An Interpreter-based Framework for Static Analysis of Variability in Space and Time

Tayná Vieira
CIC/UnB CDS/EB
Brasília, Brasil
lariifischer@gmail.com

Vander Alves
CIC/UnB
Brasília, Brasil
valves@unb.br

Leopoldo Teixeira
CIn/UFPE
Recife, Brasil
lmt@cin.ufpe.br

ABSTRACT

Software Product Lines (SPLs) enable the systematic development of configurable software systems by organizing products as families that share commonalities and differ in selected features. However, static analysis in SPLs poses scalability challenges due to variability in space (across configurations) and variability in time (across software revisions). This paper presents an interpreter-based framework that combines variational lifting and memoization to support scalable and reusable static analysis of evolving SPLs. Analyses are implemented as PCF+ programs and executed over variational representations of object programs, annotated with presence conditions. Memoization mechanisms allow the reuse of previously computed results across object program evolutions, reducing redundant computations and contributing to performance improvements. The framework is evaluated using benchmarks simulating software evolution scenarios. Results indicate that the combined use of variational lifting and memoization effectively reduces execution time, suggesting the advantages of addressing both dimensions of variability. This work contributes a reusable infrastructure for control-flow-based analyses in SPLs and provides empirical evidence supporting its efficiency.

KEYWORDS

Software Product Lines, Software Product Line Static Analysis, Variability-Aware Execution, Memoization, Functional Programming

1 Introduction

Static analysis techniques are fundamental to ensuring the correctness, safety, and maintainability of software systems [8]. In the context of highly configurable systems, such as those built using Software Product Lines (SPLs), these analyses face significant scalability challenges [14]. SPLs organize products as families that share core assets and differ in selected features [1, 12], enabling systematic reuse and reducing time-to-market. However, they introduce two critical dimensions of variability that impact static analysis: *variability in space*, due to multiple configurations, and *variability in time*, due to the continuous evolution of the codebase [15].

To illustrate the limitations of current approaches, consider the variational program in Figure 1 (a), which is part of an SPL. The code snippet shows how a user password is conditionally sanitized depending on the presence of the feature flag `SANITIZE`. The elided code ([...]) denotes common segments that are identical across all configurations and are omitted here for clarity.

```
[...]
pwd := input;
#ifdef SANITIZE
pwd := sanitized_input;
#else
skip;
#endif
result := pwd;
[...]
```

(a)

```
[...]
pwd := input;
#ifdef SANITIZE
pwd := sanitized_input;
#else
pwd := pwd * pwd;
#endif
result := pwd;
[...]
```

(b)

Figure 1: (a) `Running Example`: Variability-dependent behavior in an SPL, and (b) its evolved version, where the #ELSE branch obfuscates the input.

A precise Taint Analysis must distinguish that `pwd` is safe only when `SANITIZE` is enabled. Traditional static analyses struggle with such variability [13]. A naive solution would be to analyze each configuration separately (in Figure 1 (a), one corresponding to having `SANITIZE` enabled and another in which it is not enabled), but this quickly becomes infeasible, as the number of variants grows exponentially with the number of features [14].

Moreover, this approach redundantly re-analyzes code that is common (denoted by [...] in Figure 1) across configurations. Another common shortcut is to merge all possible behaviors into a single model, but this sacrifices precision—e.g., marking `pwd` as always tainted, even when `SANITIZE` is active [4].

To address these limitations, family-based techniques such as *variational lifting* have been proposed [2, 6, 11, 14, 16]. These techniques analyze all configurations simultaneously while avoiding redundant analysis by evaluating common code like the statements before the `#IFDEF` in Figure 1 only once.

However, these techniques largely ignore **evolution**. In realistic SPLs, most code remains unchanged between versions [5], yet existing static analyses often discard all prior results, recomputing everything from scratch—even for minor edits. This inefficiency due to lack of result reuse is a key limitation of current variability-aware approaches. For example, modifying the sanitization logic, as highlighted in Figure 1 (b), would trigger full re-analysis across all configurations, despite the change affecting only a small fragment of the code.

Memoization techniques help alleviate this problem by caching intermediate results and reusing them across versions [5, 17]. Yet, prior work has focused either on handling variability in space (via lifting) or in time (via memoization), but not both. This separation becomes problematic in complex static analyses such as data-flow analysis, where both spatial and temporal redundancies can significantly impair performance.

To address this gap, this paper presents an interpreter-based framework that integrates variational lifting and memoization to support efficient and reusable static analysis of evolving SPLs. Our approach enables variability-aware execution by performing family-based static analysis without the need to analyze each configuration separately. In addition, it supports incremental and reusable evaluation by reusing previously computed results across program versions, thereby avoiding redundant computations and improving scalability over time. The key contributions are the following:

(1) The development of classic data-flow analyses—such as Reaching Definitions and Live Variables—written in a language tailored for variability-aware interpretation;
(2) The implementation of a unified interpreter that combines variational execution and memoization to handle both configuration-space and temporal reuse efficiently[1];
(3) An empirical evaluation showing that this combination significantly reduces analysis time, especially in SPL evolution scenarios[2].

