations concretely rather than symbolically. See Section 4.4.7.

Latent Error Bug that requires specific calling context to trigger. Contrast with Manifest Error.

Latent Predicate Tobin-Hochstadt 2008. Type proposition attached to a function result. Specifies what type test the function “remembers”. See Section 2.1.7b.

Manifest Error Le 2022. Bug that triggers regardless of calling context. True Positives Property guarantees 0% false positive rate.

Name Embedding Pradel 2018. Dense vector representation of identifier names capturing semantic meaning. See Section 3.1.8.1.

Recovery Caching Trabish 2018. Optimization storing recovery results to avoid redundant recovery executions in CSE.

Recovery State Trabish 2018. State created to lazily execute a skipped function when its side effects become relevant. See Section 4.4.6.3.

Semantics Preservation Leroy 2009. Property that compiled code has same observable behaviors as source program. See Section 12.26.C.

Skip Region Trabish 2018. User-specified function to exclude from symbolic exploration in CSE. See Section 4.4.6.2.

Snapshot State Trabish 2018. Symbolic state cloned immediately before entering skip region. See Section 4.4.6.3.

Synthetic Bug Generation Pradel 2018. Technique for generating ML training data by simple mutations of correct code. See Section 3.1.8.1.

Trilean Three-valued logic result from SMT queries: DEFINITELY, DEFINITELYNOT, or UNKNOWN.

True Positives Property Le 2022, Theorem 3.4. Manifest error implies dead code OR real bug exists. Guarantees 0% false positive rate.

Type Consistency Siek 2006, Garcia 2016. Relation $G_1 \sim G_2$ for gradual types. Reflexive and symmetric but *not* transitive. AGT derivation: $G_1 \sim G_2$ iff $\gamma(G_1) \cap \gamma(G_2) \neq \emptyset$. See Section 9.1.2.

Type Narrowing Tobin-Hochstadt 2008. Reducing a union type to a subset of its members based on type tests. See Section 2.1.7b, 9.5.3.

Unified Approximation Bruni et al. 2023 (LCL_A). Framework combining over-approximation and under-approximation in a single parameterized proof system. See Section 2.1.8b.

Validation-Based Soundness QSYM 2018. Soundness model where a validator filters false positives by concrete execution. See Section 4.4.7.

Verified Compilation Leroy 2009. Compiler accompanied by machine-checked proof of semantics preservation. See Section 12.26.C.

Viper Muller 2016. Verification infrastructure for permission-based reasoning with fractional permissions, magic wands, and quantified permissions. See Section 7.1.2, 7.4.8–7.4.10.

Visible Predicate Tobin-Hochstadt 2008. Type proposition known to hold at current program point. See Section 2.1.7b.

C.10 WebAssembly

Segment Memory (MS-Wasm) Disselkoen 2019. Memory region in MS-Wasm separate from Wasm’s linear memory. Segments are bounded, typed, and accessible only via handles. See Section 7.7.2.

WebAssembly (Wasm) W3C Standard. Platform-independent bytecode language designed to run C/C++ and similar languages at near-native speed. Provides isolation from host environment but *not* memory safety within the sandbox. See Section 7.7.

WebAssembly Type Safety The property that WebAssembly’s typed stack and typed function signatures are preserved at runtime. Wasm is type-safe for its value types but *not* for memory operations. See Section 7.7.1.

C.11 Pointer Analysis (TVLA and SVF)

Canonical Abstraction TVLA. Merge nodes with same canonical name (tuple of abstraction predicate values). Ensures bounded domain.

Chi Annotation (χ) Memory SSA. Annotates a store statement with potential memory locations that may be modified. For $*p = x$, $o = \chi(o)$ indicates location o may be updated.

Coerce Operation TVLA. Apply compatibility constraints ($\varphi_1 \Rightarrow \varphi_2$), sharpen indefinite predicates, detect inconsistencies.

Conversion Strategy M&F 2007. Decouples type from conversion behavior. Examples: type-directed, zero-for-error, null-for-none.

Embedding Theorem TVLA Theorem 3.7. If $S \sqsubseteq^f S'$ then $\llbracket \varphi \rrbracket_S^3(Z) \sqsubseteq \llbracket \varphi \rrbracket_{S'}^3(f \circ Z)$. Links concrete to abstract soundly.

Focus Operation TVLA. Split summary nodes until formula evaluates to definite (0 or 1). Enables materialization for loops.

Fold/Unfold Viper 2016. Explicit operations for recursive predicates. Fold exchanges body permissions for predicate instance; unfold exchanges predicate for body permissions.

Footprint Reynolds 2002. Memory locations accessed by a command (reads, writes, allocates, frees). Essential for frame rule.

Frame Rule Reynolds 2002. $\{P\} C \{Q\} \implies \{P * R\} C \{Q * R\}$. Enables compositional analysis by preserving unrelated resources.

Guard Polarity M&F 2007. Positive = value entering stricter language (must check). Negative = from stricter (can skip). *Flips* at function arguments.

Hole Tagging M&F 2007. Tag evaluation holes with expected language to prevent “language bleeding” during reduction.

Inductive System Aiken 1999. Finite representation of all set constraint solutions via cascade $\alpha_i = L_i \cup (\beta_i \cap U_i)$.

Inhale/Exhale Muller 2016 (Viper). Asymmetric assertion [*on_exhale*, *on_inhale*]: different behavior when checked (exhale) vs assumed (inhale). Used for leak checks. See Section 7.4.10.

Instrumentation Predicates TVLA. Derived predicates (reachability, cycle membership, sharing) that dramatically improve precision.

Local Consistency Cousot 1977. $\alpha(f_c(\gamma(x))) \sqsubseteq f_a(x)$. Abstract transfer over-approximates concrete transfer after abstraction.

Memory SSA Chow 1996, Sui 2016. Extension of classical SSA to handle indirect memory accesses via μ/χ annotations. Enables def-use chains for address-taken variables.

Mu Annotation (μ) Memory SSA. Annotates a load statement with potential memory locations that may provide the value. For $x = *p$, $\mu(o)$ indicates x may read from location o .

On-the-Fly CG Rupta 2024. Interleave call graph and points-to analysis. Resolves chicken-and-egg problem.

Projection Path Rupta 2024. Field representation (*base*, [f_1, f_2, f_3]) instead of index. More precise for nested structs.

Quantified Permission Muller 2016. Pointwise permission specification for arrays: $\forall i :: \text{range}(i) \Rightarrow \text{acc}(\text{arr}[i])$. See Section 7.4.9.

Realizability Model Patterson 2022. Semantic interpretation $\mathcal{V}[\tau] = \text{target values behaving as source type } \tau$.

Representational Type (FFI) Furr & Foster 2008. Type that models C’s low-level view of high-level language data. See Section 9.4.6.

Resource Algebra Algebraic structure for modeling ownership.

Semantic IR Patterson & Ahmed 2022. IR design philosophy where the IR serves as a

semantically-typed target language for multi-language interoperability. See Section 11.2.

