École Polytechnique Fédérale de Lausanne

Developing a Query Engine for Source Code Analyzers

by Gabriel Fleischer

Master Thesis

EPFL IC IINFCOM LAMP1
INR 319 (Bâtiment INR)
Station 14
CH-1015 Lausanne

March 7, 2025

Acknowledgments

I extend my sincere gratitude to Alban Auzeill for his consistent availability and insightful discussions, which significantly contributed to the project's progression. I also express my appreciation to the entire JVM team at SonarSource for their invaluable support during my internship. In particular, I acknowledge Alban and Marco for paving the way on this topic.
My deepest thanks go to Anna Herlihy, whose insight and academic expertise were crucial in ensuring the readability of this paper.
Finally, I am grateful to Professor Odersky for supervising this project. His involvement was indispensable in making this work possible.

Lausanne, March 7, 2025 Gabriel Fleischer

Abstract

Static analysis plays a critical role in modern software development by automating the detection of bugs, security vulnerabilities, and code style violations. Rule-based systems are commonly used in static analysis tools, with each rule targeting a specific type of issue. However, selecting relevant nodes in the Abstract Syntax Tree (AST) efficiently remains a challenge.

This paper introduces a framework that enhances rule-based static analysis tools by adopting a descriptive approach to AST traversal and node selection. The framework provides a type-safe API that allows developers to express complex AST traversals through a series of operations. It optimizes the selection logic to balance the overhead associated with the added abstraction layer.

An evaluation conducted using SonarSource's Java analyzer[10] illustrates the framework's potential to simplify rule definitions while maintaining performance comparable to existing systems. The framework's descriptive approach enhances the readability and maintainability of static analysis rules, making them easier to understand, debug, and evolve.

This work contributes to the development of more declarative and user-friendly static analysis tools, ultimately promoting the creation of higher-quality software.

Contents

Chapter 1

Introduction

Static analysis (section 2.1) plays a crucial role in modern software development. Automating the detection of bugs, security vulnerabilities, and code style violations is essential for ensuring a production-ready system[16]. This paper focuses on static analysis tools that employ rule-based systems, where each rule corresponds to a specific issue type and has a dedicated implementation. This approach is prevalent in tools integrated within Continuous Integration / Continuous Delivery (CI/CD) pipelines[15]. A central challenge in static analysis is the complexity of efficiently selecting relevant nodes in the Abstract Syntax Tree (AST). Effective tooling for tree navigation, in terms of both runtime performance and usability, is fundamental to building efficient static analysis tools.

A common approach to AST node selection is a combination of imperative programming and the use of the visitor pattern. However, this approach can lead to complex and difficult-to-maintain codebases. These systems often involve numerous visitor classes, each dedicated to a specific use case. The imperative nature of this approach makes it challenging to reason about the overall behavior of a feature, as the code tends to the coupling selection and traversal logic, the what and the how. This complexity can significantly impact the readability and maintainability of the codebase, hindering understanding, debugging, and evolution of the static analysis system.

To mitigate these issues, a common and successful approach in software development is to adopt descriptive paradigms (section 2.2). This paper proposes a descriptive API for AST traversal. However, while descriptive APIs improve usability, they often introduce layers of abstraction that can impact performance. Therefore, this paper explores solutions to achieve descriptiveness without sacrificing performance.

Two primary strategies exist for creating descriptive APIs for AST node selection. matchers (section 2.3) and Deductive queries (section 2.4). Matchers offer significant advantages in selecting structural patterns within the AST, but their expressiveness struggles to establish relationships between multiple elements (section 6.1). Conversely, deductive query systems abstract away tree traversal and focus solely on node relations. However, these systems are complex to integrate into larger codebases and have difficulty dealing with incremental analysis. (section 6.2)

This paper introduces a framework positioned between matcher-based and deductive query systems. The focus is existing visitor-based systems, and the objective is to offer enhanced expressiveness relative to matches while maintaining ease of integration and avoiding the overhead associated with deductive systems. This is achieved by focusing on single-file analysis. Limiting the framework to single-file analysis aligns with the common constraints of visitor-based static analysis tools. This design choice offers two key benefits: independent rule execution, facilitating data parallelism, and a lighter, more maintainable system.

The proposed framework models each node selection as a sequence of operations starting from the AST root object. It is, in fact, a general-purpose data pipeline construction tool. Developers define the operation sequence, and the framework optimizes the pipeline once, enabling lazy and repeated execution.

The optimization strategy focuses on eliminating redundant operations, which makes it particularly efficient for rule-based systems. In these systems, each rule selects a specific set of nodes based on a pattern. Each rule has its implementation, and redundancy is a common source of inefficiency. For instance, the repeated identification of string constants nodes across multiple rules is computed every time.

This approach perfectly fits the main problem as it allows developers to write data pipelines descriptively. They are also optimized once and executed once for each file, reducing the optimization overhead. Furthermore, the framework's architecture facilitates integration into existing codebases, as it is independent of the parser and the specific AST representation.

To evaluate the effectiveness of the proposed framework, this paper presents a case study using SonarSource's Java analyzer[10], a representative case of a visitor-based system. The framework will be integrated into this system, and a subset of existing rules will be migrated. The evaluation will focus on the following aspects:

- Correctness: Demonstrating that a tool using the framework produces the same outputs as the existing system.
- Usability: Assessing whether the framework simplifies the creation and maintenance of static analysis rules.
- Performance: Evaluating the performance of the framework, aiming for performance comparable to the current system and ideally surpassing it.

Chapter 2

Background

2.1 Abstract Syntax Tree and Static Analysis

An Abstract Syntax Tree (AST) represents the syntactic structure of source code as a tree. Each node in the AST corresponds to a construct in the source code, such as expressions, statements, or declarations. The relationships between these nodes reflect the hierarchical organization of the program.

In Java, an if-statement such as `if (x > 0) { System.out.println("positive"); }` could be represented in an AST this way[1]:

```
IfStatement
  |   condition:
  |- BinaryExpression(>)
  |     |- Identifier(x)
  |     |- Literal(0)
  |
  |   body:
  |- MethodInvocationTree(System.out.println)
        |   paramerter:
        |- Literal("positive")
```

The AST captures the essential structure: the conditional check, the action to perform if the condition holds, and their relationship.

Static analysis refers to the process of analyzing computer software without actually executing it, usually using the AST. This form of analysis examines the source code to identify potential issues, such as errors, vulnerabilities, and coding standard violations.

[1]This is not the real AST, but a simpler version to visualize the concept.

2.2 Descriptive Paradigm

In software development, a descriptive paradigm focuses on specifying what a program should achieve rather than explicitly detailing how to achieve it. This contrasts with imperative paradigms, which provide step-by-step instructions for the computer to execute.

Descriptive paradigms often employ declarative languages or frameworks that allow developers to express the desired outcome using high-level abstractions. The system then takes responsibility for determining the optimal execution strategy.

2.3 Matchers Systems

Matcher systems leverage matcher constructs applied to every node in the AST. Matchers define patterns to be searched in the AST, enabling complex conditions on nodes to easily find specific nodes. It is used by tools like Clang[13], Tree-sitter[14], and TASTy Query[7][2].

2.4 Deductive Query Systems

Deductive query systems offer powerful descriptive capabilities for rule definition. These systems typically define node selection using a Datalog-like deductive language. The execution occurs in two phases: first, the entire codebase is analyzed, and the necessary information is stored in a database. Then, the node selection is executed against this database to identify matching ones. A key advantage is that the entire codebase information is available, enabling advanced analyses like taint analysis or symbolic execution.

When selecting a specific node, there is no explicit tree traversal in the selection definition, as the focus is purely on the relationships between nodes. In the recursive call example above, visitor and matcher-based systems would need to traverse the method body, searching for calls to the parent method. In contrast, deductive systems simply look for the presence of a relationship between the method and the call. This is a major advantage that advanced analysis features enable. Therefore, deductive systems completely abstract away tree traversal, addressing the primary challenge of complex traversal logic. However, these systems have some drawbacks. For this paper, the most relevant are integration and single-file analysis.

[2]while TastyQuery uses pattern matching rather than matcher objects, the underlying concept is similar

Chapter 3

Design

This chapter details the design of the proposed framework, which aims to introduce descriptive node selection capabilities to existing static analysis tools that currently rely on imperative approaches. A core objective is to ensure seamless integration with any AST representations, allowing tools to retain their existing parsers and result processing pipelines while leveraging the framework for descriptive node selection. This approach mirrors successful projects in other domains, such as parallel computing with concurrent collections or reactive programming with ReactiveX[5]. These systems abstract away complexity by representing logic as a series of operations, resulting in user-friendly and easily reasoned APIs. My framework adopts a similar philosophy, viewing node selection as a sequence of operations on an AST object.

The framework is fundamentally a generalized data pipeline creation tool. It facilitates the construction of pipelines that accept a value as input, perform a series of operations, and produce multiple outputs. The framework's descriptive nature provides much information about the logic, enabling optimizations. The data pipelines are designed for single optimization and multiple uses, minimizing the total overhead. This design maximizes the framework's efficiency when the resulting pipelines are executed repeatedly.

Setting the AST object as the input aligns perfectly with the initial problem. We define the node selection by the operations required to traverse the tree and produce the desired nodes as output, and the resulting pipeline can be executed once per file. Furthermore, the multi-output capability is ideally suited for rule-based systems. We can view each rule as a selection of nodes for further processing, and the combination of all rules forms an abstraction with one input (the AST) and multiple outputs (the selected nodes). Static analysis tools like PMD[6] or SonarQube[9] are excellent candidates for integration with this framework. This paper focuses on the Java analyzer within SonarQube[10] as a representative case study, but we designed the framework's architecture to be independent of the parser and specific AST representation.