The framework was evaluated using a suite of variational benchmarks designed to simulate realistic software product line evolution. Results show that combining variational lifting with memoization delivers substantial improvements in execution time compared to traditional, non-reusable approaches. In the case of the Reaching Definitions analysis, in our empirical evaluation, this combination achieved performance gains of **56%** for the initial program versions and **63%** for the evolved versions relative to the baseline. The ability to reuse intermediate results both across configurations and between program versions proved essential for efficiently handling the challenges introduced by spatial and temporal variability. Overall, this work provides a scalable and reusable analysis strategy for SPLs, advancing the state of the art in variability-aware static analysis.

2 Background

This section introduces the core concepts underlying our approach, in the order they are used throughout the system. We begin by presenting the extended WHILE language used to write object programs. We then describe PCF+, the functional language used to implement analyses, followed by a discussion of the data flow analyses supported in our framework. Next, we explain how variability is represented using presence conditions and variational values. Finally, we outline the execution model that enables static analyses to be evaluated compositionally across the entire configuration space. All examples refer back to the motivating `SANITIZE` scenario from Figure 1.

2.1 Adapted WHILE Language

Object programs under analysis are written in an extended version of the WHILE language [9], augmented with conditional directives to express variability. The construct `#IFDEF SANITIZE` in Figure 1 is one such example. These directives are compiled into a uniform structure where each branch is annotated with its respective presence condition (e.g., `SANITIZE` or `¬SANITIZE`).

[1] https://doi.org/10.5281/zenodo.16934537
[2] benchmarks folder at https://doi.org/10.5281/zenodo.16934537

2.2 PCF+: Analysis Language

Analyses are implemented in PCF+ [11], a small functional language originally based on Plotkin's Programming Computable Functions (PCF) [10], which was later extended to support recursion, pattern matching, lists, and higher-order functions. PCF+ enables the definition of expressive and reusable analysis logic while retaining a minimal core language. It serves as the target language for variational lifting and execution in our framework.

In our setting, analyses such as data-flow computations are written as functions over PCF+ data types and evaluated over variational inputs. For example, the following function computes the length of a list:

```
length(lst) =
  if isNil(lst) then 0
  else 1 + length(tail(lst))
```

Figure 2: Example of a PCF+ function

This concise, functional style supports structural recursion and compositional reasoning, making it well-suited for defining interpretable static analyses within our variability-aware framework.

2.3 Data Flow Analysis

We support control-flow-based analyses such as Reaching Definitions, Live Variables, Available Expressions and Very Busy expressions. These analyses rely on a Control Flow Graph (CFG) and propagate sets of definitions or variables using standard equations.

In the example shown in Figure 1 (a), a Reaching Definitions analysis must track that the sanitized assignment only reaches the final statement `result := pwd` when `SANITIZE` is enabled. Conversely, the raw input reaches that point when `SANITIZE` is disabled.

This selective propagation is made possible by preserving presence conditions at every program point, ensuring that analysis results are both precise and configuration-aware.

2.4 Variability Representation

To support static analysis over multiple configurations, we rely on *variational values* [16]. A variational value encodes alternatives associated with different configurations, each guarded by a *presence condition* (PC)—a Boolean expression indicating the configurations in which that value is valid.

A PC captures feature-dependent variability. For example, the PC `SANITIZE` denotes all configurations where the feature `SANITIZE` is active, while `¬SANITIZE` represents those where it is not.

For the example in Figure 1 (a), the value of variable `pwd` after `#ENDIF` varies across configurations:

```
pwd = Var [
  ("sanitized_input", SANITIZE),
  ("input", ¬SANITIZE)
]
```

Figure 3: Possible values for `pwd` in Running Example

This representation allows the analysis to distinguish behavior per configuration while reusing common structure across variants, as represented by [...] in Figure 1.

Variational lists are similarly encoded as lists of variational elements, enabling shared representations when values differ but structure remains consistent.

2.5 Variability-Aware Execution

To support analyses that preserve variability during execution, we adopt a *variability-aware execution* model, which systematically lifts the semantics of PCF+ to operate over variational values [11]. Rather than executing the same program repeatedly for each configuration, this approach enables a single execution to track multiple configuration-specific values simultaneously, guided by presence conditions.

Our implementation was guided by the deep lifting framework presented by Shahin and Chechik [11], which defines a rewriting-based approach for lifting functional programs written in PCF. While our goals differ—focusing on static analysis of SPLs rather than general-purpose functional programs—many of the foundational principles and operational rules of their approach were adapted to our setting.

Building on this foundation, our model represents each value as a set of alternatives, each associated with a presence condition (PC). Execution proceeds by applying lifted counterparts of standard operators and constructs (e.g., function application, conditionals, pattern matching) to these variational values, while preserving their disjointness and total coverage over the configuration space.

