University of Brasília

Exact Sciences Institute
Computer Science Department

A Framework of Memoization and Variational Lift using Interpreters

Tayná Larissa Fischer Vieira

Dissertation submitted in partial fullfilment of
the requirements to obtain a Master's Degree in Informatics

Advisor
Prof. Dr. Vander Ramos Alves

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Arcabouço de Memoização e Transformação Variacional usando Interpretadores

Tayná Larissa Fischer Vieira

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Brasília
2025

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

FV658f

Fischer Vieira, Tayná Larissa
A Framework of Memoization and Variational Lift using Interpreters / Tayná Larissa Fischer Vieira; orientador Vander Alves. Brasília, 2025.
84 p.

Dissertação(Mestrado em Informática) Universidade de Brasília, 2025.

1. Linhas de Produtos de Software. 2. Análise Estática de Linhas de Produtos de Software. 3. Execução Variacional. 4. Memoização. 5. Programação Funcional. I. Alves, Vander, orient. II. Título.

University of Brasília

Exact Sciences Institute
Computer Science Department

A Framework of Memoization and Variational Lift using Interpreters

Tayná Larissa Fischer Vieira

Dissertation submitted in partial fullfilment of
the requirements to obtain a Master's Degree in Informatics

Prof. Dr. Vander Ramos Alves (Advisor)
CIC/UnB

Prof. Dr. Paulo Borba
CIN/UFPE

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordinator of Graduate Program in Informatics

Brasília, March 26, 2025

Dedicatória

À Deus e a todos os que me ajudaram ao longo desta caminhada.

Agradecimentos

Agradeço, em primeiro lugar, a Deus, pela força, saúde e fé que me sustentaram ao longo de toda esta jornada acadêmica.

Ao meu marido, Thalles, minha eterna gratidão pelo amor, apoio incondicional e compreensão nos momentos mais desafiadores — especialmente quando precisei estar ausente em finais de semana e noites longas em prol deste sonho. Seu carinho — e o dos nossos cachorros — foi essencial para que eu chegasse até aqui.

Aos meus pais, Fernando e Marcia, por sempre acreditarem em mim e por serem meu porto seguro. Os valores que me transmitiram e a confiança que depositaram em mim foram fundamentais para que eu seguisse com coragem e determinação. À minha irmã Tayane, por, mesmo sem saber, me motivar a ser um bom exemplo.

Aos meus orientadores, Professores Vander e Leopoldo, agradeço profundamente pela dedicação, paciência e generosidade em compartilhar conhecimento ao longo de cada etapa deste processo. Obrigada por acreditarem no meu potencial e por me incentivarem a ir além. Agradeço também aos professores Rodrigo e Paulo, membros da banca, pelas contribuições e sugestões valiosas que enriqueceram significativamente este trabalho.

Ao meu colega de mestrado, Bruno, sou grata pelas inúmeras ajudas, pelas trocas de ideias e conselhos ao longo do caminho. Estendo também meu agradecimento a todos os amigos que, de diferentes formas, contribuíram com apoio, incentivo e companhia durante essa jornada.

Por fim, estendo meu agradecimento ao Exército, pela confiança depositada em mim e pela oportunidade de crescimento profissional, e aos meus colegas de trabalho — uma verdadeira família — pelo apoio e compreensão durante os momentos em que precisei conciliar as responsabilidades profissionais com os estudos acadêmicos.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Linhas de Produtos de Software (SPLs) possibilitam o desenvolvimento sistemático de sistemas configuráveis ao organizar produtos como famílias que compartilham funcionalidades comuns e diferem em recursos selecionados. No entanto, a análise estática em SPLs enfrenta desafios de escalabilidade devido à variabilidade no espaço (entre configurações) e no tempo (entre revisões de software). Esta dissertação propõe um framework baseado em interpretadores que combina lifting variacional e memoização para suportar análises estáticas escaláveis e reutilizáveis em SPLs em evolução. As análises são implementadas como programas PCF+ e executadas sobre representações variacionais dos programas, anotadas com condições de presença. A memoização permite reutilizar resultados computados anteriormente entre diferentes versões do programa, reduzindo cálculos redundantes e contribuindo para a melhoria do desempenho. O framework foi avaliado com programas que simulam cenários realistas de evolução de software. Os resultados demonstram que o uso combinado de lifting variacional e memoização reduz efetivamente o tempo de execução, evidenciando as vantagens de abordar ambas as dimensões da variabilidade. Este trabalho contribui com uma infraestrutura reutilizável para análises baseadas em fluxo de controle em SPLs, além de fornecer evidências empíricas da sua eficiência.

Palavras-chave: Linhas de Produtos de Software, Análise Estática de Linhas de Produtos de Software, Execução Variacional, Memoização, Programação Funcional

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 dissertation 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 programs, annotated with presence conditions. Memoization mechanisms allow the reuse of previously computed results across program evolutions, reducing redundant computations and contributing to performance improvements. The framework is evaluated using benchmarks simulating realistic software evolution scenarios. Results show that the combined use of variational lifting and memoization effectively reduces execution time, demonstrating 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

Contents

List of Figures

List of Tables

Listings

Abreviations and Acronyms List

AST Abstract Syntax Tree.

BDD Binary Decision Diagram.

CFG Control-Flow Graph.

DFA Data Flow Analysis.

GQM Goal Question Metric.

PC Presence Condition.

PCF Programming Computable Functions.

PCF+ Programming Computable Functions extended.

SPL Software Product Line.