3.1 Architecture

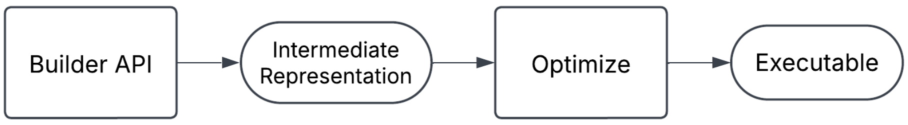

Figure 3.1: Architecture

This section details the architecture of the proposed framework, which comprises four key components: the Builder API, which creates an Intermediate Representation (IR) that is then Optimized to produce an Executable. (Figure 3.1)

- **The Builder API:** The Builder API provides the interface through which developers construct the data pipelines that define their static analysis rules. This step also constructs an underlying intermediate representation.
- **The Intermediate Representation:** The IR, generated by the Builder API, represents the pipeline's logic as a Directed Acyclic Graph (DAG). In this DAG, each node corresponds to an operation within the pipeline, and the edges represent the flow of data between operations (i.e., output-input relationships). The IR contains more information than is necessary for execution. We use this additional information to validate and optimize the pipeline during the Optimization step. The IR cannot be executed; it needs to go through the Optimization step first.
- **The Optimization:** The Optimization transforms the input IR into the final executable pipeline while applying transformations for correctness and efficiency. We designed the framework to support multiple executable forms and, thus, multiple optimizers, allowing developers to select the one that best suits their specific needs (or provide their own).
- **The Executable:** The Executable, the output of the Optimization stage, performs the actual execution of the data pipeline. The same executable can be used repeatedly with different input objects. Currently, two implementations of the Execution Engine are available, though the framework's design accommodates additional implementations.

3.2 The Builder API

The pipeline builder provides the primary interface through which developers interact with the framework. It is a type-safe API that adheres to the builder pattern. The users can describe the operations to apply to the pipeline. The following example demonstrates how to use the framework to create a rule that reports instances where a file contains more than three static imports.

```kotlin
fun createRule(manager: SelectionManager) {
  // Find all static imports
  manager.newSelection()
    .ofKind(TreeKind.IMPORT)
    .where { it.isStatic() }
    // Check if there are more than the allowed amount
    .where { it.count() > 3 }
    // Then report the found import node
    .report("There are more static imports in this file than the 3 allowed.")
}
```

Listing 3.1: Rule Example

The provided code snippet (Listing 3.1) illustrates the conciseness and readability afforded by the builder API. It clearly expresses the intent of the rule: identify imports, filter for static ones, and then check if the count exceeds a threshold. If the condition is met, the rule generates a report.

The operations are applied to an intermediate builder object. This object is parameterized with a generic type for the current output of the pipeline. This typed intermediate object can be passed as a parameter to subsequent operations (see Combine below), ensuring type safety and facilitating the construction of complex logic. This design allows for a fluent and expressive API while maintaining compile-time type guarantees.

3.2.1 Selector Types

The Builder API employs three distinct builder types to represent the number of produced elements: `SingleBuilder`, `OptionalBuilder`, and `ManyBuilder`. These correspond to a single element, zero or one element, and zero or more elements, respectively. This design offers two key benefits:

Operation Restrictions: Each selector type supports only a subset of available operations, preventing invalid ones such as calling `orElse` on a `SingleBuilder` or `first` [1] on an `OptionalBuilder`.

Type-Safe Operations: Some operations, such as `where` *(See Composed operations)*, accept selectors as arguments. Using distinct selector types ensures that these operations receive arguments of the correct cardinality. For example, `where` requires a `SingleBuilder` producing a boolean value.

[1]The `first` operation restricts a batch to one element at most, but if the batch is empty, it stays empty. Applying first on an `OptionalBuilder` has no effect.

3.2.2 Operations

Operations express all logic within the framework. We drew inspiration from functional collection operations to design them. A subset of operations, called the core, can express any logic within the framework. All other, more specialized operations are compositions of these core operations, which we use as building blocks *(See Composed operations)*. Each operation takes the output of the previous pipeline as its input. We provide the following core operations:

- **Map**: Applies a one-to-one transformation to the input element.
- **Filter**: Retains only the input element if it satisfies a given predicate.
- **FlatMap**: Applies a one-to-many transformation to each input element, flattening the resulting collections into multiple outputs.
- **orElse**: Applicable to `OptionalBuilder` [2], this operation provides a default value if the selector yields no results.
- **Aggregate**: Combines all input elements from the same batch*(See Batches)* into a single aggregated value.
- **AggregateDrop**: Same as `Aggregate`, but the result can be dropped. *See Drop variants below.*
- **Scope**: This operation is crucial for the framework's expressiveness. It partitions the input batch into sub-batches, associating each sub-batch with the specific value that generated it.
- **Combine**: Combines the outputs of two pipelines in a product-like fashion. When paired with `Scope`, this operation enables the expression of complex relationships between different pipelines, making it extremely powerful.
- **CombineDrop**: Same as `Combine`, but the result can be dropped. *See Drop variants below.*
- **Union**: Merges the results of multiple selectors into a single set of results.

Drop variants

Two core operations (`Aggregate` and `Combine`) have corresponding "Drop" variants. These variants function identically to their original counterparts, with the added capability of discarding resulting values. The functions associated with these variants produce a `Droppable` value, an interface with two subtypes: the `Keep(value)` class and the `Drop` singleton.

[2]And `ManyBuilder` in a minor way.

When the result is `Keep(value)`, the node propagates the associated value. Conversely, if the result is `Drop`, the node discards it. This mechanism provides a precise and controlled way to selectively eliminate values in operations that cannot be expressed by a `FlatMap` or a `Filter`.

Composed operations

The framework's foundation on extension functions facilitates the composition of new operations by reusing existing ones. A prime example is the `where` operation, which can be expressed as a combination of `Scope` and `CombineDrop` as shown in below (Listing 3.2).

```
pipeline.where { operations }

// Equivalent to :
pipeline.scope { filtered ->
    filtered.combineDrop(operations) { value, keep ->
        if (keep) Keep(value) else Drop
    }
}
```

Listing 3.2: Compose operation

The `where` operation filters elements based on another builder instead of a lambda. To guarantee that the resulting pipeline produces precisely one boolean value for each element in the `where` clause, it accepts a function that transforms a `SingleBuilder<T>` into a `SingleBuilder<Boolean>` and leverages the `Scope` operation to isolate the evaluation of the condition for each element.

This example highlights the power of composition to improve readability and simplify the construction of complex logic.

Batches

Within the framework, a batch is a collection of values. The size of a batch is tied to the builder type; a `SingleBuilder` will only pass batches containing a single value, while a `ManyBuilder` will pass batches with N values.

Initially, there is a single batch containing only the initial input value. The number of batches can only be altered by the `Scope` operation because it is the only operation that generates new batches. It does this by partitioning an existing batch into sub-batches.

Scope

The `Scope` operation creates sub-pipelines, each scoped to a specific value. It achieves this by generating a new batch for every input value. This isolation allows `Aggregate` and `Combine` operations to apply to smaller batches.

```
val firstReturn = newQuery()
  .ofKind(TreeKind.METHOD_DECLARATION)
  .scope { method ->
    method.subtree()
      .ofKind(TreeKind.RETURN_STATEMENT)
      .first()
  }

...
```

Listing 3.3: Scope example

The provided example (Listing 3.3) illustrates how `scope` limits the application of the `first` operation[3]. It selects all method declarations and then identifies the first return statement within each method rather than across all methods. Without the `scope`, this would have returned the first return statement of the whole AST.

Combine

The `Combine` operation is fundamental to the framework's expressive power. It includes both simple comparisons (e.g., `isEqualTo` and `isGreaterThan`) and more complex operations (e.g., `orElseUse`).

```
val methods = manager.newQuery()
  .ofKind(TreeKind.METHOD_DECLARATION)

val recursiveCall = methods
  .scope { method ->
    method.subtree()
      .ofKind(TreeKind.METHOD_INVOCATION)
      // Only keep calls with the same name as the method
      .where { invoc ->
        invoc.name().isEqualTo(method.name())
      }
  }

...
```

Listing 3.4: Combine usage

In the example above (Listing 3.4), the `isEqualsTo` operation within the `where` clause combines the names of the method invocation and the method declaration to check for equality. This

[3]The first operation is a specialized aggregate operation producing the first element of each batch.

comparison filters out any method invocations that are not recursive calls (i.e., calls to a method with the same name).

The `Combine` operation effectively creates a cartesian product of the outputs of two selectors within a common batch. Its true strength, however, is realized when used in conjunction with the `Scope` operation. Without `Scope`, the `Combine` operation's utility is limited, and the cartesian product can lead to an exponential increase in the number of elements. `Scope` restricts the combination to related elements within each batch, making `Combine` a powerful tool for expressing relationships between elements.

3.3 The Intermediate Representation

The pipeline builder constructs an intermediate representation (IR) that captures the whole logic of the defined pipeline. This IR is a Directed Acyclic Graph (DAG), where each node represents an operation, and each edge represents the data flow between operations. Nodes can have a varying number of parents (from one to n, depending on the operation type) but can have an arbitrary number of children. The IR graph has a single source node and multiple sink nodes. We can visualize this graph as a flow graph through which data batches propagate.