Set Constraints Aiken 1999. Unified framework: $X \subseteq Y$ constraints with conditionals and constructors. $O(n^3)$ resolution.

Shape Analysis Track structural properties (list, tree, DAG, cycle) of heap data using three-valued logic.

Source-Sink Analysis Analysis pattern tracking data flow from allocation sites (sources) to deallocation sites (sinks). SVF is optimized for this pattern.

Stack Filtering Rupta 2024. *Novel*. Eliminate points-to targets for stack objects whose frames aren't on call stack. $2\text{--}5\times$ faster + more precise.

Strong Update Sui 2016. A store that *kills* the old value at a memory location. Occurs when pointer uniquely points to a concrete (non-summary) location.

Summary Node TVLA. Abstract node representing multiple concrete nodes. $sm(v) = \frac{1}{2}$ indicates summary.

SVFG (Sparse Value-Flow Graph) Sui 2016. Graph where nodes are variable definitions and edges capture value-flow relationships.

Three-Valued Logic TVLA. Values: 1 (true), 0 (false), $\frac{1}{2}$ (unknown). Information ordering: $\frac{1}{2} \sqsubseteq 0$ and $\frac{1}{2} \sqsubseteq 1$.

Weak Update Sui 2016. A store that must *preserve* old values. Occurs when pointer may point to multiple locations or target is a summary node.

C.12 ZIPPER-Related Terms

ZIPPER Li 2020 selective context sensitivity identifying precision-critical methods via flow patterns. See Section 5.3.2, 12.30.

Precision-Critical Method (PCM) ZIPPER. A method that participates in direct, wrapped, or unwrapped flow patterns. Only PCMs benefit from context sensitivity.

In Method ZIPPER. A method of class C with one or more parameters. Objects can flow *into* the class through In method parameters.

Out Method ZIPPER. A method of class C with non-void return type. Objects can flow *out* of the class through Out method returns.

Direct Flow ZIPPER. Flow pattern where object enters via In method parameter, flows through assignments/field ops, exits via Out method return of same class.

Wrapped Flow ZIPPER. Flow pattern where object enters via In method, gets stored in wrapper object, wrapper flows out via Out method.

Unwrapped Flow ZIPPER. Flow pattern where carrier object enters via In method, contents are loaded from carrier, loaded contents flow out via Out method.

Precision Flow Graph (PFG) ZIPPER. Graph extending OFG with wrap/unwrap edges. Built per-class, restricted to nodes reachable from In method parameters.

Zipper_e (Express) ZIPPER. Variant with efficiency threshold. Excludes classes where $pts_e > PV \times total_pts$. Default $PV = 5\%$ achieves 94.7% precision with $25.5\times$ speedup.

C.13 JARVIS-Related Terms

Function Type Graph (FTG) JARVIS 2023. Per-function graph tracking type relations for Python call graph construction. Enables flow-sensitive type inference with strong updates. See Section 5.3.3.

Application-Centered Analysis JARVIS 2023. Analysis mode that starts from application entry points and only analyzes reachable code. Critical for scalability on Python codebases with 200+ dependencies.

Method Resolution Order (MRO) Python's algorithm for determining method lookup order in class hierarchies. Uses C3 linearization to handle multiple inheritance consistently. See Section 5.3.3.3.

C3 Linearization Python's MRO algorithm. Merges parent class MROs while preserving lo-

cal precedence and monotonicity. Ensures consistent method lookup in diamond inheritance patterns.

Duck Typing Dynamic typing paradigm where object capabilities determine validity, not declared type. Requires type inference (FTG) rather than class-based dispatch for precise call graph construction.

Magic Method Python special methods (`__getattr__`, `__call__`, `__get__`, etc.) that intercept attribute access and invocation. Must be handled by call graph construction. See Section 5.3.3.4.

Decorator Resolution JARVIS 2023. Tracking function wrapping via decorators. Call graph must trace through wrapper to original function.

Import Summary JARVIS 2023. Pre-computed mapping of import statements to resolved types.

Class Summary JARVIS 2023. Pre-computed class hierarchy and method containment. Includes inheritance relations and cached MRO computations.

PyCG Prior state-of-the-art Python call graph tool. Uses flow-insensitive worklist algorithm. JARVIS achieves 84% higher precision.

D. Gap Analysis, Theoretical Tensions, and Engineering Considerations

This appendix provides a systematic analysis of identified gaps between the theoretical framework established in Parts I–XII and the engineering requirements for production deployment.

D.1 Executive Summary

This appendix derives from:

1. **External Review:** Feedback from static analysis practitioners
2. **Literature Cross-Reference:** Comparison with extended paper collection
3. **Implementation Experience:** Lessons from prototype development

D.1.1 Gap Classification

Identified gaps are classified into three severity levels:

Table D.1: Gap Severity Classification

| Severity | Definition | Resolution Approach |
|--------------------|---------------------------|-------------------------------|
| Critical | Blocks core functionality | Must resolve before Phase 3 |
| Significant | Reduces analysis quality | Should resolve before Phase 5 |
| Minor | Engineering convenience | Address in Phase 6 or defer |

D.1.2 Gap Summary Matrix

D.1.3 Theoretical Tensions

Section D.10 documents tensions between foundational papers that require explicit resolution in the implementation:

- IFDS distributivity requirements vs general dataflow (Aiken 1999)
- Separation logic assumptions vs garbage-collected languages (Reynolds 2002)
- Field indexing vs projection paths (Rupta 2024)
- Language-agnostic vs language-specific pointer analysis
- Two-valued vs three-valued logic (TVLA 2002)
- Type-directed vs arbitrary conversions (Matthews-Findler 2007)

Table D.2: Gap Analysis Summary

| Gap | Severity | Status | Resolution |
|--------------------------|-------------|-------------|--------------------|
| Library Modeling | Significant | Addressed | D.2 (bi-abduction) |
| Call Graph Construction | Critical | ✓ Resolved | D.3 (Qilin, 5.3) |
| Path Sensitivity | Significant | ✓ Resolved | D.4 (4.3, 4.4) |
| Memory Layout / ABI | Significant | Partial | D.5 (IR ext.) |
| Dynamic Code (eval) | Minor | Partial | D.6 (conservative) |
| Build System Integration | Minor | Engineering | D.7 |
| False Positive Mgmt | Critical | ✓ Resolved | D.8 (OL class.) |
| Time Budgets | Significant | Addressed | D.9 |

D.2 Gap 1: Library and Environment Modeling

Severity: Significant **Status:** Addressed

D.2.1 Problem Characterization

The theoretical framework assumes complete source code availability. Production deployments must handle dependencies on:

- **Standard Libraries:** libc, JDK runtime, Python stdlib, Go runtime
- **Frameworks:** React, Django, Spring Boot, Express, Rails
- **Third-Party Packages:** npm, pip, cargo crates, Maven artifacts

Without semantic models for external code, analysis chains degrade:

- Taint propagation terminates at library boundaries
- Pointer analysis returns \top for library allocations
- Call graphs omit edges through unmodeled code

D.2.2 Reassessment: Problem Scope is Narrower Than Initially Estimated

Critical Observation: Modern package ecosystems distribute source code by default, substantially reducing the scope of the “library modeling problem.”