Chapter 1

Introduction

Software Product Line (SPL) have become a prominent approach to the development of highly configurable systems. An SPL is a set of related software systems that share a common set of core assets but differ in certain features to meet specific requirements of different customers or contexts [1]. Well-known examples include the *Linux Kernel*, which supports a wide range of hardware platforms and configurations, and other real-world systems listed in the *Feature-Aware Modeling and Evolution (FAME)* repository [2], such as automotive software, mobile platforms, and web applications.

SPL promote systematic reuse and enable the automated generation of tailored products, increasing productivity and reducing time-to-market [1]. However, a key challenge in SPL development is managing **variability**— the differences and commonalities among products in the product line [3].

This variability occurs in two main dimensions: **variability in space**, which refers to the set of alternative features across configurations, and **variability in time**, which arises due to software evolution and updates. Effectively handling both spatial and temporal variability is essential for the scalability and maintainability of SPL-based development [3].

Static analysis is an automated technique used to derive insights about program behavior without executing the code. It plays a crucial role in identifying potential issues such as the unintended exposure of sensitive information or runtime errors like arithmetic overflows [4]. However, traditional **static analysis techniques** often struggle to address the challenges posed by the variability inherent in Software Product Lines (SPLs) [5].

Common forms of static analysis include *Data Flow Analysis (DFA)* (which tracks how data flows through a program), *Control Flow Analysis* (which determines the possible execution paths), and *Type Checking* (which verifies that operations are applied to compatible data types) [6].

A major limitation of conventional static analysis approaches is their reliance on **per-configuration analysis**, where each product variant is analyzed separately. This approach becomes computationally expensive due to the exponential number of possible configurations in an SPL [7]. Moreover, it repeatedly analyzes common parts across configurations, leading to duplicated effort and poor scalability.

Similarly, when software evolves over time, reanalyzing each new version from scratch introduces significant **redundancy**, as only a small portion of the code typically changes between versions, and much of the analysis result could remain valid [3, 8].

To address these challenges, two independent techniques have emerged: **variational lifting**, which enables family-based analysis of multiple configurations simultaneously [9, 10, 11, 12], and **memoization**, which supports reuse of previously computed results to avoid redundant calculations across software versions [8, 13, 14].

While previous work has explored each technique in isolation, few studies have investigated how to effectively combine both *variability in space* and *variability in time*, especially in the context of more complex static analyses such as control-flow-based data-flow analysis [12, 7]. Moreover, current lifting frameworks struggle to handle polymorphic structures such as lists and pairs [12]. To overcome these limitations, we developed a new framework that combines both dimensions of variability, making static analyses more scalable and reusable.

This dissertation proposes an interpreter-based framework to efficiently analyze evolving SPLs, combining variability-aware execution and memoization. Static analyses are implemented as programs in the functional language PCF+ [15] and executed over a variational representation of the SPL, in which values are annotated with presence conditions. Memoization enables reuse of previous analysis results across software revisions, reducing redundant computations and supporting complex analyses such as Data Flow Analysis.

To evaluate the proposed approach, we conducted an empirical study comparing the performance of the interpreter with baseline interpreters (without any technique, only variational-aware and only memoized) in terms of execution time and reuse efficiency. Benchmarks included a set of variational programs with synthetic evolutions to simulate real-world SPL updates. We measured runtime differences and cache reuse metrics such as cache hits and cache misses. The results show that combining variational lifting with memoization achieves better performance in many scenarios, particularly when analyzing program evolutions where only part of the structure changes. This suggests the relevance of combining both variability dimensions to improve static analysis in SPLs.

In summary, the contributions of the present work are the following:

1. Implementation (in PCF+) of Data Flow Analyses for a simple imperative language[1];

2. The development of an interpreter-based execution model framework that lifts analyses written in PCF+ into semantically equivalent variability-aware and memoized analyses[2];

3. An experiment[3] comparing the proposed method to its non-variational and non-memoized counterparts, using a benchmark of simulated SPLs to assess performance and memoization stats.

The remainder of this dissertation is structured as follows:

- Chapter 2 presents background concepts on Variability Representation, Data Flow Analyses, the Adapted While Language, and PCF+;
- Chapter 3 discusses related work and defines the problem addressed in this dissertation;
- Chapter 4 describes the proposed approach, including the interpreter-based framework and core implementation techniques;
- Chapter 5 presents the empirical evaluation and results;
- Chapter 6 concludes this dissertation, highlighting the contributions, limitations, and directions for future work.

[1] `https://github.com/fischertayna/lifting-framework/tree/main/src/Language/Analysis/DFA`

[2] `https://github.com/fischertayna/lifting-framework`

[3] `https://github.com/fischertayna/lifting-framework/tree/main/benchmarks`

Chapter 2

Background

This section presents key concepts directly related to our research. Section 2.1 discusses how variability is represented in our study. Sections 2.2 and 2.3 introduce the two languages used in our framework: the former describes the language of the programs to be analyzed, while the latter describes the language in which the analyses are implemented. Finally, Section 2.4 presents essential concepts related to Data Flow Analyses.

2.1 Variability Representation

To efficiently represent variability in data during program analysis, we rely on variational values, denoted as `V[A]`. A variational value represents a value that can take different forms depending on the configuration. Each alternative is annotated with a presence condition, specifying under which configurations that alternative is valid. This concept follows the principles introduced by Walkingshaw et al. [10], where `V[A]` is used to express variation compactly and maintain the *choice-as-partition invariant*—ensuring that alternatives are associated with mutually exclusive presence conditions.