To simplify lambda type handling, certain operations that we could express with others are represented by distinct node types. For example, while we could implement a `Map` operation using a `FlatMap`, both have separate node types in the IR. This separation simplifies the management of lambda types and improves the overall structure of the IR.

The following list details all node types used to construct the IR:

Root The input of the execution

Parameters : None

Properties :
- There is exactly one Root node in an IR graph.
- A root node has no parent.
- The node cannot receive an input with the same context twice.
- The node does not modify the input value.

Consume The output of the execution

Parameters :
- The consumer function

Properties :
- The node is considered a sink.
- This is the only node that can take an impure function as a parameter.

Map A 1-to-1 application

Parameters :
- The application function

FlatMap A 1-to-N application

Parameters :
- The application function

Filter A filter operation. It can intercept values and remove them from a batch.

Parameters :
- The filter predicate

FilterType A filter based on the type. It is a different node for type safety.

Parameters :
- The accepted types

FilterNonNull A node that discards null values. It is also a different node for type safety.

Parameters : None

Aggregate A N-to-1 application.

Parameters :
- The application function

Properties :
- The operation applies to a whole batch of data.

AggregateDrop A N-to-optional application, but the result can be discarded. (section 3.2.2)

Parameters :
- The application function,

Properties :
- The operation applies to a whole batch of data.

Combine A combination of two nodes. It combines the outputs of two nodes.

Parameters :
- The combination function takes one value from each side and merges them.

Properties :
- The node has exactly two parents.
- Both sides produce any amount of elements; they are combined 1-to-1 using a cartesian product.
- The combination must respect the scopes. Values from two different scopes[4] should not be combined.

CombineDrop A combination of two nodes, but the result can be discarded. (section 3.2.2)

Parameters :
- The combination function takes one value from each side and merges it into a Droppable. *(See Drop variants)*

Properties :
- The node has exactly two parents.
- Both sides produce N elements combined 1-to-1 using a cartesian product.
- The combination must respect the scopes. Values from two different scopes should not be combined.

[4]Two scopes are different if, for the same scope id, there are two different values.

Union This node merges the batches of multiple nodes.

Parameters : None

Properties :

- The node has any number of parents.
- The output type is the common supertype of all parents.
- The combination must respect the scopes. Values from two different scopes should be combined in two different batches.

Scope This node spawns a new batch for each value it receives. This is the only mechanism that creates new batches. Each created batch is tied to the value that spawned it.

Parameters :

- ScopeId: The unique id of the scope. Other nodes use it to distinguish the scopes.

Properties :

- For every path that leads to a sink (consume node), there must be an *Unscope* node in the way. This ensures that a sink will receive exactly one batch of values.
- A given path can have multiple scope nodes in it. This means that multiple values can scope a batch.
- There can be multiple scopes with the same ID, but they cannot be descendant of one another.

Unscope This node merges back batches created by its related scope.

Parameters :

- ScopeIds: The id of all the related scopes. There can be multiple ones.

3.3.1 Construct Scopes

Scopes can be introduced into the IR in two ways. The first method occurs when the scopped operations are not present on the IR yet. This is the process used when defining scopes using the `scope { scoped operations }` construct. The *Scope* node is inserted, the *scoped operations* are added, and finally, the *Unscope* node closes.

The second method is used to scope existing nodes. This is the process done by the `scopedTo(node)` operation. A naive solution might be directly inserting the *Scope* and *Unscope* nodes at the desired location. However, this can alter the behavior of the newly scoped nodes. If a sink node depends on these modified nodes, the overall pipeline behavior can be affected.

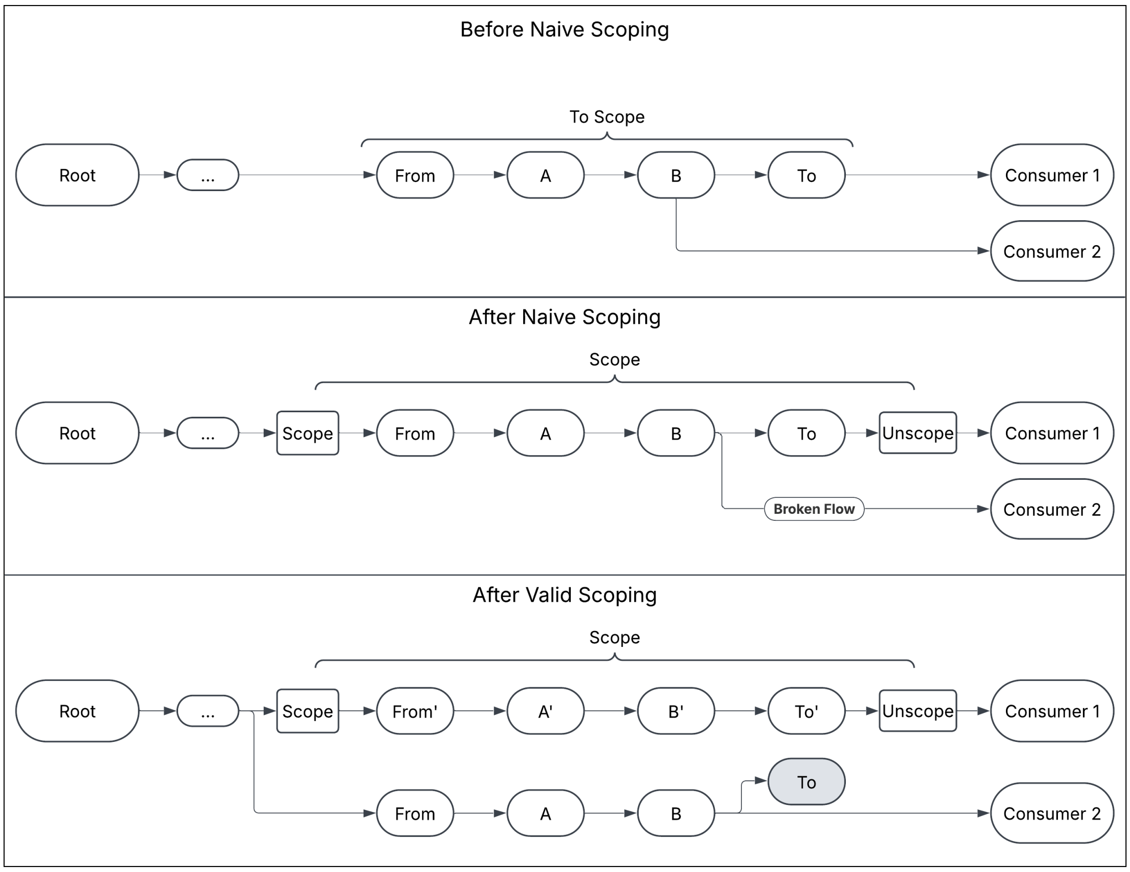

Figure 3.2: Scope existing nodes

The solution is to duplicate the nodes to be scoped and place the copies between the *Scope* and *Unscope* nodes. The original nodes remain unaffected. We then rely on the framework's dead branch discarding mechanism to eliminate the now-unnecessary original nodes, ensuring that only the correctly scoped and duplicated nodes remain in the final execution. Figure 3.2 shows the process and the problem caused by the naive approach.

3.3.2 Discard dead branches

Certain nodes may become redundant during the IR construction or subsequent transformations. A "useless" node has either no parents or no sink node among its descendants. Such nodes serve no purpose in the pipeline, and we can safely remove them from the graph. Eliminating these dead branches simplifies the IR, reduces processing overhead, and improves the overall efficiency of the execution.

3.3.3 Merge equivalent nodes

A key optimization, particularly effective at scale, addresses redundant computations. As the number of rules increases, the likelihood of equivalent operations being performed in different rules also rises. This transformation aims to minimize duplicated work using a straightforward approach. Suppose two or more children of a node are equivalent (represent the same operation with the same inputs). In that case, we can merge into a single child, reducing the number of times that operation needs to be executed.

3.4 Execution Form

After the IR is validated and optimized, it is transformed into an executable form. This executable can take various forms, provided it adheres to the IR's logic. We discuss two implementations here. The first is a batch implementation that closely mirrors the IR thought process. It served as an initial implementation. The second is a greedy implementation that attempts to minimize computation with techniques similar to iterator-based systems.

Both implementations, like the IR, are represented as flow graphs. However, we reduce the node information in the executable form to the bare minimum required for execution. Communication between nodes is signal-based; all informations, including values, are passed via signals.

In a given execution, a signal can traverse a flow only once. Nodes with multiple parents must ensure that a signal is propagated only once. Additionally, the nodes must preserve the order of signals. For nodes with a single parent, this is straightforward. However, for nodes with multiple parents, it is essential to maintain the order of signals provided by each parent. However, they can insert signals from other parents as needed.

The primary difference between the two implementations lies in how they handle data propagation. The batch version passes data to child nodes batch by batch, while the greedy version processes data value by value. This makes the batch implementation easier to reason, while the greedy version is more straightforward and efficient.

These implementations do not represent all possible execution forms. An alternative approach could involve code generation rather than relying on pre-created objects.

3.4.1 Batch Execution

The batch implementation closely follows the IR structure. It operates by passing signals along the graph's edges. It relies on two signal types: `Batch(values: List<T>, scopes: Map<ScopeId, Any>)`, which propagates the data, and `ScopeEnd(scopeId: ScopeId, scopes: Map<ScopeId, Any>)`, which the *Unscope* node uses to reconstruct batches. The `scopes` parameter in `ScopeEnd` is used when a Scope node receives an empty batch. See algorithm 2, line 13.