Table D.3: Ecosystem Source Code Availability Analysis

| Ecosystem | Source Availability |
|------------------|--|
| Python / PyPI | Source distributed by default; <code>pip -download-source</code> |
| JavaScript / npm | Full source in <code>node_modules/</code> |
| Rust / Cargo | Source fetched to <code>~/.cargo/registry/src/</code> |
| Go / Modules | <code>go mod download</code> retrieves full source |
| Java / Maven | <code>-sources.jar</code> artifacts available for most packages |
| C/C++ | Exception: <code>.so/.dll</code> without headers common |

Revised Problem Statement: The library modeling challenge reduces to:

1. **Scalability:** Whole-program analysis of transitive dependency closure—addressed via summary-based compositional analysis (bi-abduction)
2. **Closed Binaries:** Rare except in C/C++ ecosystem—addressed via conservative approximation + user-provided contracts
3. **External Boundaries:** Network APIs, databases, system calls—addressed via contract specification language

D.2.3 Resolution: Formal Specification Inference Techniques

The following established techniques address library modeling without requiring machine learning or manual annotation:

Technique 1: Bi-Abduction for Specification Inference

Reference: Calcagno 2009 “Compositional Shape Analysis by Means of Bi-Abduction”

Given partial specification $\{P\}$ *code* $\{Q\}$, bi-abduction infers:

- **Anti-frame:** Additional preconditions P' required for safety
- **Frame:** Additional postconditions Q' established by code

Application: Automatically infer specifications for library functions by analyzing their implementations.

Integration: Phase 4, Milestone 4.3 (Bi-abduction for specs)

Technique 2: Modular Typestate Verification

Reference: Bierhoff 2007 “Modular Typestate Checking of Aliased Objects”

Modules verified independently against declared contracts:

- Libraries publish pre/post conditions
- Callers verify against published contracts
- Compositional verification without whole-program analysis

Application: Standard library contracts enable modular analysis

Technique 3: Refinement Types as Contracts

References: Rondon 2008 “Liquid Types”, Vazou 2014 “Refinement Types for Haskell”

Type signatures encode behavioral contracts directly:

$$read_file : \{f : \text{String} \mid valid_path(f)\} \rightarrow \{s : \text{String} \mid length(s) \geq 0\}$$

Application: Extract contracts from existing type annotations (TypeScript types, Python type hints, Rust traits)

Technique 4: Dependent Types

Reference: Xi 1999 “Dependent Types in Practical Programming”

Types indexed by values encode rich specifications:

$$\begin{aligned} vector &: (n : \text{Nat}) \rightarrow \text{Type} \\ append &: vector\ n \rightarrow vector\ m \rightarrow vector\ (n + m) \end{aligned}$$

Application: Leverage F* dependent types for verified specification

D.2.4 Strategy for Genuinely Opaque Code

For unavailable source code (closed binaries, external network APIs, proprietary libraries), the following tiered strategy applies:

Opaque Code Handling Strategy

Tier 1: Conservative Approximation (Default)

- Return value: \top (top of lattice)
- Effect set: $\{\text{Read}, \text{Write}, \text{IO}, \text{Alloc}, \text{Free}, \text{Throw}\}$
- Taint propagation: $tainted(input) \Rightarrow tainted(output)$
- Pointer analysis: fresh allocation site per call

Tier 2: User-Provided Contracts

- ACSL-style annotations for C/C++
- JML-style specifications for Java

- Brrr-contract language (to be specified)
- Inline annotations: `@taint_source`, `@taint_sink`, `@pure`, etc.

Tier 3: API Schema Extraction

- OpenAPI/Swagger → REST endpoint contracts
- Protocol Buffers/gRPC → RPC type contracts
- GraphQL schemas → query/mutation contracts
- Database schemas → SQL query contracts

Tier 4: Uncertainty Tracking

- Classify findings crossing opaque boundaries as INCOMPLETE
- Propagate uncertainty through dependent analyses
- Report boundary crossings in SARIF output

D.2.5 Remaining Implementation Items

Table D.4: Library Modeling Implementation Items

| Item | Priority | Target Phase |
|---|----------|--------------|
| Contract specification language design | High | Phase 4 |
| Automatic contract extraction from type annotations | Medium | Phase 4 |
| Contract composition rules for multi-language | Medium | Phase 5 |
| Uncertainty propagation through analysis pipeline | High | Phase 3 |
| Standard library contract database (Python, JS, Go) | Low | Phase 6 |

D.3 Gap 2: Call Graph Construction — Addressed

Status: Addressed in Sections 5.3, 12.22

Solution Implemented:

- Section 5.3: On-the-fly call graph construction (Qilin algorithm)
- Section 12.22: Stack filtering theorems (Rupta/Li 2024)
- Interleaved CG + pointer analysis resolves chicken-and-egg

Key Insights from Rupta 2024:

- 1-callsite sensitivity better than Andersen for Rust
- Stack filtering: 2–5× faster *and* more precise
- On-the-fly: resolve virtual calls as points-to sets refine

D.3.1 Remaining Open Questions

1. **Reflection handling**—`Class.forName()`, eval-like constructs
2. **Dynamic dispatch prediction** heuristics for unsolvable cases
3. **Incremental CG update** when code changes
4. **Language-specific resolution rules** (Python MRO, JS prototype chain)

D.4 Gap 3: Path Sensitivity — Addressed

Status: Addressed in Sections 4.3 (Under-Approx) and 4.4 (Symbolic Exec)

Solution Implemented:

- Section 4.3: Eval algorithm with path-sensitive bi-abduction
- Section 4.4: Full symbolic execution with path conditions
- Section 4.3.3: Hybrid IFDS + Eval/Symbolic architecture

- Section 12.15: Symbolic execution theorems (King 1976)

Key Theorems:

- Commutativity: $instantiate(exec_symbolic(P)) = exec_concrete(P)$
- Forking semantics for conditionals
- SMT integration with trilean (DEFINITELY/DEFINITELYNOT/UNKNOWN)

D.4.1 What’s Still Missing

1. **SMT solver integration**—Z3, CVC5 for constraint solving
2. **Path merging heuristics**—When to merge vs. keep separate
3. **Widening for symbolic paths**—Bound path explosion
4. **Integration with IFDS**—Seamless hybrid approach

D.5 Gap 4: Memory Layout and ABI

D.5.1 The Problem Statement

The synthesis IR defines `Int` and `Struct` but ignores:

- Struct padding and alignment
- Integer width differences (`long`: 32-bit Windows, 64-bit Linux)
- Endianness
- Calling conventions