Building on this foundation, we represent lists with variability using the structure `List[V[A]]`. In this representation, the variational list is just an ordinary list of variational elements. This allows for shared elements across variants and provides fine-grained control over variability at the element level. For example, the variation between the lists `[1,5,10]` and `[1,2,10]` can be compactly represented as `[1, <5,2>, 10]`, where `<5,2>` is a variational value that encodes the difference in the second position. Section 4.2.1 presents our implementation of the variational data structure.

This representation was chosen based on the principles of variational lifting, as proposed by Shahin and Chechik [12]. Their work demonstrates how static analyses can be automatically lifted to operate directly over variational data structures, allowing analysis across all configurations simultaneously in a scalable manner. Furthermore, this repre-

sentation builds on prior work already implemented by another member of our research group, which facilitated its integration and reuse in our framework.

Advantages

- **Compact Sharing:** Common elements among variants can be represented only once, significantly reducing redundancy in scenarios where list variants differ in only a few elements.
- **Simple Integration:** Since the underlying list structure remains unchanged, traditional list operations (e.g., mapping, folding) can be reused with minimal adaptation.
- **Efficiency for Uniform-Length Variants:** When variants share the same length, `List[V[A]]` allows compact and efficient traversal and manipulation.

Disadvantages

- **Inflexibility in Length Variation:** A key limitation of `List[V[A]]` is its inability to represent variants of different list lengths. All variants must have the same number of elements, with variability restricted to the values, not to the structure.
- **Limited Expressiveness:** Certain kinds of structural variation (e.g., optional elements, variable length configurations) cannot be modeled directly and require workarounds such as padding with dummy elements or extending to more expressive encodings like `OList[A]` [10].
- **Complexity in Semantics:** Reasoning about the meaning of a variational list requires tracking and propagating presence conditions for each individual element, which can increase cognitive and implementation complexity.

Despite its limitations, `List[V[A]]` represents a pragmatic trade-off between expressiveness and simplicity. It is particularly well-suited in contexts where the primary variation lies in values rather than structural changes in the data.

2.2 Adapted WHILE Language

WHILE is a simple imperative language introduced in the book *Principles of Program Analysis* [6]. A program written in WHILE consists of statements, which can be either individual commands or sequences of commands executed in order.

An example of a WHILE program that computes the Fibonacci sequence of the number stored in ‘x‘, leaving the result in ‘r‘, is:

$$[a := 0]^1;\ [b := 1]^2;\ [i := 0]^3;\ [r := 0]^4;\ \text{while}\,[i < x]^5\,\text{do}\,([r := a]^6;\ [a := b]^7;\ [b := r{+}b]^8;\ [i := i{+}1]^9;\,)$$

For convenience, data flow information is addressed as a label and is associated with the assignment statement, the tests that appear within conditionals or loops and the skip statement. These labeled elements are known as *elementary blocks*.

The language has the following syntactic categories:

- $a \in \mathbf{AExp}$ arithmetic expressions
- $b \in \mathbf{BExp}$ boolean expressions
- $c \in \mathbf{Stmt}$ statements
- $x, y \in \mathbf{Var}$ variables
- $n \in \mathbf{Num}$ numerals
- $l \in \mathbf{Lab}$ labels
- $op_a \in \mathbf{Op_a}$ arithmetic operators
- $op_b \in \mathbf{Op_b}$ boolean operators
- $op_c \in \mathbf{Op_c}$ relational operators

The syntax of the language is defined as follows:

$$a ::= x \mid n \mid a_1\ op_a\ a_2$$

$$b ::= \text{true} \mid \text{false} \mid \text{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2$$

$$S ::= [x := a]^l \mid [\text{skip}]^l \mid S^1; S^2 \mid \text{if}\,[b]^l\,\text{then}\,S_1\,\text{else}\,S_2 \mid \text{while}\,[b]^l\,\text{do}\,S$$

2.2.1 Introducing Variability in WHILE

To support variability in the WHILE language, we extend its syntax with choice constructs, inspired by the principles of the choice calculus [16]. In this model, variation is represented as explicit choices between alternatives, each associated with a presence condition that determines under which configurations the alternative is selected.

Following this approach, we introduce conditional compilation directives into WHILE using constructs similar to preprocessor languages. Specifically, we add the directives `#IFDEF`, `#ELSE`, and `#ENDIF`, which allow parts of a program to be included or excluded based on a presence condition (e.g., a feature flag).

The extended syntax for statements in WHILE now includes variability constructs as follows:

$$S ::= \ldots \mid \#\text{IFDEF}\, pc\ S_1\ \#\text{ELSE}\ S_2\ \#\text{ENDIF}$$

Here, `pc` represents a presence condition. The semantics is straightforward: if `pc` holds in the current configuration, the program executes statement S_1; otherwise, it executes S_2. Note that both branches are always required — each variability point must specify an alternative path, maintaining alignment with the choice-as-partition principle from the choice calculus, where each alternative is exclusive and exhaustive.

This design enables a clean and modular way to represent feature-dependent program variants in WHILE and also ensures that the structure aligns with existing theories of variation programming and variational analysis.

Example: Feature-Dependent Fibonacci Calculation

In Listing 2.1, we present an adapted version of the Fibonacci program that includes an optimization enabled only if the feature **FAST_FIB** is defined:

$$
\begin{array}{l}
[a := 0]^1;\ [b := 1]^2;\ [i := 0]^3;\ [r := 0]^4; \\
\text{while}\, [i < x]^5\, \text{do}\, (\\
\quad \#\text{IFDEF FAST\_FIB} \\
\quad [r := b]^6; \\
\quad \#\text{ELSE} \\
\quad [r := a]^7;\ [a := b]^8; \\
\quad \#\text{ENDIF} \\
\quad [b := r + b]^9;\ [i := i + 1]^{10}; \\
)
\end{array}
$$

Listing 2.1: Fibonacci With Optimization in WHILE

This version allows the program to switch between two implementations depending on whether **FAST_FIB** is enabled.