Variability-aware function application, for instance, is defined as follows: given a lifted function and a lifted argument—each represented as a set of `(value, PC)` pairs—the application produces a new lifted result by computing the cross-product of input pairs and combining their PCs via conjunction. Only pairs with satisfiable presence conditions are retained [11].

Conditionals and pattern matching are similarly lifted. A lifted conditional first evaluates the Boolean guard, partitions the configuration space according to the outcome (true/false), and then restricts the evaluation of each branch to the corresponding subset of configurations.

Overall, variability-aware execution enables compositional reuse of shared computations and preserves correctness across all configurations without explicit enumeration. This lifted execution model serves as the foundation for our interpreter based analysis framework. Here, correctness refers to the formal criterion established in Shahin & Chechik [11], which requires that, for any given configuration, the result produced by the variability-aware analysis coincides with the result of executing the original (non-lifted) analysis on the corresponding product variant. In other words, lifting preserves the semantics of the original analysis for each configuration individually.

3 Method

To address the challenges discussed in Section 1, the inefficiency of per-configuration analyses and the loss of reusable results during SPL evolution—we implemented a novel interpreter-based framework that combines variability-aware execution and memoization[3]. This framework enables variability-aware and static analysis over evolving software product lines.

Figure 4 provides an overview of our approach. It illustrates how an analysis written in PCF+ is evaluated over variational inputs encoded from successive SPL versions, while previous evaluation results are reused through memoization.

The remainder of this section is structured to reflect the execution flow illustrated in Figure 4. Section 3.1 describes how the object program is encoded into a variational representation. Section 3.2 presents the static analysis logic written in PCF+. Section 3.3 introduces the memory component used for memoization. Section 3.4 explains how the interpreter runs the analysis over variational data. Section 3.5 discusses how SPL evolution is modeled, and Section 3.6 explains how the interpreter reuses results during incremental analysis. We describe each step of this process using the conditional sanitization example introduced earlier (Figure 1).

3.1 Encoding Object Program as Variational Data

Building upon the concept of variational values and presence conditions introduced in Section 2.4, we now describe how object programs are systematically encoded into a structured variational form suitable for variability-aware analysis. The original program, written in the extended WHILE language, is transformed into a `VarValor`-based representation that captures both control flow and configuration-dependent behavior. The encoding process follows two main steps, detailed below.

Step 1: Annotated Intermediate Representation (`StmtPC`*).* The program is first translated into a form where each statement is explicitly paired with a presence condition. This representation, called `StmtPC`, encodes conditional constructs such as `#IFDEF` using a disjoint annotation scheme. Each branch of a `Variant` is recursively translated under its corresponding condition (e.g., `SANITIZE` and ¬`SANITIZE`). Internally, presence conditions are implemented using Binary Decision Diagrams (BDDs) [3] to support compact and efficient symbolic reasoning.

To ensure label consistency across configurations, inactive statements are replaced with labeled `skip` commands. The label for such a `skip` is set to the negative of the label assigned to the active statement in the other branch. This convention enables uniform control-flow graph alignment and helps track which blocks are configuration-specific. For example, in the second block of Figure 5, the assignment under condition `SANITIZE` is labeled 2, and its complement `skip` is labeled −2.

Step 2: Deep Encoding into Variational Structures. Next, the annotated program is encoded into a deeply nested `VarValor`, using constructors such as `VarPair`, `VarString`, and `VarList`. This encoding captures the control-flow graph (CFG) structure while preserving variability at every level of the program representation.

Figure 5 illustrates the control flow of the example in Figure 1 (a). Each block corresponds to a labeled item, such as assignments,