Listing D.1: Example Bug Not Caught

```
// C code (Windows x64)
struct Data { int x; long y; }; // sizeof = 8 (long is 4 bytes)

// Rust code (Windows x64)
#[repr(C)]
struct Data { x: i32, y: i64 } // sizeof = 16 (i64 is 8 bytes)

// FFI call passes wrong struct layout → buffer overflow
```

D.5.2 Proposed IR Extension

To detect ABI mismatches at FFI boundaries, the IR must be extended with explicit layout and platform information. The key insight is that the same high-level type (e.g., `long`) may have different physical representations on different platforms (4 bytes on Windows x64, 8 bytes on Linux x64). The following type definitions capture the necessary metadata:

Extended IR types with layout information

```
(* Endianness affects multi-byte value interpretation *)
type endian =
  | BigEndian
  | LittleEndian

(* Physical memory layout for a type *)
type layout_info = {
  size : nat;           (* Size in bytes *)
  alignment : nat;      (* Alignment requirement *)
  endianness : endian;  (* Big or little *)
}

(* Platform-specific ABI configuration *)
type target_abi = {
  pointer_size : nat;    (* 4 or 8 bytes *)
  int_sizes : map int_width nat; (* Width-specific sizes *)
  struct_packing : packing_mode;
  calling_convention : calling_conv;
```

```

}

type packing_mode =
| PackDefault      (* Platform default padding *)
| PackPacked       (* No padding, #pragma pack(1) *)
| PackExplicit     (* Explicit padding fields *)

(* Extended struct type with layout metadata *)
type ir_struct = {
  name : string;
  fields : list (string * ir_type * offset); (* Include byte offset *)
  layout : layout_info;
  abi : target_abi;
}

(* Function to detect ABI incompatibilities between source and target *)
val check_abi_compatibility :
  source_struct:ir_struct →
  target_struct:ir_struct →
  list abi_mismatch

(* Specific types of ABI mismatches that can cause memory corruption *)
type abi_mismatch =
| SizeMismatch : expected:nat → actual:nat → abi_mismatch
| AlignmentMismatch : field:string → expected:nat → actual:nat → abi_mismatch
| OffsetMismatch : field:string → expected:nat → actual:nat → abi_mismatch
| TypeWidthMismatch : field:string → expected:nat → actual:nat → abi_mismatch

```

The `check_abi_compatibility` function compares two struct representations across a boundary. Each `abi_mismatch` variant identifies a specific problem: `SizeMismatch` means the total struct size differs, `OffsetMismatch` means a field starts at different positions (likely due to padding differences), and `TypeWidthMismatch` means the same field name has different sizes (e.g., long on different platforms).

D.5.3 What’s Still Missing

1. **ABI database** for common platforms (Windows, Linux, macOS × x86, x64, ARM)
2. **Struct layout calculation** algorithm per ABI
3. **FFI marshaling verification** at boundaries
4. **Endianness conversion tracking** for network protocols

D.6 Gap 5: Dynamic Code Generation (Eval)

D.6.1 The Problem Statement

The synthesis recognizes `eval()` as a taint sink, but not as a program modifier:

Listing D.2: `eval()` as Program Modifier

```

# This isn't just a sink -- it CHANGES the program
user_method = input()
eval(f"obj.{user_method}()") # Creates new call edge at runtime!

```

D.6.2 Conservative Strategy

Handling Eval Without String Analysis

1. Detection:

- Flag all `eval/exec/Function()` calls
- Report as “dynamic code generation” finding

2. Taint Tracking:

- If `eval` argument is tainted → HIGH severity (RCE)

- If eval argument is constant \rightarrow analyze the constant
- 3. Constant Propagation:**
- If `eval("x.foo()")` where string is known \rightarrow add call edge
 - Use existing constant propagation analysis
- 4. Over-Approximation:**
- If string unknown: assume eval can call ANY method
 - Add edges to all methods (conservative)
 - Mark these edges as “speculative”
- 5. User Annotation:**
- Allow `@eval_targets("foo", "bar")` annotation
 - Restrict analysis to specified targets

D.6.3 What’s Still Missing

1. **String constraint solving** for eval argument analysis
2. **Dynamic call graph edges** marked as uncertain
3. **Reflection API modeling** (`Class.forName`, `getattr`, etc.)
4. **Code generation patterns** (template engines, ORMs)

D.7 Gap 6: Build System Integration

D.7.1 The Problem Statement

Cannot correctly parse code without build configuration:

- **C/C++:** `-DDEBUG`, `-I/include/path` affect preprocessing
- **Java:** `CLASSPATH` determines which classes are visible
- **TypeScript:** `tsconfig.json` controls module resolution
- **Python:** `PYTHONPATH`, `virtualenv` affect imports

D.7.2 This is Engineering, Not Research

No papers directly address this because it’s an engineering concern.

Existing Solutions:

- `compile_commands.json` for C/C++ (CMake, Bear, intercept-build)
- Language Server Protocol (LSP) provides project model
- Build system plugins (Bazel aspects, Gradle plugins)

D.7.3 Practical Implementation

Build System Integration Strategy

C/C++ Require `compile_commands.json`. Parse with clang’s libtooling. Each TU gets its own preprocessor state.

Java Read `pom.xml` / `build.gradle`. Extract classpath. Use javac API or ECJ.

TypeScript Parse `tsconfig.json`. Use TypeScript compiler API. Respect module resolution.

Python Detect `virtualenv` / `conda`. Parse `pyproject.toml` / `setup.py`. Use `ast` module with correct `sys.path`.

Rust Parse `Cargo.toml`. Use `rustc` or `rust-analyzer`. Respect feature flags.

Go Use `go/packages` API. Respects `go.mod` automatically.

D.7.4 What’s Still Missing

1. **Unified project model** abstraction in brrr-machine
2. **Build system detection** heuristics

3. **Incremental build integration**—trigger re-analysis on build
4. **Cross-project analysis** for monorepos

D.8 Gap 7: False Positive Management

False positive management uses the provable **Manifest/Latent classification** from Le 2022.

- **Manifest bugs** have *zero* false positive rate by Theorem 3.4
- **Latent bugs** require specific calling context (context provided)
- **Incomplete** classifications include explicit reason for incompleteness

See Section 12.3 for the complete formal foundation (ISL triples, `classify_bug` algorithm, theorems).

See Section 13.4 Layer 6 for system integration.

D.8.1 Implementation Status

Table D.5: False Positive Management Implementation Status

| Component | Status | Section |
|-------------------------------------|------------------|---------|
| ISL Triple type | ✓ Defined | 12.3 |
| Manifest proof structure | ✓ Defined | 12.3 |
| True Positives Property | ✓ Theorem stated | 12.3 |
| <code>classify_bug</code> algorithm | ✓ Implemented | 12.3 |
| Witness generation (concolic) | □ TODO | 12.15 |

D.9 Gap 8: Extended Confidence Sources (Implementation Details)

Implementation Details for Section 13.4 Layer 6 (Verification & Confidence)

Core types are in Section 12.3; this section provides algorithms & detail.