Abstract Syntax Tree (AST) for the Adapted WHILE Language

The following Haskell data type (Listing 2.2) represents the adapted syntax, extending WHILE's standard statements with a **Variant** constructor to handle conditional compilation:

Listing 2.2: Abstract Syntax Tree for Adapted WHILE Language in Haskell

```haskell
type Program = Stmt

data Stmt = Assignment Id AExp Label
          | Skip Label
          | Seq Stmt Stmt
          | IfThenElse (BExp, Label) Stmt Stmt
          | While (BExp, Label) Stmt
          | Variant PresenceCondition Stmt Stmt
```

The Fibonacci example can be represented as a program using the abstract syntax of the While language encoded in Haskell. Listing 2.3 shows the corresponding Haskell representation using the Program data type. Each statement is annotated with a label for identification, and the presence condition FAST_FIB is modeled using the Variant constructor to represent compile-time variability. The While loop iteratively computes Fibonacci values, with different behaviors depending on whether the FAST_FIB feature is enabled.

Listing 2.3: Example of a program in While (Hakell)

```haskell
exampleProgram :: Program
exampleProgram =
  Seq (Assignment "a" (Const 0) 1)
  $ Seq (Assignment "b" (Const 1) 2)
  $ Seq (Assignment "i" (Const 0) 3)
  $ Seq (Assignment "r" (Const 0) 4)
  $ While (LTExp (Variable "i") (Variable "x"), 5)
      (Seq
        (Variant "FAST_FIB"
          (Assignment "r" (Variable "b") 6)
          (Seq (Assignment "r" (Variable "a") 7)
               (Assignment "a" (Variable "b") 8))
        )
```

```
14          (Seq (Assignment "b" (Add (Variable "r") (Variable "b
               ↪ ")) 9)
15              (Assignment "i" (Add (Variable "i") (Const 1))
                   ↪ 10)
16          )
17      )
```

2.3 PCF+

PCF+ is an extension of the Programming Computable Functions (PCF), a typed functional language introduced in 1977 by Gordon Plotkin [15].

The design of PCF+ supports, at a primary level, expressing analyses such as DFA, which are central to static analysis and abstract interpretation techniques. The language includes:

- Primitive data types such as integers, booleans, strings, and lists;
- Pair types and pattern matching constructs;
- Conditional expressions and recursive function definitions;
- First-class support for higher-order functions.

In our implementation, PCF+ serves as the core analysis language interpreted over both non-variational and variational input. Programs written in PCF+ define analysis logic that is applied to a model of the program under analysis, encoded as input data. The example below (Listing 2.4) illustrates a simple PCF+ function that computes the length of a list of integers.

Listing 2.4: Example of PCF+ program

```
1 length :: [int] -> int
2 length (lst) {
3     if ((isNil (lst)))
4         then 0
5         else 1 + length (tail (lst))
6 }
```

Because of its simplicity and formal grounding, PCF+ enables a clear separation between the analysis logic and the mechanics of variability and memoization handling in our framework. This makes it a fitting choice for implementing reusable and interpretable analysis pipelines.

2.4 Data Flow Analysis

Data flow analysis (DFA) is a technique used in static program analysis to track how data values propagate through a program [6]. It helps in program optimization and correctness verification by analyzing properties of variables and expressions across different program points.

A key structure underlying DFA is the Control-Flow Graph (CFG). A CFG is a directed graph that represents the flow of execution in a program. Each node in the graph corresponds to a *basic block*, which is a sequence of consecutive statements with a single entry and a single exit. The edges represent control flow transitions between these blocks. CFGs are foundational for many static analysis techniques, and they serve as the basis over which data flow equations are formulated and solved [6].

In Data Flow Analysis, each analysis defines transfer functions and equations that describe how information propagates across the nodes and edges of the CFG, and solutions are computed using techniques such as *worklist algorithms* or *fixed-point iteration* [6].

There are several fundamental data flow analyses, including:

- **Reaching Definitions**
- **Live Variables Analysis**
- **Available Expressions Analysis**
- **Very Busy Expressions Analysis**

Each of these analyses is typically formulated as a set of equations that must be solved iteratively over the CFG using techniques such as *worklist algorithms* or *fixed-point iteration*.

2.4.1 Reaching Definitions

Reaching definitions analysis determines which variable definitions may reach a given point in the program. A definition of a variable x at a program point p *reaches* a point q if there exists a path from p to q that does not redefine x.

Mathematically, the analysis computes sets of reaching definitions at each program point:

- $gen_{RD}(B^\ell)$: The set of definitions generated by the block l.
- $kill_{RD}(B^\ell)$: The set of definitions overwritten by the block l.

The result of the analysis is a pair of $RD_{entry}(\ell)$ and $RD_{exit}(\ell)$. They are defined as:

$$RD_{entry}(\ell) = \begin{cases} (x, ?) \mid x \in FV(S_\star) & \text{if } \ell = init(S_\star) \\ \bigcup RD_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star) & \text{otherwise} \end{cases}$$

$$RD_{exit}(\ell) = (RD_{entry}(\ell) \setminus kill_{RD}(B^\ell)) \cup gen_{RD}(B^\ell)$$

This analysis is forward, meaning information flows from entry to exit of the CFG [6].

2.4.2 Live Variables Analysis

Live variable analysis determines which variables are *live* at each program point. A variable x is *live* at a point p if its value may be used on some path from p without being overwritten.