[3] https://doi.org/10.5281/zenodo.16934537

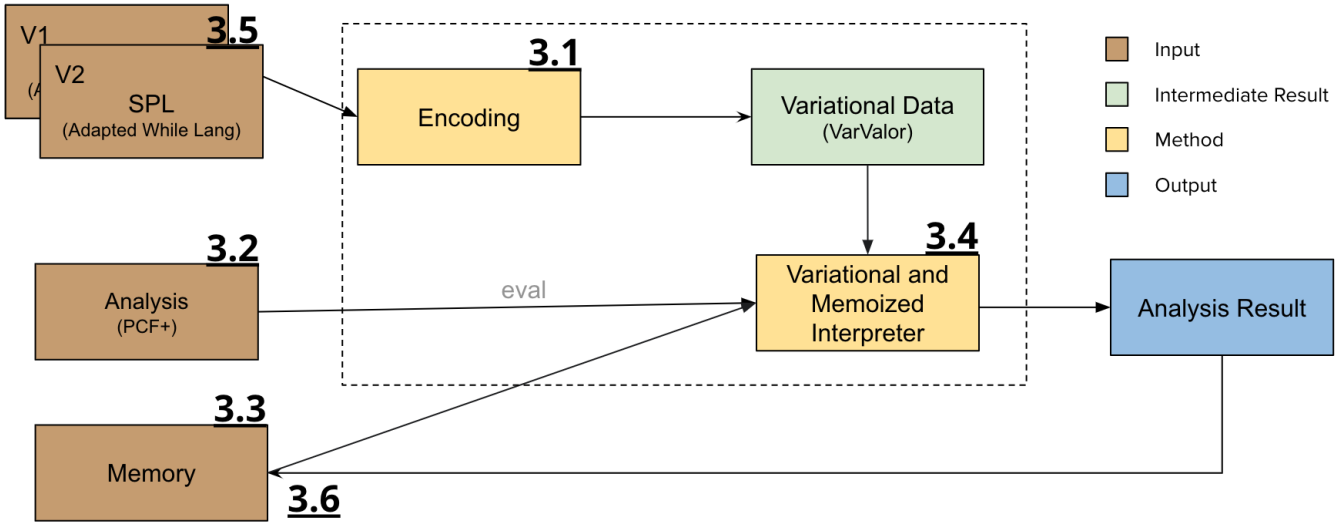

Figure 4: Overview of the proposed framework combining variational and memoized interpretation for static analysis of evolving SPLs.

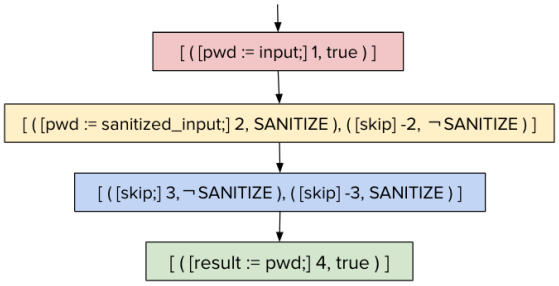

Figure 5: Variational control flow graph corresponding to the `Running Example`. Each node contains one or more labeled items (blocks) guarded by presence conditions.

tests and skip statements, annotated with a presence condition that encodes under which configurations the block is active.

These blocks are represented as deeply nested structures of `VarValor`. At the outermost level, the control-flow graph is encoded as a `VarList`, where each element corresponds to a block in the program. Each block itself is encoded as a `VarPair`: the first component stores the type of the statement—such as `"ASGN"`, `"IF"`, or `"SEQ"` for assignment, conditional, and sequence statements, respectively —annotated with its presence condition; the second component contains the content of the block, including its label, variables, literal values, or any substructures, all of which are also annotated with their corresponding presence conditions.

This encoding provides a uniform structure for the interpreter to traverse and evaluate. Each node is self-contained and explicitly declares the presence conditions under which it applies. As a result, the interpreter can perform variability-aware analysis without replicating the program per configuration, maintaining both *precision* (by tracking configuration-specific behavior) and *coverage* (by analyzing all configurations at once).

3.2 Analysis in PCF+

Static analyses in our framework are implemented as PCF+ programs. The framework currently includes four built-in data flow analyses: Reaching Definitions, Live Variables, Available Expressions, and Very Busy Expressions. However, the interpreter is general-purpose and capable of executing any static analysis expressed in PCF+ (a Turing-complete language), making it flexible and extensible for future use.

For example, the Reaching Definitions analysis determines, for each program point, the set of variable definitions that may reach it. This is expressed using two key sets: `genRD`, which contains the definitions generated within a basic block (i.e., assignments that introduce new definitions), and `killRD`, which contains the definitions killed by a block (i.e., definitions that are overwritten by a new assignment to the same variable). The function `flow` returns the set of control-flow edges in the program, represented as pairs of labels (ℓ', ℓ), indicating that control can pass from label ℓ' to label ℓ.

The analysis computes the sets $\texttt{RD}_{\texttt{entry}}$ and $\texttt{RD}_{\texttt{exit}}$ for each program point ℓ, following the standard data flow equations:

$$RD_{\text{entry}}(\ell) = \begin{cases} \emptyset & \text{if } \ell \text{ is the entry point} \\ \bigcup_{(\ell',\ell)\in flow} RD_{\text{exit}}(\ell') & \text{otherwise} \end{cases}$$

$$RD_{\text{exit}}(\ell) = \left(RD_{\text{entry}}(\ell) \setminus killRD(B_\ell)\right) \cup genRD(B_\ell)$$

This constitutes a *forward analysis*, where information propagates from the start of the control flow graph toward its end.

In the example shown in Figure 1, the statement `pwd := input` reaches the final assignment in all configurations. However, the

alternative assignment `pwd := sanitized_input` only reaches the final use `result := pwd` when the `SANITIZE` feature is enabled. This illustrates how presence conditions affect the propagation of definitions in variability-aware analyses.

3.3 Initial Memory

To enable reuse across evolving analyses, our interpreter uses an external memory structure to store previously computed results. This memory plays a key role in reducing redundant computations, especially when analyzing successive versions of an SPL.

The analysis begins either with an empty memoization memory—used when analyzing the SPL for the first time—or with a previously saved memory, allowing the reuse of results from earlier analyses across different versions.

3.4 Running the Interpreter

At the core of our approach lies an interpreter-based framework designed to execute static analyses written in PCF+ efficiently. The framework simultaneously handles variability across configurations (variability in space) and incremental changes due to software evolution (variability in time).