D.9.1 Test Discrepancy: Compare test assumptions vs code behavior. Match → confidence BOOST | Contradict → confidence REDUCE | None → NEUTRAL

D.9.2 Uncertainty Estimation: Each analysis block outputs confidence. Propagation rules and adaptive precision algorithms.

D.9.3 Runtime Debugger: Ground truth from actual execution. Confirms → 0.95+ | Contradicts → 0.0 (false positive!) | Never → unchanged

D.9.1 Test vs Code Discrepancy

The Insight

Tests encode programmer *assumptions* about code behavior. Code encodes *actual* behavior. These often diverge. Detecting discrepancies *statically* (without running tests) reveals:

1. **Test bugs**—Tests that would never catch the bugs they’re meant to catch
2. **Code bugs**—Code that behaves differently than any test expects
3. **Specification gaps**—Behaviors neither tested nor clearly intended

Detection Categories

The following F* type definitions model different categories of test/code discrepancies. Each constructor captures a specific way that test assumptions can diverge from actual code behavior, enabling automated detection of test quality issues.

Test/Code Discrepancy Types

```
(* Note: 'side_effect' is used instead of 'effect' which is reserved in F* *)
type side_effect =
  | SideRead
  | SideWrite
  | SideIO

type io_category =
  | FileIO
  | NetworkIO
  | DatabaseIO
  | FFIBinding
  | SystemCall

type test_code_discrepancy =
  (* Test assumes input type that code doesn't actually accept *)
  | InputTypeMismatch : test_input:ir_type → code_accepts:ir_type
    → test_code_discrepancy
  (* Test assertion can never fail given actual code behavior *)
  | UnreachableAssertion : assertion:node_id → reason:string
    → test_code_discrepancy
  (* Test assertion can never PASS given actual code behavior *)
  | ImpossibleAssertion : assertion:node_id → reason:string
    → test_code_discrepancy
  (* Code has error path that no test exercises *)
  | UntestedErrorPath : error_node:node_id → condition:ir_expr
    → test_code_discrepancy
  (* Mock returns values code can never actually produce *)
  | MockBehaviorMismatch : mock:node_id → impossible_value:ir_expr
    → test_code_discrepancy
  (* Test expects side effect that code doesn't perform (or vice versa) *)
  | SideEffectMismatch : expected:side_effect → actual:option side_effect
    → test_code_discrepancy
  (* File/network/binding behavior differs *)
  | IOBehaviorMismatch : io_type:io_category → test_assumes:behavior
    → code_does:behavior → test_code_discrepancy
```

The key insight is that each discrepancy type provides actionable information: `InputTypeMismatch` indicates the test is using inputs the function will reject, while `UnreachableAssertion` means the test is checking something that can never fail (a useless test).

Language-Specific Test Framework Integration

- **Rust:** `#[test]`, `#[should_panic]`, `assert!`, `assert_eq!`
→ Parse test functions, extract assertions, compare with code paths
- **Go:** `func Test*`, `t.Error`, `t.Fatal`, testify assertions
→ Extract test table patterns, compare with actual branches
- **Python:** `pytest`, `unittest`, `assert` statements
→ Parse fixtures, mocks, `parametrize` decorators
- **JS/TS:** `jest`, `mocha`, `expect()`, `describe/it` blocks
→ Extract mock implementations, compare with real modules

D.9.2 Classification Composition (Provable, Not Heuristic)

The Principle

Every analysis block must output not just a *result* but also:

1. **Classification**—Manifest, Latent, or Incomplete (not a float!)
2. **Complexity**—How expensive was this? (provable bound)
3. **Incompleteness Reasons**—Explicit list if analysis is incomplete

Classification Composition Rules

Classification Composition (Formal, Not Heuristic)

Old (Heuristic): $conf_out = conf_1 \times conf_2$ or $conf_out = \min(conf_1, conf_2)$

Problem: What does $0.7 \times 0.8 = 0.56$ mean semantically?

New (Formal): Classification is preserved through composition

Rule 1: Incomplete propagates

If ANY dependency is Incomplete \rightarrow result is Incomplete

Reason: Can't prove manifest if inputs uncertain

Rule 2: Manifest preserved through complete analysis

Manifest finding + Complete analysis \rightarrow Manifest

Reason: True Positives Property still applies

Rule 3: Latent context accumulates

$LATENT(ctx_1) + LATENT(ctx_2) \rightarrow LATENT(ctx_1 \wedge ctx_2)$

Reason: Both contexts required to trigger

Rule 4: Manifest + Latent = Latent

$MANIFEST \circ LATENT(ctx) \rightarrow LATENT(ctx)$

Reason: The latent constraint dominates

The following F* implementation formalizes the classification composition rules. The key insight is that `completeness` is tracked separately from `bug_classification`, and incomplete results always propagate conservatively. This differs from heuristic confidence scores (e.g., $0.7 \times 0.8 = 0.56$) which have no clear semantic interpretation.

Formal Composition of Classifications

```
(* Completeness tracks whether an analysis is definitive or approximated *)
type completeness =
  | Complete (* No approximations used *)
  | Incomplete of incompleteness_reason (* Explicit reason for approximation *)

(* Type signature: compose two bug classifications with their completeness *)
val compose_classifications :
  upstream:bug_classification → upstream_complete:completeness →
  downstream:bug_classification → downstream_complete:completeness →
  bug_classification

let compose_classifications up up_c down down_c =
  (* Rule 1: Incomplete propagates *)
  match up_c, down_c with
  | Incomplete r1, _ →
    (* Upstream incomplete - can't trust downstream *)
    RelaxedManifest [IncompleteDependency r1]
  | _, Incomplete r2 →
    (* Downstream incomplete *)
    RelaxedManifest [IncompleteDependency r2]
  | Complete, Complete →
    (* Both complete - apply composition rules *)
    match up, down with
    (* Rule 2: Manifest preserved *)
    | Manifest p1, Manifest p2 →
      Manifest (combine_proofs p1 p2)
    (* Rule 3: Latent contexts accumulate *)
    | Latent ctx1, Latent ctx2 →
      Latent (ctx1 'conj' ctx2)
    (* Rule 4: Manifest + Latent = Latent *)
    | Manifest _, Latent ctx → Latent ctx
    | Latent ctx, Manifest _ → Latent ctx
    (* RelaxedManifest composes conservatively *)
    | RelaxedManifest vs1, RelaxedManifest vs2 →
```

```

    RelaxedManifest (vs1 @ vs2)
  | RelaxedManifest vs, _ → RelaxedManifest vs
  | _, RelaxedManifest vs → RelaxedManifest vs

```

The function preserves the True Positives Property (Le 2022, Theorem 3.4): if a **Manifest** bug is found through complete analyses, it is guaranteed to be a real bug.

Uncertainty Sources

Every analysis operation that introduces approximation or encounters missing information must explicitly record the uncertainty source. This enables: (1) explaining to users why a finding has reduced confidence, (2) adaptive precision—retrying with more precise analysis when uncertainty is high, and (3) audit trails for verification.