The equations for live variable analysis are:

- $gen_{LV}(B^\ell)$: The set of definitions generated by the block l.
- $kill_{LV}(B^\ell)$: The set of variables used in the block l before being defined.

The result of the analysis is a pair of $LV_{entry}(\ell)$ and $LV_{exit}(\ell)$. They are defined as:

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in final(S_\star) \\ \bigcup LV_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_\star) & \text{otherwise} \end{cases}$$

$$LV_{entry}(\ell) = (LV_{exit}(\ell) \setminus kill_{LV}(B^\ell)) \cup gen_{LV}(B^\ell)$$

This is a backward analysis since information flows from exit to entry of the CFG [6].

2.4.3 Available Expressions Analysis

Available expressions analysis determines which expressions have already been computed and are available for reuse at each program point. An expression e is available at point p if every path from the entry to p evaluates e without redefining any of its operands.

The equations for this analysis are:

- $gen_{AE}(B^\ell)$: Expressions generated at block l.

- $kill_{AE}(B^\ell)$: Expressions invalidated by the block l.

The result of the analysis is a pair of $AE_{entry}(\ell)$ and $AE_{exit}(\ell)$. They are defined as:

$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in init(S_\star) \\ \bigcap AE_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star) & \text{otherwise} \end{cases}$$

$$AE_{exit}(\ell) = (AE_{entry}(\ell) \setminus kill_{AE}(B^\ell)) \cup gen_{AE}(B^\ell)$$

This is a forward analysis used for optimizations like common subexpression elimination [6].

2.4.4 Very Busy Expressions Analysis

Very busy expressions analysis identifies expressions that must be evaluated along every path from a program point to the exit without being invalidated.

The equations are:

- $gen_{VB}(B^\ell)$: Expressions computed at block l that are not invalidated before their next use.
- $kill_{VB}(B^\ell)$: Expressions invalidated at block l.

The result of the analysis is a pair of $VB_{entry}(\ell)$ and $VB_{exit}(\ell)$. They are defined as:

$$VB_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell \in final(S_\star) \\ \bigcap VB_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_\star) & \text{otherwise} \end{cases}$$

$$VB_{entry}(\ell) = (VB_{exit}(\ell) \setminus kill_{VB}(B^\ell)) \cup gen_{VB}(B^\ell)$$

This is a backward analysis, useful in code motion optimizations such as loop-invariant code motion [6].

Chapter 3

Related Work and Problem Statement

Understanding the challenges of variability-aware static analysis requires examining how variability in space and time impacts program analysis. This chapter begins with Sections 3.1 and 3.2, which discuss related works that apply various techniques to address variability in space and time. Section 3.3 presents the initial attempts at writing data-flow analyses using a combined approach. Finally, Section 3.4 illustrates the challenges of integrating both spatial and temporal variability, highlighting the specific problem addressed in this work.

3.1 Variability in Space

Variability in space refers to the coexistence of multiple product variants within a Software Product Line (SPL), each defined by a unique combination of features. The primary challenge of analyzing this dimension lies in the exponential growth of possible product configurations, as every combination of features results in a distinct variant. Performing static analysis on each individual variant is computationally exhaustive due to the vast number of combinations [7].

To address this challenge, several approaches have been proposed in the literature aiming to perform **family-based analysis** [17], where all variants are analyzed simultaneously rather than individually.

One approach involves the use of efficient data structures that explicitly represent variability within computations. Walkingshaw et al. [10] proposed **variational data structures**, such as variational lists and trees, which allow a compact representation of all possible values and behaviors across variants. These data structures are designed to be aware of presence conditions, enabling operations to be performed once while accounting

for all configurations that a value belongs to. This strategy reduces redundancy and improves performance compared to analyzing variants separately.

Another important line of work is the automatic lifting of variability-aware functional programs. Shahin et al. [12] introduced a technique that automates the transformation of conventional (non-variability-aware) programs into their variability-aware counterparts. Their approach leverages **variational programming** principles to systematically lift standard operations so that they can operate over variational data types. This allows analysts to reuse existing code and analyses without manually rewriting them for each product configuration, making variability-aware analysis more practical and scalable.

In addition, there is a rich body of work focused on lifting traditional Data Flow Analyses to handle variability at scale. For instance, Brabrand et al. [18] proposed an Intraprocedural Data Flow Analysis framework tailored for SPLs, while Bodden et al. [9] introduced SPL^{LIFT}, an approach capable of scaling static analysis to large product lines efficiently by lifting existing analyses. Other notable works include scalable analyses using variability-aware control flow graphs [19] and systematic derivations of correct variability-aware analyses [20].

These contributions illustrate the diversity of approaches—ranging from novel data structures to systematic lifting frameworks—that aim to address the challenge of scalable analysis in the presence of variability. Acknowledging these works provides context for the growing interest in techniques that improve the performance and practicality of program analyses in SPLs.

3.2 Variability in Time

Variability in time, on the other hand, refers to the evolution of software through successive revisions and updates. Each new version of the software may introduce changes that impact its behavior, leading to the need for reanalysis. It is important to note that variability in time does not necessarily imply the presence of a Software Product Line (SPL); any evolving system is subject to such temporal changes. However, this problem becomes even more critical in the context of SPLs, where both temporal and spatial variability must be managed concurrently [3].