The Scopes as a Map

First, as the system relies on the scoping value to differentiate batches (see algorithm 2 and 5, we need to keep track of the values. Most systems employing a scoping mechanism utilize a stack, but this approach is unsuitable for our framework due to the presence of *Combine* nodes. Both parents of a *Combine* node can provide scoped values. (see Figure 3.3) For instance, the left parent might provide values scoped by *Scope* A, and the right parent, values scoped by *Scope* B. The contract of the *Combine* node specifies that the output value will be scoped by both *Scope* A and *Scope* B. A stack-based approach would require a deterministic order of scope application, which is currently not guaranteed. In this system, scopes do not have a defined order. Therefore, a stack is not suitable to store scopes.

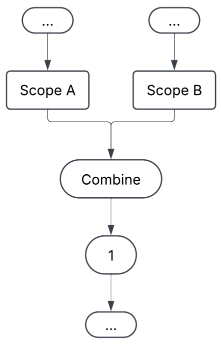

Figure 3.3: Combine scopes

3.4.2 Greedy Execution

The greedy implementation was born from the following observation: Some operations like **first** and **isPresent** only need to receive one value per batch. There is no need to process more values as it would be wasted computation.

The solution followed by the greedy implementation is to split batches in individual signals per value and rely on a completion mechanism that propagates backward to mark nodes as complete for a batch, preventing new values from being given.

It relies on two signal types: `Value(value: T)`, which propagates the data, `ExecutionEnd` which signal the end of the execution, is will always be the last signal emitted by a node. Nodes should clean their state when emitting it. And finally `BatchEnd(creator: ScopeId, disabledBy: Set<ScopeId>)`, which signals the end of a batch. The fields are used by the scope mechanism as well as batch-aware nodes. See section 4.3.3 and section 4.3.4.

3.4.3 The execution context

Each execution is associated with a unique context object. This context serves two purposes: it differentiates executions for stateful nodes (such as *Combine* nodes) and stores metadata. This metadata can be pre-computed (e.g., AST flattening) or computed during execution (e.g., for caching).

3.4.4 Flattened tree optimization

Subtree traversal is a common and computationally expensive operation. It requires accessing numerous nodes, incurring significant overhead. To mitigate this, we designed the framework to allow the replacement of specific operations with specialized nodes. Thus, a dedicated node specifically optimized for this task can replace the *FlatMap* node responsible for subtree computation.

The current implementation optimization is achieved by flattening the AST. The nodes are traversed in pre-order and stored in an array. We also maintain a *Map* that links each node to the range of its subtree. Subtree traversal then becomes a simple array traversal. The detailed algorithm is described in the implementation section. (See algorithms 10 and 11)

The array and map are computed once at the start of execution and stored as metadata within the execution context.

Chapter 4

Implementation

This chapter explores the framework's implementation details, providing insights into its practical construction and optimization strategies. We based the framework's proof of concept on SonarSource's Java analyzer. As the analyzer is written in Java, the framework uses Kotlin to stay within the JVM ecosystem while taking advantage of the extension functions feature available in the language.

4.1 Extension Functions Based

The choice of extension functions is motivated by their ability to operate on types with generics. This allows operations that are specific to builder outputs. For example, we can define an operation to find the first character in a string as an extension function specifically applicable to `SingleBuilder<String>`. This ensures type safety and prevents the accidental application of this operation to builders of other types.

However, this approach presents a challenge due to the need to separate extensions for each Selector type. This duplication increases the codebase size and is not elegant. It's worth noting that this issue is specific to compiled languages like Kotlin, Java, and Scala. For interpreted languages such as Python or Typescript, this could address this more efficiently.

4.1.1 Generated operations

To simplify the API and minimize boilerplate code, the framework includes automatic generation of accessor operations for certain types, particularly AST nodes. This feature enables developers to directly access the properties of AST nodes without manually defining mapping operations. For instance, instead of writing `methods.map(MethodDeclarationTree::parameters)`, developers can simply use `methods.parameters()`, significantly improving readability and conciseness.

While a dynamic system would be ideal for this purpose, limitations in JVM languages restrict this functionality. Interpreted languages like Python or Typescript offer more flexibility in this regard.

4.2 Identified Function

Lambda functions pose a challenge for optimization due to their opaque nature. The optimizer cannot analyze the internal logic of a lambda, hindering potential optimizations. To overcome this, the framework introduces the concept of Identified Function.

Instead of directly accepting lambdas, the IR utilizes the `IdentifiedFunction` interface, which exposes a unique identifier. Two `IdentifiedFunction` instances sharing the same identifier are defined to have identical behavior. This allows the optimizer to identify and merge equivalent operations, even if they are expressed through different lambda instances.

We decided to make the identifier optional when providing custom lambdas to improve the user experience at the cost of optimization potential. In practice, we primarily use identifiers within pre-defined framework operations.

The `IdentifiedFunction` interface has two sub-interfaces: `IdentifiedLambda` and `NodeFunction`. `IdentifiedLambda` wraps a lambda with a manually assigned identifier, while `NodeFunction` serves as a placeholder for operations that will be replaced with specialized nodes during executable creation. This replacement applies to operations like `first`, `count`, `exists`, `subtree`, etc., enabling targeted optimizations for these specific cases.

4.3 Execution Form

We designed the execution forms for minimal overhead. Nodes operate independently, aware only of their direct children. `IdentifiedFunctions` are resolved to lambdas, with `NodeFunction` translated into specialized nodes.

Data propagation relies on signals, ensuring the correct order of information flow. This signal-based approach simplifies node interactions.

4.3.1 Batch Execution

In the batch execution form, there are two signals: `Batch(values: List<T>, scopes: Map<ScopeId, Any>)` which passes data, and `ScopeEnd(scopeId: ScopeId)` which is used by the Scope mechanism. As the values propagate through the Batch signal, they traverse the graph batch per batch, and operations are applied to the entire batch at each step. This straightforward approach is easier to reason with. Value propagation involves modifying the batch as necessary and passing it to child nodes, while ScopeEnd signals are propagated without modification.

- RootNode
- ConsumerNode
- MapNode
- FlatMapNode
- FilterNode
- FilterTypeNode
- FilterNonNullNode
- AggregateNode
- AggregateDropNode

However, some nodes are more complex. They rely on a state (different per context to allow easy parallelization), and the implementation is worth discussing.

- ScopeNode
- UnScopeNode
- CombineNode
- CombineDropNode
- UnionNode

4.3.2 Greedy Execution

The architecture closely resembles the batch implementation, with the primary difference being that functions are applied to individual values instead of the entire batch. This means that the aggregate nodes must retain the values until the batch is complete in order to perform the aggregation.

4.3.3 Scope mechanism

As seen in the design chapter, the scope mechanism is central to the framework. And its implementation is not as trivial as it may seem. Here is the detailed implementation.

Batch

For the batch implementation, the following algorithms describes the scope mechanism. (Algorithm 1 and Algorithm 2)

Algorithm 1 Batch Scope

Node parameter:
scopeId: The unique id of this scope

```
1:  ▷ Takes a value signal characterized by a batch of values $batch$ and a scope map $scopes$
2:  function OnValue($batch$, $scopes$)
3:      for each $value \in batch$ do
4:          $newScope \leftarrow scopes \cup (this.scopeId \rightarrow value)$
5:          $newBatch \leftarrow [value]$  ▷ A list containing only $value$
6:
7:          PropagateValue($newBatch$, $newScope$)
8:      end for
9:
10:     Propagate($ScopeEnd(this.scopeId)$)
11: end function
```

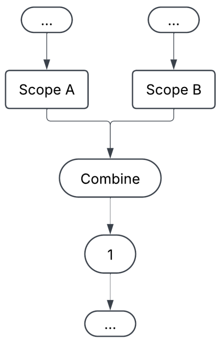

Figure 4.1: Combine scopes

The logic of the *Scope* node involves creating a new batch for each input value. After all inputs have been processed, a ScopeEnd signal indicates the batch completion. On the other side, the Unscope node stores every signal it receives the correct ScopeEnd signal. At that moment, the stored values are merged back into a single signal and propagated further.

Algorithm 2 Batch Unscope

Node parameter:
$scopeId$: The created scope unique id
Node state:
$queue$: A queue that stores signals of the current scope until it changes.

```
▷ Takes a value signal characterized by a batch of values $batch$ and a scope map $scopes$
**function** ONVALUE($batch, scopes$)
    $newScope \leftarrow scopes - this.scopeId$
    ADDTOQUEUE($batch, newScope$)
**end function**

▷ Takes a ScopeEnd signal characterized by the ended $scopeId$ and a $scopes$ map
**function** ONSCOPEEND($scopeId, scopes$)
    **if** $scopeId \neq this.scopeId$ **then**
        ADDTOQUEUE($scopeId, scopes$)
    **else**
        **if** $this.queue =$ **then**
            ▷ The related Scope node received an empty batch and propagated nothing.
            ▷ We need to recreate an empty batch.
            PROPAGATEVALUE(, $scopes$)
            **return**
        **end if**

        $curBatch \leftarrow \varnothing$
        $curScopes \leftarrow NULL$

        **for each** $signal \in queue$ **do**
            **if** $signal$ is Value($batch, scopes$) **then**
                **if** $curScope = NULL$ **then**
                    $curBatch \leftarrow curBatch \cup batch$
                    $curScopes \leftarrow scopes$
                **else if** $curScope = scopes$ **then**
                    $curBatch \leftarrow curBatch \cup batch$
                **else**
                    PROPAGATEVALUE($curBatch, curScopes$)
                    $curBatch \leftarrow batch$
                    $curScopes \leftarrow scopes$
                **end if**
            **else if** $signal$ is ScopeEnd **then**
                PROPAGATEVALUE($curBatch, curScopes$)
                PROPAGATE($signal$)

                $curBatch \leftarrow \varnothing$
                $curScopes \leftarrow NULL$
            **end if**
        **end for**
    **end if**
**end function**
```

An important detail is that the scope node must propagate empty batches to maintain system integrity, as an *Aggregate* or *Combine* node may utilize it. Additionally, due to the functionality of the *Combine* node, an Unscope node may receive batches created by an unrelated *Scope* node. (Figure 4.1 shows such a case.) This behavior is intentional; however, the unscoping process should not merge batches scoped differently by that other *Scope* node. Therefore, to merge batches, they must share the same scope after removing the target scopes.