Uncertainty Source Types

```

type uncertainty_source =
  (* Analysis approximations *)
  | WideningApplied : iterations:nat → uncertainty_source
  | ContextDepthLimited : max_k:nat → uncertainty_source
  | HeapAbstractionMerged : merged_count:nat → uncertainty_source
  (* Missing information *)
  | ExternalCodeUnmodeled : func:func_id → uncertainty_source
  | DynamicDispatchUnresolved : call_site:node_id → uncertainty_source
  | ReflectionOrEval : location:node_id → uncertainty_source
  (* Language weirdness *)
  | UndefinedBehaviorPossible : ub_type:string → uncertainty_source
  | LanguageSpecContradiction : spec_section:string → uncertainty_source
  | PlatformDependent : varies_on:string → uncertainty_source

  (* Every analysis result bundles value with metadata about how it was computed *)
type analysis_result (a : Type) = {
  value : a;
  confidence : float; (* LEGACY: For backwards compatibility only. *)
                      (* NEW CODE should use bug_classification from Section 12.3 *)
  complexity : complexity_class;
  uncertainties : list uncertainty_source;
  approximations_used : list approximation;
}

(* Complexity classification for time budget management *)
type complexity_class =
  | Linear : complexity_class
  | NLogN : complexity_class
  | Quadratic : complexity_class
  | Cubic : complexity_class
  | CubicInDomain : domain_size:nat → complexity_class (* IFDS:  $O(ED^3)$  *)
  | Exponential : base:nat → complexity_class

```

The `analysis_result` record separates the computed value from its provenance metadata. Note that the `confidence : float` field is marked as legacy; new code should use the structured `bug_classification` type from Section 12.3 which provides provable guarantees rather than heuristic scores.

Adaptive Precision Based on Uncertainty

Listing D.3: Adaptive Precision Algorithm

```

IF uncertainty > threshold THEN
  TRY more precise analysis:
    - Increase context depth
    - Disable widening (if bounded iterations possible)
    - Use relational domain instead of non-relational
    - Run symbolic execution on uncertain paths

```

```

IF complexity > budget THEN
  DEGRADE precision:
    - Reduce context depth
    - Apply widening earlier
    - Use cheaper domain (Steensgaard vs Andersen)
    - Skip shape analysis
    - Mark as "partial analysis"

```

D.9.3 Runtime Debugger Integration

The Vision

Build debugger-like plugins for each target language that:

1. **Intercept** execution at configurable points
2. **Capture** concrete values, types, and execution paths
3. **Feed** this ground truth back to static analysis
4. **Validate** static predictions against runtime reality

Architecture

Table D.6: Language Runtime Hooks

| Language | Hook Mechanism |
|----------|---------------------------|
| Python | <code>sys.settrace</code> |
| Node | V8 inspector |
| Rust | LLDB/rr |
| Go | delve DAP |
| C/C++ | GDB/LLDB |

Trace Collector captures:

- Function entry/exit with arguments
- Variable values at key points
- Branch decisions taken
- Memory allocations/deallocations
- I/O operations performed

Trace Analyzer classifies traces:

- ALWAYS HAPPENS: seen in 100% of runs
- COMMON: seen in >50% of runs
- RARE: seen in <10% of runs
- NEVER SEEN: predicted by static, never observed

Static ↔ Dynamic Reconciliation:

- Static predicted X , runtime showed Y :
 - If $Y \subset X$: static is sound (expected)
 - If $Y \not\subset X$: static MISSED something (bug in us!)
 - If $X \gg Y$: static too imprecise (optimize)

Trace Classification

The trace classification system bridges static and dynamic analysis by recording how often execution paths are observed at runtime. This enables: (1) identifying false positives when static predictions are never observed, (2) finding soundness bugs when runtime discovers paths static analysis missed, and (3) precision tuning when static analysis is too conservative.

Trace Classification Types

```

(* Frequency categories based on observed execution runs *)
type trace_frequency =
| AlwaysHappens      (* 100% of observed runs *)
| VeryCommon        (* >90% of runs *)
| Common             (* >50% of runs *)
| Occasional         (* 10-50% of runs *)
| Rare               (* <10% of runs *)
| NeverObserved      (* 0% - but may be possible *)

(* Complete classification of a trace with static/dynamic comparison *)
type trace_classification = {
  path : list node_id;          (* Sequence of CPG nodes in the trace *)
  frequency : trace_frequency;
  observed_count : nat;
  total_runs : nat;
  static_says : reachability;    (* What static analysis predicted *)
  discrepancy : option discrepancy_type; (* Any mismatch detected *)
}

(* Types of disagreement between static and dynamic analysis *)
type discrepancy_type =
| StaticMissed : trace_classification → discrepancy_type (* Runtime found path static
  ↪ missed *)
| StaticOverApprox : ratio:float → discrepancy_type (* Static predicted 100, only 5
  ↪ taken *)
| StaticWrong : expected:value → actual:value → discrepancy_type (* Value mismatch *)

```

The discrepancy field is critical for analysis quality: `StaticMissed` indicates a soundness bug in our analysis (we missed a real path), while `StaticOverApprox` indicates an opportunity for precision improvement.

Integration with Confidence Model

Runtime traces BOOST or REDUCE confidence:

- Finding X with static confidence 0.7:
 - If runtime confirms: confidence $\rightarrow 0.95+$
 - If runtime contradicts: confidence $\rightarrow 0.0$ (false positive!)
 - If runtime never exercises: confidence unchanged
- Path Y predicted reachable:
 - If runtime reaches: `CONFIRMED`
 - If runtime never reaches after N runs: likely `UNREACHABLE` (but could be rare edge case)

D.10 Theoretical Tensions and Resolutions

This section documents tensions between foundational papers that require explicit reconciliation in the implementation. Each tension is characterized by its source, implications, and adopted resolution.

D.10.1 IFDS Distributivity vs General Dataflow (Aiken 1999)

Tension: IFDS Scope Limitation

Synthesis Position (Part IV):

IFDS serves as the primary interprocedural dataflow framework

Contradicting Reference:

Aiken 1999 “Introduction to Set Constraint-Based Program Analysis”

IFDS is a *restricted fragment* of the more general set constraint formalism

Theoretical Implications:

- IFDS requires *distributive* transfer functions: $f(a \sqcup b) = f(a) \sqcup f(b)$

- Pointer analysis (Part V) is *not* distributive—cannot use IFDS directly
- Set constraints handle non-distributive cases via projection operations

Adopted Resolution:

- **IFDS:** Efficient $O(ED^3)$ implementation for distributive analyses (taint, nullability, reaching definitions, live variables)
- **Set Constraints:** General framework for non-distributive analyses (pointer analysis, type inference)
- Section 12.18: Provides set constraint theorems for non-distributive cases

Implementation Impact:

- Phase 2: Pointer analysis uses dedicated solver (not IFDS)
- Phase 3: IFDS for taint/nullability only

D.10.2 Separation Logic vs Garbage-Collected Languages (Reynolds 2002)