A static analysis that fails to consider variability in time will often redundantly reanalyze portions of the software that have not changed, leading to inefficient computations [3]. Empirical works on SPL evolution, such as the study by [8], provide detailed insights into the frequency and intensity of variability changes within SPLs. Their analysis of the Linux Kernel and other SPLs, including coreboot, BusyBox, and axTLS, revealed that changes to variability information (such as feature models, build configurations, and

code artifacts) occur infrequently and typically affect only small parts of the system. This contradicts the assumption that variability information would undergo substantial modifications over time. Instead, their findings suggest that the majority of changes are localized and minimal, implying that SPLs evolve in a more controlled and incremental manner than previously thought.

3.3 Combining the Variability Dimensions for Data-Flow Analysis

Analyzing programs with variability in both space and time simultaneously introduces unique challenges. Without variability-awareness, static analysis must be performed independently for each possible variant, leading to redundancy and scalability issues due to the combinatorial explosion of configurations. Conversely, without memoization, reanalyzing evolving programs forces recomputation of unchanged parts, which undermines the benefits of incremental analysis.

To address both dimensions simultaneously, our initial approach combined the memoization strategy with the automatic lifting technique proposed by Shahin et al. [12]. This strategy, originally suggested by another member of our research group in his master's dissertation, aimed to reuse existing analyses by lifting them to operate over variational data structures while memoizing intermediate results to avoid repeated computations across program evolutions.

As a first experiment, we expressed a simple data-flow analysis using the lifted functional programming style. For example, we wrote an analysis that returned a list of variable-assignment counts using `foldl`. However, we encountered practical limitations when using common data structures such as lists and pairs in the deeply lifted setting. Functions like `foldl'` and `map'` triggered unexpected type errors during compilation due to incompatibilities in how polymorphic types were handled by the lifting framework.

These issues stem from a well-documented limitation of the deep lifting mechanism [12]: while the framework assumes that a type T is lifted to $T^{\uparrow}$, it does not properly account for the distinction between $[T^{\uparrow}]$ (a list of lifted elements) and $[T]^{\uparrow}$ (a lifted list). In practice, deep lifting a list yields $[T^{\uparrow}]$, which clashes with the framework's expectations in contexts that require $[T]^{\uparrow}$. This discrepancy becomes particularly problematic when using higher-order functions like `foldl`, whose types are polymorphic over the structure being lifted.

Listing 3.1 shows a basic analysis that uses `foldl` to accumulate results. In the deep-lifted version (Listing 3.2), `foldl` is replaced with `foldl'` to handle variational input.

However, the compiler produces a type mismatch error, expecting `Var b` but finding `[a]`, as it incorrectly assumes the type should be $[T]^{\uparrow}$ rather than $[T^{\uparrow}]$.

Listing 3.1: Analysis that returns number of Assignments of a Program

```
analyze :: CFG -> [(String, Integer)]
analyze cfg =
    let _ns = _nodes cfg
        _fns = filter isFnRoot _ns
        totalAssignments = foldl (\acc fnRoot -> acc ++
          ↪ assignmentsPerVar cfg fnRoot) [] _fns
    in totalAssignments
```

Listing 3.2: Deep lifted Analysis

```
analyze :: (Var CFG) -> [((Var String), (Var Integer))]
analyze cfg  = let _ns = _nodes' cfg
                   _fns = filter' isFnRoot _ns
                   totalAssignments = foldl' (\acc fnRoot ->
                     ↪ acc `++ assignmentsPerVar cfg fnRoot
                     ↪ ) [] _fns
               in totalAssignments
```

This type mismatch reflects a broader limitation of the lifting framework: support for deeply lifting polymorphic structures like lists and pairs remains incomplete. Key components—such as the `caseSplitter` function and the `VClass` instances for lists and pairs—were either commented out or lacked implementation, requiring manual adjustments (e.g., replacing CFG with a lifted variant in generated code).

Ultimately, these limitations revealed fundamental gaps in existing tool support for expressive and reusable analyses. As a result, we were motivated to design our own interpreter-based framework that integrates variability-awareness and memoization more seamlessly. Our goal was to enable efficient and scalable static analysis of evolving Software Product Lines (SPLs), without relying solely on fragile automatic lifting mechanisms.

These challenges are not merely theoretical. In practice, evolving Software Product Lines (SPLs) frequently combine variability in both space and time—introducing changes to feature-controlled behavior across versions. To better illustrate these difficulties, we now present a motivating example written in the Adapted WHILE Language. This example highlights how both forms of variability can affect static analysis, particularly in the context of security-relevant computations. It also sets the stage for understanding why

combining variability-awareness with memoization is essential for supporting efficient, incremental analysis of evolving SPLs.

3.4 Problem Statement

To illustrate the challenges of analysing an evolving SPL, we consider a simple program written in the Adapted WHILE Language that models user input handling. This program captures a security-relevant scenario: the propagation of potentially unsanitized user input.

Base SPL (with variability)

$[pwd := input]^1;$
#IFDEF `SANITIZE`
$[pwd := sanitized\_input]^2$
#ELSE
$[skip]^3$
#ENDIF
$[result := pwd]^4$

Variant `SANITIZE`

$[pwd := input]^1;$
$[pwd := sanitized\_input]^2;$
$[result := pwd]^4$

Variant ¬`SANITIZE`

$[pwd := input]^1;$
$[skip]^3;$
$[result := pwd]^4$

Listing 3.3: Base Program representing a SPL with variability and its concrete variants.

The program in Listing 3.3 introduces a feature flag named `SANITIZE` that controls whether user input is passed through a sanitization step. When this feature is not enabled, the password is propagated without sanitization, representing a typical scenario in which variability may introduce a potential vulnerability. Here, the construct `sanitized_input` is a stub that represents a complex sanitization function applied to the input.