Scopes should not be tangled for the batch execution

The batch implementation's reliance on `ScopeEnd` signals to reconstruct batches can lead to complications when Scope nodes are tangled, as illustrated in Figure 4.2.

In this configuration, *Scope B* is applied on top of *Scope A*. Consequently, when *Unscope A* receives a `ScopeEnd` signal, it cannot fully reconstruct the batch due to the partitioning imposed by *Scope B*. This renders *Unscope A* ineffective, and *Unscope B* also fails to reconstruct the original batch, as it buffers results until receiving its own `ScopeEnd` that are more frequent.

A simple solution exists: reposition *Unscope A* immediately after *Unscope B*[1]. This resolves the issue because only six node types are affected by scoping: *Aggregate*, *Combine*, *CombineDrop* and *Union*. After repositioning, nodes 5 and 6 become scoped by both *Scope A* and *Scope B*.

The behavior changes only if the number of batches is altered for the remaining node types. However, the problem arises precisely because *Unscope A* has no effect in the tangled configuration. Therefore, repositioning *Unscope A* maintains the same number of batches for nodes 5 and 6, resolving the issue without affecting the behavior of other nodes.

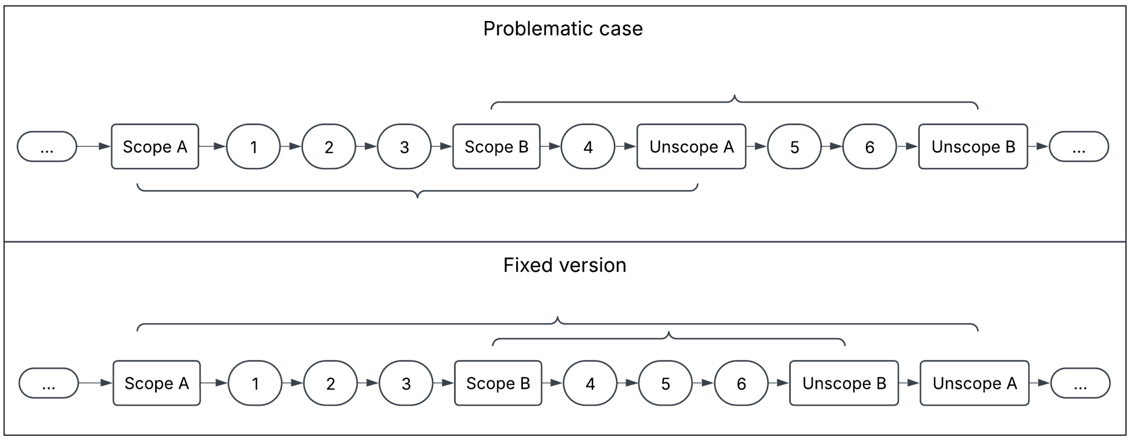

Figure 4.2: Tangled scopes fix

[1]In practice, as the Unscope node supports multiple Unscopes, Unscope A is removed and its ScopeId is added to Unscope B.

Greedy

The greedy implementation of the scope mechanism is quite straightforward. As the beginning of a new batch is indicated by a batch end signal, we can easily merge batches by removing such signal.

One notable aspect of this implementation is that multiple batch end signals can occur in succession. To prevent confusion among the batch-aware nodes, a batch end signal is disabled if it is not within the scope that created it. This ensures that only one active batch end signal is applied for each actual batch. (Algorithm 3 and Algorithm 4.3.3)

Algorithm 3 Greedy Scope

Node Parameters:
$scopeId$: The created scope unique id

```
▷ Takes a $value$ signal
function ONVALUE($value$)
    PROPAGATEVALUE($value$)
    PROPAGATEBATCHEND($this.scopeId,$ )
end function

▷ Takes a batch end signal characterized by its $creator$ and the set of scopes that disables it
function ONBATCHEND($creator$, $disabledBy$)
    $newDisabledBy \leftarrow disabledBy \bigcup this.scopeId$
    PROPAGATEBATCHEND($creator$, $newDisabledBy$)
end function
```

Algorithm 4 Greedy Unscope

Node Parameters:
$scopeId$: The created scope unique id

```
▷ Takes a $value$ signal
function ONVALUE($value$)
    PROPAGATEVALUE($value$)
end function

▷ Takes a batch end signal characterized by its $creator$ and the set of scopes that disables it
function ONBATCHEND($creator$, $disabledBy$)
    if $creator = this.scopeId$ then
        return ▷ The signal's propagation ends here
    end if
    $newDisabledBy \leftarrow disabledBy - this.scopeId$
    PROPAGATEBATCHEND($creator$, $newDisabledBy$)
end function
```

4.3.4 Combine mechanism

While conceptually simple, the combine mechanism becomes more intricate when considering scopes.

Batch

In the batch implementation (Algorithm 5), the *Combine* node prioritizes correctness and simplicity over performance. It stores signals from both parent nodes in two queues until a matching *ScopeEnd* is present in both queues. Then, batches with mergeable scopes are combined using a Cartesian product and propagated.

Greedy

The combine mechanism of the greedy execution is the most complex part of the system. It must integrate value signals generated from the same scopes. However, the two sides are not synchronized, which means that values from a future scope can be received before the other side has finished delivering its own values.

We implemented this mechanism using a state machine with the following three states:

- Base: Both sides are in the same scope and the values can be safely combined.
- LeftPending: The left node has advanced to the next scope, meaning its values should be stored in a queue until the right node completes its current scope.
- RightPending: This is the mirrored version of LeftPending. It is necessary for type-safety as the pending queue is tied to the type of the pending node.

The complete algorithm is provided by algorithm 6 and algorithm 7. The RightPending algorithm is the mirrored version of algorithm 7.

4.3.5 The Union

The implementation of the *Union* node differs significantly between the batch and greedy execution forms due to their handling of batches. The batch version only supports scenarios where all parent nodes provide the same number of batches. This limitation arises from the challenge of aligning batches from different parents that may have different scopes.

Algorithm 5 Batch Combine

Node State:

$queueLeft$: Stores all signals from the left node of the current scope.
$queueRight$: Stores all signals from the right node of the current scope.

```
▷ Takes a value signal characterized by a batch of values $batch$ and a scope map $scopes$ as well as the $side$ that produced it.
function onValue($batch, scopes, side$)
    addToQueue($side, batch, scope$)
end function

▷ Takes a ScopeEnd signal characterized by the ended $scopeId$ and a $scopes$ map as well as the $side$ that produced it.
function onScopeEnd($scopeId, side$)
    addToQueue($side, scopeId$)

    $LIndex \leftarrow$ indexOf($scopeId, queueLeft$)
    $RIndex \leftarrow$ indexOf($scopeId, queueRight$)

    for $l \leftarrow 1$ to $LIndex$ do
        $left \leftarrow queueLeft[l]$
        if $left$ is ScopeEnd then
            ▷ ScopeEnds should not be combined
            propagate($left$)
        else
            for $r \leftarrow 1$ to $RIndex$ do
                $right \leftarrow queueRight[r]$
                if $right$ is ScopeEnd then
                    ▷ ScopeEnds should not be combined
                    propagate($right$)
                else
                    $left$ and $right$ are both batches; their values can
                    be combined one by one and merged into a single batch.
                end if
            end for
        end if
    end for

    ▷ Drop the signals that have been consumed
    $queueLeft \leftarrow queueLeft[Lindex + 1 :]$
    $queueRight \leftarrow queueRight[Rindex + 1 :]$
end function
```

Algorithm 6 Greedy Combine - Base State

Node Parameter:

commonScopes: the set of scopes ids that are common to both parents.