Tension: Separation Logic Deallocation Assumptions

Synthesis Position (Part VII):

Use separation logic uniformly for memory reasoning across languages

Contradicting Reference:

Reynolds 2002 “Separation Logic: A Logic for Shared Mutable Data Structures”

Explicitly notes: “GC interaction is problematic”

Theoretical Implications:

- Separation logic frame rule assumes explicit deallocation timing
- GC may collect memory at unpredictable points
- “Disconnected garbage” has no separation logic representation
- Ownership transfer semantics differ fundamentally

Adopted Resolution:

For GC Languages (Python, Go, Java, JavaScript, Ruby):

- Separation logic applies to *resources* only: file handles, network connections, database cursors, locks, semaphores, condition variables
- Memory safety granted axiomatically from runtime
- Track resource ownership, not heap cell ownership

For Manual Memory Languages (C, C++, unsafe Rust):

- Full separation logic reasoning applies
- Frame rule enables compositional analysis
- Track allocation/deallocation correspondence

Implementation Impact:

Language configuration specifies `memory_model`: GC | Manual | Hybrid

D.10.3 Field Index vs Projection Path (Rupta 2024)

Tension:

- **Synthesis Position (Part V):** Use field *index* for field sensitivity:

`FieldLoad(dst, base, field_index)`

- **Rupta 2024:** Use *projection path* for nested structs:

`(base, [field1, field2, field3])`

Implications:

- Field index: `x.a` and `y.a` conflated if same index
- Projection path: distinguished by *full* path
- Nested structs lose precision with index-based approach

Resolution: Adopt projection-based representation for Rust analysis. For simpler languages, index-based may suffice. Update Part V Section 5.1 to note both approaches.

D.10.4 Steensgaard Default vs Language-Specific (Rupta 2024)

Tension:

- **Synthesis Position (Part V):** “Steensgaard for speed, Andersen for precision”
- **Rupta 2024:** “1-callsite-sensitive is FASTER and MORE PRECISE for Rust”

Implications:

- Generic recommendation ignores language characteristics
- Rust ownership model changes aliasing patterns
- Stack filtering (novel) enables context-sensitivity at low cost

Resolution: Add language-specific recommendations:

- **Rust:** 1-callsite + stack filtering (Rupta)
- **C/C++:** Andersen or demand-driven (Sridharan)
- **Python:** Type-based + dynamic traces
- **Java:** Object-sensitive for OOP patterns

D.10.5 Two-Valued vs Three-Valued Logic (TVLA 2002)

Tension:

- **Synthesis Position (Part II):** Boolean lattices with explicit “Maybe” variants:
TaintLevel = Tainted | Untainted | Unknown
- **TVLA 2002:** Three-valued logic with Kleene semantics:
three_value = TV0 | TV1 | TVHalf

Implications:

- Synthesis approach is ad-hoc per-domain
- TVLA approach is principled with information ordering
- Kleene semantics provide correct conjunction/disjunction

Resolution: Section 12.23 provides three-valued foundation. Section 5.4 uses three-valued for shape analysis. Consider refactoring existing domains to use three-valued base.

D.10.6 Type-Directed vs Arbitrary Conversion (M&F 2007)

Tension:

- **Synthesis Position (Part IX):** Assume type-directed conversion:
 $type_map : Type_1 \rightarrow Type_2$
- **M&F 2007:** Conversion strategies (κ) decouple type from behavior:
Zero-for-error, sentinel values, custom conversions

Implications:

- Real FFIs have arbitrary conventions (C’s -1 for error, etc.)
- Type-directed assumption breaks for C/Python interop
- Need explicit conversion strategy abstraction

Resolution: Section 12.20 adds `conversion_strategy` type. Section 9.3 integrates with guard generation. Part IX should note non-type-directed cases.

D.10.7 GC vs Manual Memory (VeriFFI 2025)

Tension: The framework supports both GC languages (Python, Java, Go) and manual memory languages (C, C++, Rust). Their ownership models conflict:

Implications:

- GC languages don’t track deallocation—GC handles it

Table D.7: GC vs Manual Memory Comparison

| Aspect | GC Languages | Manual Memory |
|--------------------|-----------------------------|-------------------------|
| Deallocation | Automatic, timing uncertain | Explicit, deterministic |
| Pointer validity | Until collected | Until freed |
| Object movement | GC may relocate | Fixed address |
| Ownership transfer | Reference semantics | Move/copy semantics |

- Manual languages must track use-after-free, double-free
- FFI boundaries are especially dangerous:
 - GC may collect object still referenced by C code
 - C may free memory still visible to GC language
- Ownership states don’t translate directly

Resolution Strategy:

1. **Language-specific ownership states** (Section 7.6.2):
 - GC languages: GCROOTED / GCREACHABLE / GCFINALIZED
 - Manual: ACQUIRED / INUSE / RELEASED (Section 7.1)
2. **FFI boundary requirements** (Section 7.6.3):
 - Pin objects during cross-boundary calls
 - Copy small data to avoid cross-heap references
 - Register pointers as roots when necessary
3. **GC-isomorphism preservation** (Section 7.6.1):
 - Representation predicates survive GC cycles
 - Analysis results valid before AND after GC
 - Addresses may change but structure preserved
4. **Boundary guards:**
 - Source GC, target non-GC: Pin or Copy
 - Source non-GC, target GC: Register root if storing reference
 - Both GC but different runtimes: Cross-heap reference handling

Reference: VeriFFI (Wang et al. 2025)—“Verified FFI for GC Languages”

D.10.8 Summary Table

D.11 Collection 2 Paper Integration Summary

This section documents the integration of Collection 2 papers into the synthesis.

D.11.1 Papers Already Integrated in Main Synthesis

| Paper | Integration Location | Key Contribution |
|-----------------------|---|--|
| Leijen 2014 | Section 12.27 (Effect Absence Theorems) | Row-polymorphic effects, semantic soundness theorems. Effect absence proves bug impossibility. |
| Yamaguchi 2014 | Part II (CPG), Section 5.3, throughout | CPG (AST+CFG+PDG), traversal algebra. Found 18 Linux kernel CVEs. Foundation of analysis infrastructure. |

Continued on next page

| Paper | Integration Location | Key Contribution |
|------------------------------|---|---|
| Reynolds 2002 | Section 7.1 (Ownership), Part VII | Separation logic, frame rule, compositional heap reasoning. Enables modular memory analysis. |
| Steensgaard 1996 | Section 5.1, Section 12.22 | $O(n \cdot \alpha)$ unification-based points-to, cjoin optimization. Fast pointer analysis baseline. |
| Sivaramakrishnan 2021 | Section 6.1.2, 6.4, 6.4.1 | Production effect handlers in OCaml. Fiber-based implementation, 1% overhead, C FFI limitations. |
| Petricek 2014 | Section 6.6 (Coeffects), Part VII intro | Coeffect systems dual to effects. Flat (capabilities) and structural (liveness, usage). Connects to linear types. |