This example highlights the challenge of analyzing variability in *space* — different configurations may lead to different behaviors with security implications. For instance, while the sanitized variant ensures safer propagation, the alternative one exposes the system to unsafe input usage.

A **traditional analysis without variability-awareness** would require analyzing each variant separately, resulting in duplicated effort. Conversely, a **variability-aware analysis** evaluates the entire variational program in a single pass, sharing computation

across common parts and producing a **variational result** that encodes outcomes for all configurations.

However, another key dimension arises in practice: the evolution of the program over time, or **variability in time**. Even small changes between versions may impact the analysis results and, under traditional approaches, necessitate full recomputation for every configuration.

Consider the evolution of the non-sanitizing variant to an obfuscation-based transformation:

Evolved SPL

$[pwd := input]^1;$
#IFDEF `SANITIZE`
$[pwd := sanitized\_input]^2$
#ELSE
$[pwd := pwd * pwd]^3$
#ENDIF
$[result := pwd]^4$

Variant `SANITIZE`

$[pwd := input]^1;$
$[pwd := sanitized\_input]^2;$
$[result := pwd]^4$

Variant ¬`SANITIZE`

$[pwd := input]^1;$
$[pwd := pwd * pwd]^3;$
$[result := pwd]^4$

Listing 3.4: Evolved Program representing a SPL after its evolution

This seemingly minor evolution changes the behavior of the non-sanitizing variant. Rather than skipping the sanitization step, it now transforms the password using a reversible obfuscation (e.g., squaring), which may give the illusion of protection but does not provide true sanitization. This change affects the program's security properties while retaining most of the original structure.

In a traditional setting, even this small change would trigger a full re-analysis of all variants. However, **memoization** offers an opportunity to reuse previously computed analysis results for unchanged parts — such as the sanitizing branch or the final assignment — significantly reducing computational overhead.

Importantly, **memoization and variability-awareness address orthogonal concerns**. Variability-aware analysis improves scalability across feature combinations, while memoization enhances efficiency across program versions. When combined, they enable more effective and scalable analysis of evolving SPLs.

While prior work has demonstrated the benefits of analyzing variability in isolation, **Data Flow Analysis in evolving SPLs** remains a largely unexplored area, especially when aiming to leverage reuse across both product configurations and program versions.

Problem Statement

There is a gap in current approaches to Data Flow Analysis for Software Product Lines: existing techniques do not adequately support simultaneous reuse across product configurations and evolving program versions.

Chapter 4

Method

This chapter presents the proposed approach for performing variability-aware and memoized static analysis using an interpreter-based method. In this approach, the analysis is represented as a *program* written in PCF+, which is evaluated by an interpreter designed to support variability-aware inputs. The outcome of this evaluation corresponds to the result of the static analysis. The interpreter integrates mechanisms for both variability handling and memoization, enabling efficient and optimized execution of the analysis. The framework is available in our source code repository[1].

Section 4.1 provides an overview of the method, Section 4.2 describes Core Data Structures, Section 4.3 explains the encoding of the Adapted WHILE Language, Section 4.4 gives more details about the interpreter framework, and, finally, Section 4.5 describes the implementation of static analyzes as executable programs.

4.1 Method Overview

This section provides a high-level overview of the proposed method. The approach integrates variability-aware static analysis with memoization techniques to improve the efficiency of evaluating variational programs. Figure 4.1 illustrates the full pipeline of the method.

The method takes as inputs:

- A program to be analyzed, written in the Adapted WHILE Language, which may contain variability through `#IFDEF`, `#ELSE`, and `#ENDIF` directives.
- A static analysis written in PCF+, such as Reaching Definitions, Live Variables, Available Expressions, or Very Busy Expressions. Alongside the analysis program,

[1] `https://github.com/fischertayna/lifting-framework`

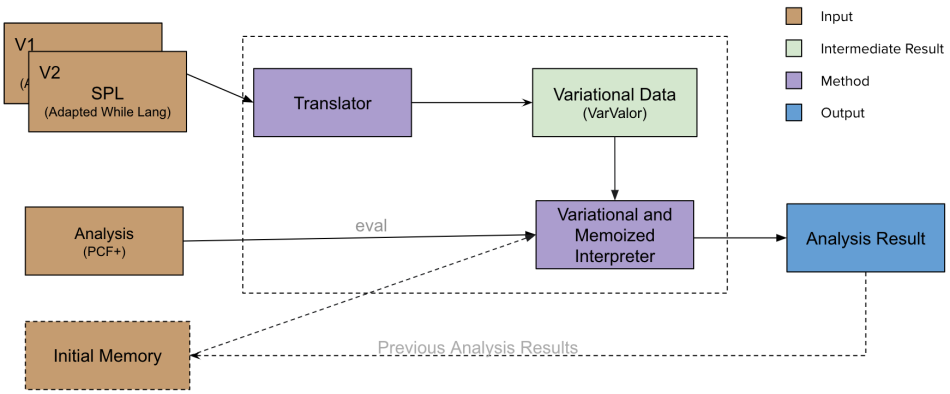

Figure 4.1: Overview of the Variability-Aware Static analysis pipeline

a list of function names is provided to indicate which analysis functions should benefit from memoization during interpretation.

- Previous analysis results (referred to as *Initial Memory*) that can be reused in the current analysis to improve performance through memoization.

The pipeline is composed of the following steps:

1. **Program:** A variational program written in the Adapted WHILE Language is provided. It can represent different versions (e.g., V1, V2) of an SPL.

2. **Translation:** The input program is passed through a Translator component, which encodes the program into a variational data representation (`VarValor`). This representation captures configuration-specific behaviors and structures in a uniform format suitable for variational evaluation.