Node State:

leftBuffer: a buffer of left signals in current scope.

rightBuffer: a buffer of right signals in current scope.

```
▷ Called when a $value$ is received for the left parent.
function onValueLeft($value$)
    append($this.leftBuffer$, $value$)
    ▷ The cartesian product is done greedily using the buffered signals
    combineAllAndPush($value$, $this.rightBuffer$)
end function

▷ Called when a batch end signal is received from the left parent. It is characterized by its
$creator$ and the set of scopes that disables it
function onBatchEndLeft($creator$, $disabledBy$)
    if $creator \in this.commonScopes$ then
        append($this.leftBuffer$, $BatchEnd(creator, disabledBy)$)
        propagateBatchEnd($creator$, $disabledBy$)
        return
    end if

    ▷ We received a batch end signal that is common to both parents. We need to wait for right
    to push it before going further.
    $signal \leftarrow BatchEnd(creator, disabledBy)$
    ▷ As there will be no more left signal to combine, the right buffer can be dropped
    $nextState \leftarrow LeftPending(commonScopes, leftBuffer, \{signal\})$
    MoveToNextState($nextState$)
end function

▷ The right versions of the functions are functionally the same as the left ones but mirrored.
function onValueRight($value$)
    append($this.rightBuffer$, $value$)
    combineAllAndPush($value$, $this.leftBuffer$)
end function

function onBatchEndRight($creator$, $disabledBy$)
    if $creator \in this.commonScopes$ then
        append($this.rightBuffer$, $BatchEnd(creator, disabledBy)$)
        propagateBatchEnd($creator$, $disabledBy$)
        return
    end if

    $signal \leftarrow BatchEnd(creator, disabledBy)$
    $nextState \leftarrow RightPending(commonScopes, rightBuffer, \{signal\})$
    moveToNextState($nextState$)
end function
```

Algorithm 7 Greedy Combine - LeftPending State

Node Parameter:
commonScopes: the set of scopes ids that are common to both parents.
Node State:
leftBuffer: a buffer of left signals in current scope.
pending: the queue of left signal from the next scope.

```
▷ Called when a $value$ is received for the left parent.
function ONVALUELEFT($value$)
    QUEUE($this.pending$, $value$)
end function

▷ Called when a batch end signal is received from the left parent. It is characterized by its $creator$ and the set of scopes that disables it
function ONBATCHENDLEFT($creator$, $disabledBy$)
    $signal \leftarrow BatchEnd(creator, disabledBy)$
    QUEUE($this.pending$, $signal$)
end function

▷ Called when a $value$ is received for the right parent.
function ONVALUERIGHT($value$)
    ▷ As no more signal from current scope will be received from left, there is no need to buffer.
    COMBINEALLANDPUSH($value$, $this.leftBuffer$)
end function

▷ Called when a batch end signal is received from the right parent. It is characterized by its $creator$ and the set of scopes that disables it
function ONBATCHENDRIGHT($creator$, $disabledBy$)
    if $creator \in this.commonScopes$ then
        ▷ As no more signal from current scope will be received from left, there is no need to buffer.
        PROPAGATEBATCHEND($creator$, $disabledBy$)
        return
    end if
    ▷ The pending batch end has been received

    ▷ The first signal in the queue is the batch end that signals the next scope.
    $pendingBatchEnd$ = DEQUEUE($this.pending$)
    $mergedDisabledBy = disabledBy \bigcap pendingBatchEnd.disabledBy$
    PROPAGATEBATCHEND($creator$, $mergedDisabledBy$)

    ▷ Move back to the base state, with pending signals as the left buffer and an empty right buffer
    $nextState \leftarrow Base(this.commonScopes, this.pending, )$
    MOVETONEXTSTATE($nextState$)
end function
```

Consider a scenario where a *Union* node has two parents, left and right. The node receives two batches from its left parent and one batch from its right parent. In this case, the desired behavior is to group elements based on the parent value that generated them. This grouping is straightforward in the greedy execution form, which naturally achieves this through its depth-first approach. However, batch-based processing in the batch form renders it impossible.

Due to this constraint, the batch Union node implementation is simplified. It assumes that all parent nodes will produce the same number of signals and must store them until all are available.

The greedy *Union* node, on the other hand, requires a more complex implementation to handle scopes effectively. It is similar to the combine mechanism, but there is no need to differentiate left and right states, nor should signals be combined.

Algorithm 8 Greedy Union - Base State

Node Parameter:
$commonScopes$: the set of scopes ids that are common to both parents.

```
1: ▷ Called when a $value$ is received.
2: function ONVALUE($value$)
3:     PROPAGATEVALUE($value$)
4: end function
5:
6: ▷ Called when a batch end signal is received from a $side$. It is characterized by its $creator$ and the set of scopes that disables it
7: function ONBATCHEND($creator$, $disabledBy$, $side$)
8:     if $creator \in this.commonScopes$ then
9:         PROPAGATEBATCHEND($creator$, $disabledBy$)
10:        return
11:    end if
12:
13:    ▷ We received a batch end signal that is common to both parents. We need to wait for the other side to push it before going further.
14:    $signal \leftarrow BatchEnd(creator, disabledBy)$
15:    $nextState \leftarrow SidePending(commonScopes, side, [signal])$
16:    MOVETONEXTSTATE($nextState$)
17: end function
```

4.3.6 Flattened AST for the subtree operation

Algorithm 9 Greedy Combine - SidePending State

Node Parameter:

commonScopes: the set of scopes ids that are common to both parents.

Node State:

pending: the queue of signal from the next scope.

```
▷ Called when a $value$ is received from the given $side$.
function OnValue($value, side$)
    if $side = this.pendingSide$ then
        Queue($this.pending, value$)
    else
        PropagateValue($value$)
    end if
end function

▷ Called when a batch end signal is received from a $side$. It is characterized by its $creator$ and
the set of scopes that disables it
function OnBatchEnd($creator, disabledBy, side$)
    if side = this.pendingSide then
        $signal \leftarrow BatchEnd(creator, disabledBy)$
        Queue($this.pending, signal$)
    else
        if $creator \in this.commonScopes$ then
            PropagateBatchEnd($creator, disabledBy$)
            return
        end if
        ▷ The pending batch end has been received
        ▷ The first signal in the queue is the batch end that signals the next scope.
        $pendingBatchEnd$ = Dequeue($this.pending$)
        ▷ The merged signal is disabled by a scope if it is present on both sides.
        $mergedDisabledBy = disabledBy \bigcap pendingBatchEnd.disabledBy$
        PropagateBatchEnd($creator, mergedDisabledBy$)

        ▷ Propagate all pending signals
        PrpagateAll($pending$)
        $nextState \leftarrow Base(this.commonScopes)$
        MoveToNextState($nextState$)
    end if
end function
```

Algorithm 10 Flatten AST

Input: The $root$ of the AST to flatten **Output:** The flattened AST and a map from each node to the range of their subtree.

```
function FLATTENAST($root$)
    $counter \leftarrow 0$
    $flattenAST \leftarrow []$
    $nodeToSubtree \leftarrow \varnothing$

    function ONNODE($node$)
        $startIndex \leftarrow counter$
        $counter \leftarrow counter + 1$
        $flattenAST[startIndex] \leftarrow node$

        ▷ The node will call onNode for each of its children.
        $node$.VISIT($onNode$)
        $endIndex \leftarrow counter$

        $range \leftarrow (startIndex, endIndex + 1)$
        $nodeToSubtree \leftarrow nodeToSubtree \bigcup (node \rightarrow range)$
    end function

    ONNODE($root$)
    return $flattenAST$, $nodeToSubtree$
end function
```

Algorithm 11 Optimized sub-tree visit

Input:
The result of the AST flattening: $flattenAST$ and $nodeToSubtree$.
The root of the subtree $node$.
A predicate to decide if a node should be traversed $shouldTraverse$
A consumer that will receive every traversed node $consumer$

```
function VISITSUBTREE($node, shouldTraverse, consumer$)
    $(cur, end) \leftarrow nodeToSubtree[node]$
    while $cur < end$ do
        $next \leftarrow flattenAST[cur]$
        if SHOULDTRAVERSE($next$) then
            CONSUME($next$)
            $cur \leftarrow cur + 1$
        else
            ▷ Skip the ignored sub-tree by setting the pointer to that subtree's end
            $(skippedStart, skippedEnd) \leftarrow nodeToSubtree[next]$
            $cur \leftarrow skippedEnd$
        end if
    end while
end function
```

Chapter 5

Evaluation

This chapter presents the evaluation of the proposed framework, focusing on three key aspects: correctness, performance, and usability. The evaluation involves migrating a subset of existing rules from SonarSource's Java analyzer to the new framework and comparing the results against the original implementation. The primary goal is to demonstrate that the framework delivers accurate results, does not hinder performance, and enhances the usability of rule creation and maintenance. The source code of the demonstration is linked in the appendix .4.

5.1 Methodology

The evaluation process involved migrating 8 rules from SonarSource's Java analyzer to the new framework. The migrated rules were then evaluated against their original counterparts using various open-source projects, detailed in the appendix.

5.1.1 Correctness

The evaluation of correctness focused on ensuring that the migrated rules produced the same output as the original rules. This process involved comparing the results from both sets of rules across selected open-source projects. We verified that they identified the same issues and generated identical reports.

5.1.2 Performance

The performance evaluation aimed to measure the efficiency of the framework in comparison to the original rule execution engine and to assess the benefits of each optimization. To isolate the impact of the framework, both systems were simplified by removing all rules except for the 15 that were migrated. These stripped-down systems were then tested on open-source projects, with measurements taken for rule execution time, the pipeline build and AST flattening when applicable.

The evaluation considered several scenarios:

- **Framework (No Optimization):** The new framework, without any optimizations enabled, is used as the base of comparison.
- **Framework (Individual Optimizations):** The new framework with each optimization enabled individually:
  Merge Common Base: Merging of equivalent operations.
  Fast Subtree: Pre-computed AST flattening for efficient subtree traversal.
  Greedy Execution: Using the greedy implementation for the execution.
- **Framework (All Optimizations):** The new framework with all major optimizations enabled.
- **Framework (Parallelized):** The new framework enabling multi-core computation.
- **Original Implementation:** The original rule execution engine without any modifications.

The measures were done on a stable computer, the detailed specifications are available in appendice.2.