Initially, the interpreter receives three inputs: (1) a static analysis specified as a PCF+ program (e.g., Reaching Definitions), (2) a variational input representing the object program to be analyzed (such as the `Running Example` shown in Figure 1), and (3) a memoization memory structure. The variational input encodes the object program using the `VarValor` data type, which associates each value with a presence condition (e.g., `SANITIZE` or its negation). This representation ensures that multiple configurations can be analyzed simultaneously in a single execution, sharing computation across similar variants.

The interpreter propagates these presence conditions through the analysis computation, ensuring precision by maintaining distinct information for each variant. For instance, in the `Running Example`, the interpreter precisely identifies that the assignment *pwd* := *sanitized_input* is relevant only under the configuration where `SANITIZE` is enabled.

A critical aspect of data-flow analyses, such as Reaching Definitions, is their iterative fixpoint computation strategy. Functions like `genRD` and `killRD` are evaluated repeatedly until stabilization, leading to many redundant evaluations. To optimize this process, our framework supports selective memoization: expensive, pure functions can be declared memoizable, and their results reused automatically within a single analysis execution.

The technical criteria for which functions are safe to memoize, and how reuse is managed across program versions, are discussed in detail in Section 3.6.

Although this work does not formally verify the correctness of the memoization strategy or its interaction with variational lifting, practical confidence is built empirically by testing. The tests validate intermediate and final results of the analyses, including unit tests for auxiliary interpreter functions (e.g., presence propagation and Boolean operations on BDDs). Cross-validation between memoized and non-memoized interpreter configurations ensures result consistency. Additionally, the memoization mechanism relies on a purely functional model, where function outputs depend solely on their inputs without side effects, and cache keys are computed based on input arguments. This ensures outdated cached results are naturally bypassed when program changes occur, further reinforcing confidence in the interpreter's correctness.

Upon completion, the interpreter provides two outputs: (1) the variational result of the analysis, capturing the analysis outcome for all configurations (e.g., $\texttt{RD}_{\texttt{entry}}$ and $\texttt{RD}_{\texttt{exit}}$), and (2) the updated memoization memory, including all results reused and newly computed during this execution. This dual output enables efficient incremental analysis of evolving software product lines, enhancing both scalability and performance.

3.5 Evolving the SPL

Software Product Lines are not static: they evolve over time through feature additions, code modifications, or behavior refinements [5]. In our framework, such changes are captured by generating a new variational input while preserving the structure of the previous one as much as possible.

Consider the evolution of the `Running Example` described in Figure 1 (b), where the non-sanitizing branch is updated to apply a transformation.

Although the new version introduces a different assignment in the #else branch, most of the program remains unchanged. This is a typical scenario in evolving SPLs: incremental edits affect only specific configurations or blocks, while the rest of the structure is preserved.

3.6 Incremental Analysis via Memoization

Our framework supports incremental analysis by reusing results stored from previous executions. Rather than recomputing all function outputs from scratch, the interpreter consults a memoization cache to identify results that remain valid across versions.

This reuse is guided by the programmer, who specifies which functions in the PCF+ analysis are safe to memoize. Suitable candidates include functions that are (i) pure, i.e., have no side effects, and (ii) local, in the sense that their output depends only on the current block or input expression, rather than on global control or data-flow context. Examples include `genRD`, `killRD`, and `flow`, which operate on local syntactic structures such as basic blocks or labeled transitions. In contrast, functions like `RDentry` or `RDexit`, which depend on global control flow and the fixpoint of the entire analysis, are not memoized, as their correctness cannot be guaranteed when upstream context changes.

For each call to a memoized function, the interpreter checks whether a result for the given inputs—along with the associated presence condition—is already stored in memory. If so, the cached result is reused; otherwise, the function is evaluated, and the result is stored for future reuse. Calls to functions not marked for memoization are evaluated normally, without querying or updating the cache.

During incremental evaluation, the interpreter computes a hash based on a memoized function's input arguments and their associated presence condition. This design ensures that reuse occurs only when both the structure and the configuration context remain unchanged. The memoization mechanism does not rely on positional identity in the program but instead on structural and

semantic equivalence of inputs, making it resilient to reordered or relocated code.

For instance, in the evolved version of the `Running Example` in Figure 1 (b), the definitions `pwd:=input` and `result:=pwd` remain structurally and contextually unchanged across versions. Therefore, any memoized computations over these statements—such as `genRD` or `killRD`—can be safely reused. Only the updated fragment (e.g., `pwd := pwd * pwd` under ¬`SANITIZE`) results in new computations and cache insertions.

Additionally, since fixpoint-based analyses repeatedly invoke the same local functions during convergence, memoization further reduces redundant evaluations even within a single version. This dual reuse—across versions and across fixpoint iterations—contributes to the scalability and efficiency of the framework.