D.11.2 Papers Added to Appendix A (Priority Matrix)

| Paper | Priority | Category | Key Contribution | Gap Addressed |
|-----------------------|----------|--------------|--|---------------------------|
| Bierhoff 2007 | 9 | Ownership | 5 access permissions, modular typestate with aliasing | D.1, D.7 |
| Siek 2006 | 8 | Types | Gradual typing, non-transitive consistency, cast insertion | D.5, Section 9.1.2 |
| Kang 2017 | 10 | Memory Model | Promising Semantics—prevents thin-air, validates compiler opts | Theorem 12.26.3 soundness |
| Lee 2020 | 10 | Memory Model | Promising 2.0—capped memory, value-range analysis, ARMv8 fix | Section 6.5.7 |
| Podkopaev 2019 | 10 | Memory Model | IMM— $O(n + m)$ compilation proofs, 33K Coq | Section 6.5.9 |
| Godefroid 2005 | 8 | Testing | DART—concolic execution origins, interface extraction | Section 4.4.5 |

D.11.3 Integration Details

Bierhoff 2007 (Modular Typestate Checking).

- **Enhances:** Part VII (Ownership and Resources), Section 7.1.3

Table D.8: Theoretical Tensions Summary

| Tension | Synthesis | Paper | Resolution |
|-----------------|--------------------|----------------------|--|
| IFDS scope | Primary frame-work | Restricted frag-ment | Both: IFDS for distributive, constraints for general |
| Sep logic + GC | Use everywhere | GC problematic | Resources only for GC languages |
| Field handling | Index-based | Projection path | Adopt projection for precision |
| Pointer default | Steensgaard | Language-specific | Per-language recommendations |
| Value logic | Two-valued + Maybe | Three-valued | Use three-valued foundation |
| Conversions | Type-directed | Arbitrary strategies | Add conversion strategy type |
| GC vs Manual | Unified ownership | Incompatible models | Language-specific states + FFI requirements |

- **Addresses:** Gap D.1 by providing formal aliasing handling for library specifications
- **Key insight:** Permission taxonomy with fraction *functions* (not just fractions) enables precise tracking of shared state
- **Temporary state:** Weak permissions carry state that is forgotten after effects, enabling practical use of share/pure

Siek 2006 (Gradual Typing).

- **Enhances:** Part IX (Multi-Language Analysis), Section 9.1.2 (boundary risk analysis)
- **Critical fix:** Replaces informal “types_compatible” with rigorous *type consistency*
- **Non-transitivity:** $t_1 \sim ?$ and $? \sim t_2$ does *not* imply $t_1 \sim t_2$. This prevents unsound coercion chains.
- **Cast insertion:** Systematic algorithm for generating boundary casts from Section 9.1

Garcia 2016 (Abstracting Gradual Typing).

- **Enhances:** Part IX (Section 9.1.2), connects to Part II (Abstract Interpretation)
- **Critical insight:** Gradual types *are* abstract interpretations of static type sets
- **Galois connection:** $\gamma(?) = \text{all types}$, $\gamma(\text{Int}) = \{\text{Int}\}$. Consistency is: $\gamma(G_1) \cap \gamma(G_2) \neq \emptyset$
- **Derived non-transitivity:** Non-transitivity *emerges* from set intersection (not stipulated)
- **Consistent subtyping:** $G_1 \lesssim G_2$ when $\exists T_1 \in \gamma(G_1), T_2 \in \gamma(G_2)$ with $T_1 <: T_2$. Essential for record types.
- **Evidence semantics:** Tracks *how* consistency was established for precise blame at boundaries
- **Gradual guarantee:** Less precise types \rightarrow more runtime checks, same semantics
- **Integration:** Connects Section 9.1.2 to Section 2.1.2, unifying gradual typing with abstract interpretation framework

Sivaramakrishnan 2021 (Retrofitting Effect Handlers onto OCaml).

- **Enhances:** Section 6.1.2 (Algebraic Effects), Section 6.4 (Effect Handler Limitations)
- **New section:** Section 6.4.1 (Production Effect Handler Implementation)

- **Key contributions:**
 - Fiber-based stack segmentation for delimited continuations
 - 1% mean overhead for non-effect code (validated on 54 benchmarks)
 - C FFI boundary limitation: effects cannot cross C stack frames
 - Runtime-enforced continuation linearity (one-shot by default)
 - DWARF compatibility for debugging (GDB, perf work correctly)
- **Practical insight:** Effect handlers *are* practical for production systems when carefully implemented

Petricek 2014 (Coeffects: A Calculus of Context-Dependent Computation).

- **Enhances:** Part VI (complete effect/coeffect duality), Part VII (ownership via usage coeffects)
- **New section:** Section 6.6 (Coeffect Systems)
- **Key contributions:**
 - Effects and coeffects are *dual*: effects = what computation *produces*, coeffects = what it *requires*
 - Flat coeffects for capabilities (network, filesystem, platform requirements)
 - Structural coeffects for per-variable properties (liveness, usage counting)
 - Semiring algebra enables compositional reasoning
 - Usage coeffects connect to linear types and Rust ownership
- **Critical insight:** Complete analysis requires *both* effects and coeffects

D.11.4 Cross-Paper Connections

Both Bierhoff 2007 and Siek 2006 address partial knowledge:

- **Siek:** Types are partially known (? represents unknown portions)
- **Garcia (AGT):** ? represents *set* of all types; partial knowledge = set abstraction
- **Bierhoff:** State is partially known (weak permissions have temporary assumptions)
Effect/Coeffect duality connects multiple papers:
- **Petricek 2014:** Coeffects are dual to effects (Moggi 1991, Plotkin 2003)
- **Sivaramakrishnan 2021:** Production implementation validates effect handler theory
- **Girard 1987:** Linear types map to usage coeffects (Section 6.6.3 and Section 7.2.2)
AGT connects gradual typing to abstract interpretation (Part II \leftrightarrow Part IX):
- **Garcia 2016:** Gradual types form Galois connection with static type sets
- **Consistency derived:** $G_1 \sim G_2$ iff $\gamma(G_1) \cap \gamma(G_2) \neq \emptyset$
- **Precision ordering:** G_1 more precise than G_2 iff $\gamma(G_1) \subseteq \gamma(G_2)$
- **Implication:** Part IX boundary analysis is an *instance* of Part II framework
This suggests a unified treatment of uncertainty:

Partial Knowledge Framework

- **For types (Siek/Garcia):** consistency = non-transitive (from set intersection)
- **For states (Bierhoff):** assumption = temporary (may be invalidated)
- **For permissions:** fractions = summable (ownership accounting)
- **For gradual (AGT):** precision = set inclusion (Galois connection)

The integration enables the synthesis to handle:

1. **Dynamic language analysis** (Python, JavaScript) via gradual typing
2. **Aliased resource tracking** via access permissions
3. **Modular library verification** via frame-based inheritance

4. **Precise boundary checking** via type consistency (not subtyping)

End of Appendix D