5.2 Usability

The usability evaluation focused on assessing the framework's ability to simplify the creation and maintenance of static analysis rules. This assessment was based on five key criteria, aligned with SonarSource's target:

- **General-Purpose Language Integration:** The framework should seamlessly integrate into a general-purpose language, allowing developers to write rules using familiar syntax and tools.
- **Compile-Time Type Safety:** The framework should provide compile-time type safety, ensuring that rule definitions are checked for errors before execution.

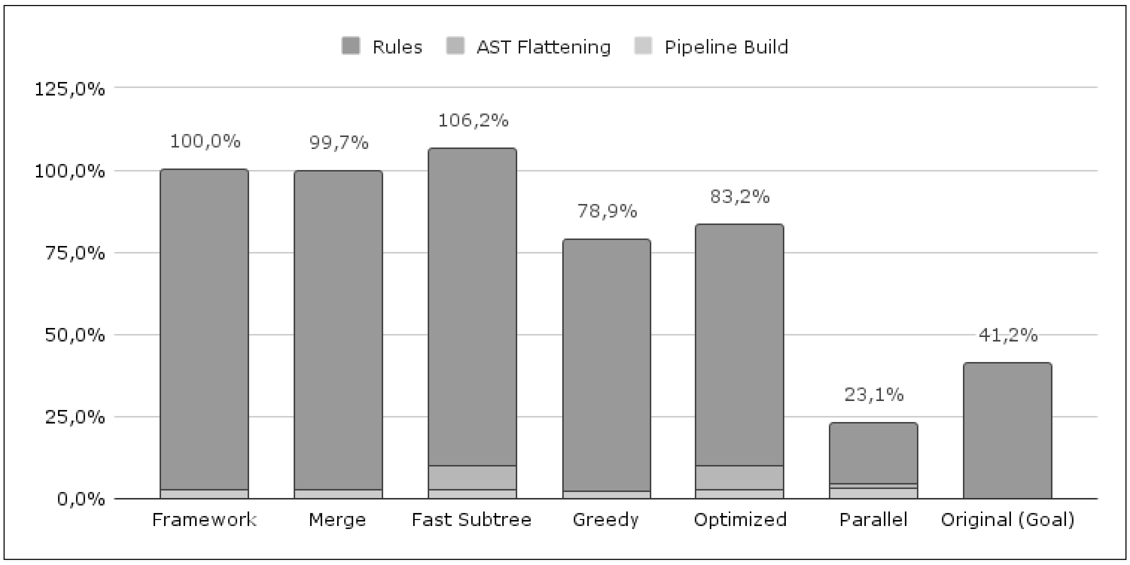

Figure 5.1: Chart of benchmark results (Appendix .3)

- **Result Processing:** The framework should allow for further processing of the results, enabling developers to customize how issues are reported and handled.
- **Expressiveness:** The framework should be expressive enough to migrate most, if not all, existing rules without significant limitations.
- **Cross-Language Support:** Ideally, the framework should support writing a single rule that can be applied to the ASTs of multiple languages.

5.3 Results

5.3.1 Correctness

The correctness evaluation was successful, with the migrated rules producing identical outputs to the original rules. This confirms that the framework accurately implements the logic of the selected rules and generates the same results.

5.3.2 Performance

The pipeline construction occurs once per analysis, once for each of the 126 analyzed modules. The flattening and rule execution are performed once per file, totaling 8,215 times. The time required for rule execution increases with the number of rules, while the flattening time remains constant.

The results, presented in Figure 5.1, show that the framework's performance without optimizations is lower than the original, which is expected due to the significant overhead introduced by the additional abstraction layer. The optimizations aim to outweigh this overhead.

Both merge and fast subtree optimizations show no significant improvement. For merge, the gain during rules execution is 0.6%, but the build time increases by 0.3%. As the strategy targets scalability, the number of migrated rules is too low for this optimization to have any real impact. The same effect can be observed for the fast subtree optimization, but it is worse as the tree flattening operation consumes 7.5% of the overall time. The optimized subtree traversal has a measurable impact on the rules execution (1.2%), but there are not enough subtree operations in the evaluated ruleset to outweigh the fixed flattening cost.

The greedy execution engine significantly improves the overall execution time, but it is still twice the time of the original implementation. If we enable all optimizations, the tree flattening operation hinders the performance. However, if we look at rule execution alone, it is the fastest single-threaded framework version by 3.5%. With more rules, the relative cost of tree flattening should decrease, likely making it the fastest overall.

Even with all optimizations, the execution time is still twice the goal. The final attempt relies on parallelization. The framework supports it out of the box, while the original implementation would require a refactor as large as migrating rules to support it. With parallelization, the framework is almost two times faster than the original implementation.

Currently, we cannot confidently say that the merge and fast subtree optimizations are worthwhile. We need to repeat the experiment with a larger ruleset to get a definitive answer. The greedy execution engine improves performance but is not sufficient by itself to compete with the original implementation. We had to rely on parallelization to beat it, which we believe is not sufficient. We think that every layer of the framework has room for improvement performance-wise. The IR can be further optimized with more advanced techniques (e.g., find common subgraph and cache the results). We can explore alternative execution strategies (e.g., pulling signals like iterators instead of pushing them or generating the code of the execution engine instead of relying on abstractions). The algorithms can certainly be improved, and we believe that the implementation is far from perfect.

The combine mechanism is the main source of latency. We could add more advanced optimizations inspired by common database systems that face the same issue with joins. Another possibility could be to focus on the very common construct that `where` and `groupWith` operations produce as they occur frequently.

5.3.3 Usability

General-Purpose Language Integration

The framework's development in Kotlin allows for seamless integration with any JVM language, fulfilling this criterion. However, the design itself is not limited to Kotlin and can be adapted to other general-purpose languages, such as C#, TypeScript, or Python. A particularly promising approach involves combining Python's dynamic typing for the API with a C module for the execution engine, potentially offering both flexibility and performance.

Compile-Time Type Safety

The developed framework is fully type-safe, ensuring that rule definitions are checked for errors at compile time. This helps prevent runtime errors and improves the reliability of the static analysis process.

Result Processing

The framework's design allows for further processing of the results through the Consume operation. This flexibility is crucial for integrating the framework into existing static analysis tools and workflows.

Expressiveness

While theoretically any rule can be migrated to the framework by moving the code inside lambdas, the goal is to achieve high expressiveness without resorting to embedding complex logic within lambdas as it defeats the usability goal.

In practice, the framework proved capable of migrating a wide range of rules without significant limitations. However, a comprehensive assessment of its expressiveness would require migrating a larger number of rules.

Cross-Language Support

Currently, the framework does not support writing a single rule for multiple languages. This limitation stems from its reliance on existing AST representations, which differ between languages. Achieving multi-language support would require a universal AST, such as the one proposed by JetBrains for the JVM ecosystem. But this solution falls outside this paper's scope.

Chapter 6

Related Work

This section examines existing approaches to descriptive tree traversal and node selection within static analysis, situating the proposed framework within the broader landscape of current solutions. We explore matcher-based and deductive query-based systems and then discuss related approaches to the general technical aspect of the framework from outside the static analysis domain.

6.1 Matcher Systems

Matcher-based systems, exemplified by tools such as Clang[13] and Tree-sitter[14], offer a more declarative style compared to purely visitor-based approaches, offering significant advantages in expressing structural patterns within the AST. However, a key challenge arises when needing to establish relationships between matched elements within the AST. For instance, a tool might need to detect methods with direct recursive calls. This requires not only identifying method declarations but also locating all method call nodes within the body that refer to the same method. While matchers excel at identifying individual elements, linking these elements together to represent such relationships requires complex combinations of matchers or auxiliary data structures. This complexity can quickly lead to rules that are difficult to read, write, and maintain, potentially negating the benefits of the declarative approach. Furthermore, existing solutions for linking matched elements may present challenges related to type safety. This paper explores a novel framework designed to address these limitations, aiming to simplify the creation of these links in a descriptive and type-safe manner.

6.2 Deductive query systems

Deductive query systems, introduced by Datalog[8] and demonstrated by projects such as CodeQL[2] and Soufflé[3], offer powerful descriptive capabilities for rule definition but relying on specialized languages like Datalog for rule definition. This necessitates a separation between rule definition and core program logic in tools written using general-purpose languages like Java, Python, or JavaScript, potentially increasing development complexity. Integrating the results of deductive queries back into the main program flow can introduce additional overhead. Because deductive systems are external to the main tool, they require specially crafted bindings. This is a common problem also encountered with general database systems, where Object-Relational Mapping (ORM) tools are typically employed to bridge the gap between the external database and the general-purpose codebase. Similar techniques would be required for deductive systems, adding another layer of complexity to the integration process.

Furthermore, the two-phase execution model, involving initial codebase analysis and subsequent rule execution, can introduce a significant upfront cost. For very large codebases, this initial analysis phase can be time-consuming, potentially impacting the responsiveness of the static analysis process. This is particularly relevant in modern development workflows where static analysis is often integrated directly within Integrated Development Environments (IDEs). IDE integration is essential for providing developers with immediate feedback on code quality and potential issues. However, IDEs are subject to very frequent file updates as developers write and modify code. In such scenarios, deductive systems, with their requirement for global analysis, would incur the upfront cost of re-analyzing the entire codebase after each file modification. While incremental analysis, where only changed code portions are re-analyzed, is a potential solution to this problem, it remains an active area of research for deductive query systems.

However, deductive systems introduce overhead. They rely on specialized languages, necessitating a separation between rule definition and core program logic in tools written in general-purpose languages. This separation can increase development complexity and introduce integration challenges. Additionally, the two-phase execution model can introduce significant upfront costs, particularly for large codebases, impacting the responsiveness of static analysis. Incremental analysis, while a potential solution, remains an active research area for deductive query systems [12][11] and has yet to be brought into production.