By focusing analysis effort on newly introduced or modified parts, our memoization strategy supports fast incremental updates, a critical capability for realistic SPL scenarios where changes are often small and localized [5]. The impact of this reuse mechanism is assessed empirically in Section 4.

4 Empirical Evaluation

We evaluated the proposed framework to assess the impact of combining variability-aware execution and memoization in static analysis of evolving SPLs. The evaluation addresses two questions: (i) how do these techniques affect runtime? (ii) how does memoization effectively reuse prior computations?

4.1 Experimental Setup

We compared four interpreter variants: a baseline interpreter (no variability, no memoization), a memoized-only version, a variational-only version, and a combined variational and memoized interpreter. When variability is not supported, the analysis is executed once per configuration.

The evaluation used a suite of synthetic programs written in the adapted WHILE language to simulate SPL-like scenarios. These programs were manually constructed to cover a range of variability and structural patterns. Specifically, they include examples with deeply nested variability (up to 5 levels), interprocedural control flow, loop-based arithmetic computation, variational initialization, and combinations of factorial and Fibonacci algorithms under feature control. Each program is provided in two versions: `v1` and `v2`. The transition from `v1` to `v2` introduces structural evolution, such as new computations after loops, additional conditional branches, increased loop bounds, or new arithmetic assignments based on prior values.

The size of the programs varies from approximately 6 to 30 statements, and the number of features ranges from 1 to 5, depending on the example. Programs like `deep_nested_variants` and `loop_multi_variant` exhibit maximal nesting of variability, while others like `factorial_rec_sim` emphasize iterative computation with conditional variability.

For each program, we executed Reaching Definitions, Live Variables, Available Expressions, and Very Busy Expressions analyses using all interpreter variants. Memoization targeted core functions like control-flow construction and free-variable computation, which are invoked repeatedly and yield high reuse potential.

Analyses were benchmarked using the Criterion library[4] to measure average runtime, with memoized interpreters serializing memory after `v1` and reusing it in `v2`. We collected runtime statistics and cache efficiency metrics (cache hits, misses, and size).

Results were aggregated into two output files: one containing per-analysis execution times and another reporting memoization statistics. These metrics support quantitative comparison of analysis performance and reuse benefits across versions.

Experiments ran on a 2017 MacBook Pro with 8 GB RAM and dual-core Intel i5 processor. Analysis outputs were cross-checked to ensure semantic equivalence across interpreters.

4.2 Results and Discussion

We present results for all four interpreter configurations (Base, Memoized, Variational, and Variational+Memoized) across the entire benchmark suite. This discussion highlights the **Reaching Definitions** analysis, illustrating performance and reuse behavior. Although we omit detailed figures for the other analyses—Live Variables, Available Expressions, and Very Busy Expressions—due to space constraints, we observed broadly similar trends in runtime and memoization reuse across them. The definitions of these classic analyses follow standard formulations in static analysis literature and were implemented analogously in our framework. Complete benchmark outputs, including per-analysis metrics and results for all configurations, are available in our online repository[5].

4.2.1 Runtime Performance. Figure 6 reports cumulative runtimes for the **Reaching Definitions** analysis on the first program version (v1) and its evolution (v2). The **Variational+Memoized** (VMemoized) interpreter is the clear winner, slashing execution time by **56 %** on v1 (429.7 ms → 188.6 ms) and **63 %** on v2 (710.0 ms → 261.6 ms) relative to the Base interpreter.

Memoization alone already provides substantial performance benefits, reducing execution time by 30% on v1 and 44% on v2. Even in the initial run, repeated function calls—such as recursive traversals of the control-flow graph—can be resolved using cached results, avoiding redundant computations and lowering runtime. Variability-aware execution shows mixed results: in v1, the overhead of managing presence conditions slightly outweighed reuse opportunities, leading to a small slowdown. In contrast, v2 benefited from sharing across variants, resulting in a 6% speed-up over the Base interpreter.

VMemoized inherits the strengths of both approaches while mitigating their individual weaknesses. It removes redundant computations *within* each configuration (memoization), reuses results *across* configurations (variational lifting), and—crucially for evolving SPLs—reuses cache entries between program versions. As a result, it delivers the fastest execution times across the board.

These results are specific to the Reaching Definitions analysis under the benchmark conditions described, and they highlight the practical value of integrating both memoization and variability-aware techniques in static analysis of evolving SPLs.

4.2.2 Memoization Efficiency. Table 1 reports the cache metrics collected across all programs when using the VMemoized and the

[4] http://www.serpentine.com/criterion/
[5] https://doi.org/10.5281/zenodo.16934537

Program	VMemoized						Memoized					
	Cache Size (B)		Miss		Hits		Cache Size (B)		Miss		Hits	
	v1	v2	v1	v2	v1	v2	v1	v2	v1	v2	v1	v2
Arithmetic Heavy	**10463**	**12041**	**33**	**37**	367	533	12155	14280	53	60	549	790
Deep Loop	7468	8968	**25**	**29**	299	401	**6788**	**8639**	36	42	197	319
Deeply Nested Variants	11435	**12228**	**29**	**33**	272	367	**6645**	12806	43	68	469	1372
Interprocedural	**8567**	**12163**	**29**	**41**	229	471	8627	13786	45	66	387	846
Loop with Multiple Variants	**11023**	**12581**	**37**	**41**	533	599	16101	19585	77	88	1363	2320
Nested Loop	**9643**	**10695**	**33**	**37**	367	533	11150	12665	53	60	506	740
Recursion Sim	9376	10929	**29**	**33**	401	467	**8790**	**10613**	43	49	282	468
Variational Initialization	**9521**	**10617**	**29**	**33**	272	367	22506	29804	111	136	593	1024

Table 1: Memoization efficiency for Reaching Definitions analysis. Bold values in the table indicate more efficient configurations, those that required less storage (for cache size) or performed fewer unique computations (for cache misses).

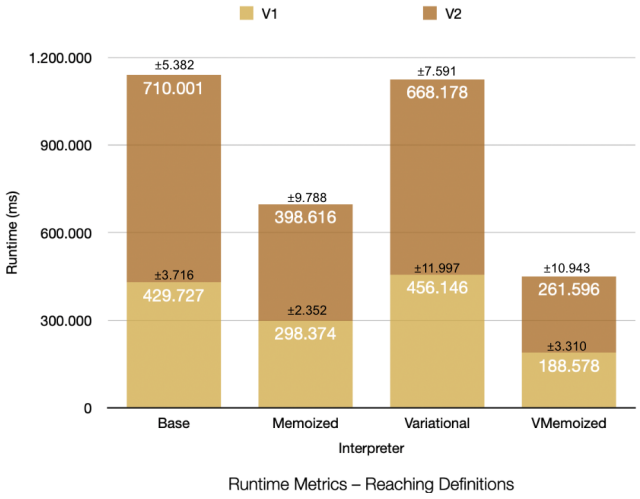

Figure 6: Runtime – Reaching Definitions Analysis

Memoized interpreters for the Reaching Definitions Analysis. The metrics include the cache size, which refers to the storage size of the file outputted with the memory state (in bytes); the number of cache misses, representing the unique memory state entries not found in cache; and the number of cache hits, which corresponds to the reused function results found in cache.

A notable pattern is that the **Variational and Memoized (VMemoized)** interpreter often yields **lower cache size and fewer cache misses** than the Memoized interpreter, even though both perform the same analyses. This shows that combining variability-aware execution with memoization results in more compact and efficient caching. By processing all variants simultaneously, it avoids redundant evaluations and maximizes reuse of cached results.

Programs like `Deep Nested Variants` and `Loop with Multiple Variants`, with many presence conditions and branches, demonstrate high cache reuse due to their structural regularity and the interpreter's ability to memoize sub-computations shared across configurations and versions.

Interestingly, a similar number of cache misses in v2 compared to v1 does not imply little reuse; rather, it often means that existing cache entries from the earlier version remained useful, requiring few new computations. This is especially true for programs that evolve but retain much of their structure.

The fact that cache hits are significantly higher than cache misses confirms that the interpreter effectively avoids recomputation by retrieving stored results. This reuse occurs both within a single program execution—such as in recursive or repetitive constructs—and across versions of the same program. The sum of hits and misses reflects the total number of function applications with unique input arguments during execution, indicating that a substantial portion of computations could be resolved through memoization, thereby improving performance and scalability.

4.3 Threats to Validity and Limitations

Performance results may have been affected by internal factors, such as system resource variability during benchmarking. Externally, generalizability is limited, as the benchmarks are synthetic and relatively simple. Construct validity may be impacted by the choice of memoized functions, which influences reuse patterns and memory usage. Although Criterion ensures statistically sound measurements, some runtime variability is inherent to empirical evaluations.

The framework is built on an extended PCF+, which lacks features like algebraic data types and lazy evaluation, limiting its expressiveness for more complex analyses. Memoization is function-level only; finer-grained caching (e.g., per statement) could enhance reuse. While the framework lacks formal verification, empirical tests support correctness. It is also a standalone tool, not yet integrated into broader workflows.

Variational lists are encoded as lists of variational elements, enabling structural sharing when values differ. This approach assumes uniform length across configurations, simplifying analysis but reducing expressiveness when list lengths vary between variants.

5 Related Work

Two main strategies have emerged to address the problem of analyzing large configuration spaces and evolving codebases: *variational lifting* and *memoization*.

Family-based analyses based on variational lifting aim to analyze all SPL configurations simultaneously, reusing results across shared

code paths to avoid redundant computation [14]. SPLLIFT [2] and Walkingshaw et al. [16] are examples that lift standard analyses (e.g., data flow) to operate over variational data structures. Other studies extended this idea to control-flow construction [6] and feature-sensitive analyses [11]. Despite their efficiency gains in analyzing variability in space, existing lifting frameworks struggle with more complex program structures, such as polymorphic lists and pairs, limiting their applicability to realistic languages and analyses.