6.3 Data Pipeline Creation Framework

The proposed framework positions is a general data pipeline creation tool. While, to our knowledge, no directly analogous frameworks exist in the literature, several related approaches merit discussion.

- *Functional Collections:* Using extension functions, one can achieve descriptive capabilities similar to the framework. However, this approach lacks the optimization and lazy execution features of the proposed framework.
- *Functional Sequences (Iterator-Based Systems):* Functional sequences offer descriptive capabilities and lazy execution. However, sequences can only be executed once, conflicting with the reusability requirements of data pipelines.
- *Pre-compiled Queries in Databases:* This solution fulfills the descriptive requirements; however, this method bears a close resemblance to deductive systems with lower expressiveness power.

While conceptually relevant, existing optimization strategies from graph databases[4][1] present practical limitations for our framework. These strategies often cater to the complexities of large-scale databases, involving intricate graph traversals and distributed processing. As a result, they are either incompatible with our framework's structure or focus on elements irrelevant to single-file AST analysis.

However, some underlying principles of these optimizations remain valuable. For instance, the AST flattening optimization, which precomputes and stores derived information about the AST, is inspired by precomputation techniques used in graph databases to accelerate query execution.

Chapter 7

Conclusion

This paper introduces a framework for enhancing the development of static analysis tools by enabling a descriptive approach to defining rules. The framework's architecture, based on an intermediate representation and a flexible execution engine, allows for seamless integration with existing tools and AST representations.

The framework's primary contribution is its potential to improve the usability and maintainability of static analysis tools. The framework reduces the complexity of AST traversal and node selection code, allowing developers to focus on the rule's intent rather than the implementation details. This can lead to more efficient development, maintenance and ultimately better code quality.

An evaluation using SonarSource's Java analyzer by migrating existing rules to the framework showed promising results, but we do not think that the framework is mature yet. The evaluation results indicate that the batch and greedy execution forms are slower than the original system. Only the parallel optimization leads to a significant performance improvement, surpassing the original system. The merge and fast subtree optimizations do not significantly affect performance in the experimental environment.

The experiment needs to be redone, with more migrated rules to properly evaluate the scalability-focused optimizations. Other than that, future work will focus on extending the framework's API and investigating other optimization techniques on all layers.

Despite these open questions, the framework presented in this paper represents a significant step towards more declarative and user-friendly static analysis tools. By simplifying rule definition and providing a flexible architecture, the framework has the potential to make static analysis more accessible and effective, ultimately contributing to the development of higher-quality software.

Bibliography

[1] Ali Ben Ammar. “Query Optimization Techniques In Graph Databases”. In: *CoRR* abs/1609.01893 (2016). arXiv: 1609.01893. URL: http://arxiv.org/abs/1609.01893.

[2] GitHub. *“CodeQL.* [Accessed 24-02-2025]. 2025. URL: https://codeql.github.com.

[3] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. “Soufflé: On synthesis of program analyzers”. In: *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28.* Springer. 2016, pp. 422–430.

[4] Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, and Jingren Zhou. *A Modular Graph-Native Query Optimization Framework.* 2024. arXiv: 2401.17786 [cs.DB]. URL: https://arxiv.org/abs/2401.17786.

[5] Andrea Maglie and Andrea Maglie. “Reactivex and rxjava”. In: *Reactive Java Programming* (2016), pp. 1–9.

[6] PMD. *“PMD.* [Accessed 24-02-2025]. 2025. URL: https://pmd.github.io.

[7] Scala Center. *“Tasty-query.* [Accessed 24-02-2025]. 2025. URL: https://github.com/scalacenter/tasty-query.

[8] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. “On fast large-scale program analysis in Datalog”. In: *Proceedings of the 25th International Conference on Compiler Construction.* CC ’16. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 196–206. ISBN: 9781450342414. DOI: 10.1145/2892208.2892226. URL: https://doi.org/10.1145/2892208.2892226.

[9] SonarSource. *“Better Code & Better Software.* [Accessed 24-02-2025]. 2025. URL: https://www.sonarsource.com.

[10] SonarSource. *“GitHub — SonarSource/sonar-java.* [Accessed 24-02-2025]. 2025. URL: https://github.com/SonarSource/sonar-java.

[11] Tamás Szabó. “Incrementalizing Production CodeQL Analyses”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 1716–1726. ISBN: 9798400703270. DOI: 10.1145/3611643.3613860. URL: https://doi.org/10.1145/3611643.3613860.

[12] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. "Incremental whole-program analysis in Datalog with lattices". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 1–15.

[13] The Clang Team. *Welcome to Clang's documentation*. [Accessed 24-02-2025]. 2025. URL: `https://clang.llvm.org/docs/#using-clang-as-a-library`.

[14] tree-sitter. *Introduction - Tree-sitter*. [Accessed 24-02-2025]. 2025. URL: `https://tree-sitter.github.io/tree-sitter/`.

[15] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 334–344. DOI: `10.1109/MSR.2017.2`.

[16] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. "On the value of static analysis for fault detection in software". In: *IEEE Transactions on Software Engineering* 32.4 (2006), pp. 240–253. DOI: `10.1109/TSE.2006.38`.

.1 Evaluated projects

The section lists in detail the open-source projects that were used to evaluate the correctness and performance of the framework.

Common Beans-utils
- **Repo:** https://github.com/apache/commons-beanutils
- **Author:** Apache
- **Commit:** 0e77eae2bcc6507ce3ba5bf0cdacc98ed54cfaea

Dubbo
- **Repo:** https://github.com/apache/dubbo
- **Author:** Apache
- **Commit:** 403e127385782e2cf4db48df7c25a4cbfc5fcfd5

Guava
- **Repo:** https://github.com/google/guava
- **Author:** Google
- **Commit:** 008e78e799cfe6ef47beab313a99d23fdf334b87

JBoss EJB3
- **Repo:** https://github.com/jbossejb3/jboss-ejb3
- **Author:** JBoss EJB 3
- **Commit:** db22c6b4ade48b3f485e6146d4d3b37ef3540ce0

Jetty
- **Repo:** https://github.com/jetty/jetty.project
- **Author:** Eclipse
- **Commit:** 5d1d7c9c26153a6549f38f795dd068abc6d18653

Netty
- **Repo:** https://github.com/netty/netty
- **Author:** Netty
- **Commit:** 24479a05cd20394d6d393b1ce3037046218158cc

Ribbon
- **Repo:** https://github.com/Netflix/ribbon
- **Author:** Netflix
- **Commit:** 45e59260c1e0b8834d4cc960895ba59ecbda7f63

SonarQube Server
- **Repo:** https://github.com/SonarSource/sonarqube
- **Author:** SonarSource
- **Commit:** ceb51f1d06b8a2e357ef42df10433d350f27175a

RxJava
- **Repo:** https://github.com/ReactiveX/RxJava
- **Author:** ReactiveX
- **Commit:** 0fe87b8add2147f7b09fe83d611be51edfdd27da

Zuul
- **Repo:** https://github.com/Netflix/zuul
- **Author:** Netflix
- **Commit:** 9510021b79f0279d512be12fea1a54e95ee4f6e0

.2 Benchmark Environment

The computer used to execute the benchmark had the following specifications:

Model: HP Precision 3591

Processor: Intel Core Ultra 7 165H

RAM: 64 GB

OS: Windows 10 Pro 22H2

Java Runtime: Corretto-17.0.13.11.1 (build 17.0.13+11-LTS)

.3 Benchmark results

Table 1: Benchmark results compared to framework's total

		Framework	Merge	Fast Subtree	Greedy	Parallel	Optimized	Original (Goal)
Rules	Run 1	97,5%	98,8%	97,1%	76,2%	75,7%	18,8%	42,1%
	Run 2	97,0%	96,9%	92,6%	72,8%	71,4%	16,8%	40,4%
	Run 3	97,3%	95,4%	96,7%	75,4%	73,4%	17,7%	41,6%
	Run 4	97,2%	95,6%	97,2%	78,3%	70,5%	17,0%	41,0%
	Run 5	97,0%	96,1%	96,3%	78,3%	73,2%	20,8%	41,0%
Pipeline Build	Run 1	2,5%	3,1%	2,8%	3,0%	2,9%	3,4%	
	Run 2	3,0%	3,1%	2,7%	2,4%	2,8%	2,9%	
	Run 3	2,7%	3,2%	2,7%	2,6%	3,2%	3,5%	
	Run 4	2,8%	3,0%	2,9%	2,6%	2,9%	3,3%	
	Run 5	3,0%	3,2%	2,8%	2,7%	2,9%	3,0%	
Flattening	Run 1			7,2%		7,5%	1,3%	
	Run 2			7,1%		7,2%	1,8%	
	Run 3			7,4%		7,6%	2,0%	
	Run 4			7,5%		7,2%	1,4%	
	Run 5			7,6%		7,7%	1,7%	

Table 2: Aggregated results

	Framework	Merge	Fast Subtree	Greedy	Optimized	Parallel	Original (Goal)
Rules	97,2%	96,6%	96,0%	76,2%	72,8%	18,2%	41,2%
Pipeline Build	2,8%	3,1%	2,8%	2,7%	2,9%	3,2%	
Flattening			7,4%		7,5%	1,6%	

.4 Project Repository

- Core library: https://github.com/GabrielFleischer/pipeline-builder
- SonarJava demonstration: https://github.com/GabrielFleischer/ast-query-sonar-java