Orthogonal to lifting, memoization has been used to improve reuse across software versions, addressing variability in time. Techniques such as regression verification [17] and caching of intermediate analysis results [5, 7] allow analyses to avoid recomputation when the code evolves incrementally. These approaches have proven effective in scenarios where most of the program remains unchanged between versions. However, they typically target single-configuration systems and do not consider the challenges of variability across configurations.

Few works have investigated how to integrate these two orthogonal techniques, variational lifting and memoization, into a unified framework. This gap becomes critical for complex analyses such as control-flow-based data-flow analysis, where redundancies exist both within and across configurations, and between software versions. Our work is, to the best of our knowledge, the first to propose an interpreter-based approach that systematically combines these techniques for SPL static analysis, while also supporting polymorphic data structures such as lists and pairs.

Our comparison with existing frameworks focuses primarily on *expressiveness*. A direct empirical comparison in terms of precision or performance would require reimplementing the same set of analyses across tools and executing them on a common benchmark, which is left for future work. Nonetheless, our empirical results (Section 4) provide evidence of the benefits of combining memoization and variational lifting in a unified setting.

6 Conclusion

This paper presented a framework that combines *variational lifting* and *memoization* to support scalable and reusable static analysis in the context of evolving SPLs. By adopting an interpreter-based architecture, the proposed approach enables the execution of static analyses—implemented as PCF+ programs—over variational representations of SPL code, where each value is annotated with a presence condition.

The empirical evaluation suggests performance benefits of combining variational lifting and memoization. Variability-aware execution reduced runtime by avoiding redundant computations across configurations, while memoization significantly improved reuse across program evolutions. Together, they achieved lower execution time and reduced analysis effort, demonstrating the scalability of the proposed framework for evolving Software Product Lines.

In future work, we intend to conduct larger-scale evaluations, as well as work on improving the implementation, that is, adding other static analyses, and integrating the tool within a developer workflow. Finally, we aim to explore the formal verification of the framework to strengthen its correctness guarantees.

ARTIFACT AVAILABILITY

The complete artifact (source code, datasets, and reproduction scripts) has been archived on Zenodo and can be accessed at https://doi.org/10.5281/zenodo.16934537. The archive corresponds to the exact version evaluated in this paper (tag v1.0.4) and is released under the MIT License.

ACKNOWLEDGMENTS

We thank Rodrigo Bonifácio, Paulo Borba, and the anonymous reviewers for suggestions to improve this work. Vander Alves and Leopoldo Teixeira were partially supported by CNPq (grants 313097/2021-6 and 315532/2021-1). We also acknowledge support from INES.[6]

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/978-3-642-37521-7_1

[2] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPLLIFT: statically analyzing software product lines in minutes instead of years. *SIGPLAN Not.* 48, 6 (June 2013), 355–364. doi:10.1145/2499370.2491976

[3] Randal Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. doi:10.1109/TC.1986.1676819

[4] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. 2015. Variability Abstractions: Trading Precision for Speed in Family-Based Analyses (Extended Version). arXiv:1503.04608 [cs.PL] https://arxiv.org/abs/1503.04608

[5] Christian Kröher, Lea Gerling, and Klaus Schmid. 2023. Comparing the intensity of variability changes in software product line evolution. *Journal of Systems and Software* 203 (2023), 111737. doi:10.1016/j.jss.2023.111737

[6] André Lanna, Thiago M. Castro, Vander Alves, Genaína Nunes Rodrigues, Pierre-Yves Schobbens, and Sven Apel. 2018. Feature-family-based reliability analysis of software product lines. *Inf. Softw. Technol.* 94 (2018), 59–81. doi:10.1016/J.INFSOF.2017.10.001

[7] Donald Michie. 1968. "Memo" Functions and Machine Learning. *Nature* 218, 5136 (1968), 19–22. doi:10.1038/218019a0

[8] Anders Møller and Michael I. Schwartzbach. 2018. Static Program Analysis. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/spa.pdf.

[9] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[10] G.D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255. doi:10.1016/0304-3975(77)90044-5

[11] Ramy Shahin and Marsha Chechik. 2020. Automatic and efficient variability-aware lifting of functional programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–27. doi:10.1145/3428225

[12] SPLC. 2024. Software Product Line Conference - Hall of Fame. https://splc.net/fame.html. Accessed: 2024-10-02.

[13] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. 2012. *Analysis Strategies for Software Product Lines*. Technical Report FIN-004-2012. School of Computer Science, University of Magdeburg, Germany.

[14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (July 2014), 1–45. doi:10.1145/2580950

[15] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*. ACM, Paris France, 57–64. doi:10.1145/3307630.3342414

[16] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, Portland Oregon USA, 213–226. doi:10.1145/2661136.2661143

[17] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. 2018. Verified Memoization and Dynamic Programming. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Vol. 10895. Springer International Publishing, Cham, 579–596. doi:10.1007/978-3-319-94821-8_34 Series Title: Lecture Notes in Computer Science.

[6] https://www.ines.org.br