3. **Evaluation:** The analysis, written in PCF+, is evaluated by the Variational and Memoized Interpreter using the variational representation of the program as input. This interpreter integrates two key techniques:

   - *Variational Lift*: to ensure the analysis accounts for all configuration variants.
   - *Memoization*: to avoid recomputation by leveraging previously stored intermediate results from past analyses.

4. **Memory:** During evaluation, for certain predefined functions, the interpreter consults the Memory, which may contain results from previous analyses. If the required value has already been computed, it is reused; otherwise, the interpreter computes the result and stores it in memory. This mechanism enables efficient reuse of analysis results across future executions or revised versions of the SPL.

5. **Output:** The result of the analysis is a lifted value in variational form (e.g., a list of live variables or definitions per configuration), which we refer to as the **Analysis Result**.

4.2 Core Data Structures

4.2.1 `VarValor`: Variational Representation of Values

The `VarValor` data type lies at the heart of the variational evaluation model. As introduced in Chapter 2.1, this structure builds directly on the concept of variational values (`V[A]`), where each alternative value is annotated with a presence condition indicating the configurations under which it is valid [10]. `VarValor` generalizes this concept by extending traditional value types to their variational counterparts, allowing the interpreter to simultaneously represent and evaluate multiple configurations in a compact and systematic manner.

For example, while a standard `Valor` might represent an integer simply as `Integer`, in `VarValor`, it is represented as `Var Integer`—a variational value consisting of multiple annotated alternatives. Each alternative corresponds to an integer value paired with a presence condition, precisely following the structure of `V[A]` discussed earlier. This distinction is crucial: `Var Integer` is not a single value but a set of possible values, each tied to specific feature combinations.

Similarly, for lists, the type `VarValor` adopts the structure `VarList [VarValor]`, in alignment with the `List[V[A]]` representation described in Chapter 2.1. This representation allows for fine-grained variability at the element level, enabling compact sharing and efficient manipulation of value-level variation across configurations.

Formally, the structure is defined as:

Listing 4.1: Var definition

```
1 type Val a = (a, PresenceCondition)
2 newtype Var t = Var [Val t]
```

For example, a variational integer might look like:

Listing 4.2: Var example

```
1 Var [(5, A), (10, ~A)]
```

This means:

- The value **5** is the result if the feature `A` is enabled.

- The value **10** is the result if the feature `A` is disabled.

Thus, `Var Integer` encapsulates a list of pairs (i, pc), where i is an integer and pc is the presence condition under which i is valid. This general structure applies similarly to booleans, strings, and other types, enabling fine-grained tracking of variability throughout the evaluation process.

The `VarValor` type supports the following variational data constructors:

- **VarInteger**: A variational integer, represented as `Var Integer`.
- **VarBool**: A variational boolean, represented as `Var Bool`.
- **VarString**: A variational string, represented as `Var String`.
- **VarList**: A variational list, i.e., a list of `VarValor` elements.
- **VarPair**: A variational pair, consisting of two `VarValor` elements.

Formally, the definition of `VarValor` is given by:

Listing 4.3: VarValor definition

```
1 data VarValor
2 = VarInteger { int :: Var Integer }
3 | VarBool { bool :: Var Bool }
4 | VarString { str :: Var String }
5 | VarList { list :: [VarValor] }
6 | VarPair { pair :: (VarValor, VarValor) }
7 deriving (Show, Eq, Read)
```

Each `VarValor` instance contains not just a value, but a collection of alternative values, each annotated with its corresponding presence condition. These presence conditions are internally represented as Binary Decision Diagrams (BDDs), ensuring efficient manipulation and combination of configuration constraints.

Operations on `VarValor` maintain the associated presence conditions by leveraging logical combinators such as `andBDD`, `orBDD`, and `notBDD`, thus preserving correctness in variational evaluation.

Additionally, `VarValor` is immutable and implements the `Hashable` type class. This immutability and hashability are essential to support optimizations such as memoization, which rely on identifying repeated computations based on input values and their presence conditions.

4.2.2 Presence Conditions (`Prop`)

The variability in `VarValor` is controlled using presence conditions, represented through the Prop type. A presence condition indicates under which configuration(s) a given value or operation is active in a (SPL). Technically, these conditions are encoded as a Binary Decision Diagram (BDD), a canonical and compact representation of Boolean expressions, allowing for efficient logical manipulation and reasoning.

Listing 4.4 shows the definition of the Prop type, which internally stores a BDD node (`b`) and a human-readable name (`pname`) for identification purposes. The name also facilitates the serialization and deserialization processes via the `Show` and `Read` instances, enabling more readable representations and easier reconstruction of presence conditions.

Listing 4.4: Prop definition

```
1
2 data Prop = Prop
3   { b :: DDNode
4   , pname :: String
5   }
```

Presence conditions are created using utility functions such as `newBDD` and `mkBDDVar` (Listing 4.5). These functions encapsulate Binary Decision Diagram (BDD) creation and feature-variable lookup using a shared BDD manager.

The function `mkBDDVar` is used to create a presence condition associated with a specific feature (e.g., `"A"` or `"DEBUG"`). It ensures that every feature name corresponds to a unique BDD variable index within the global manager. This is accomplished using the auxiliary function `lookupVar`, which maps each feature name to a unique integer index. If the feature has already been seen, its index is reused; otherwise, a new index is allocated and stored in an internal hash table.

This mechanism guarantees that presence conditions remain consistent and comparable across different parts of the program, even when created in different contexts. The symbolic name is preserved in the `Prop` structure for debugging and display purposes, while the BDD node ensures efficient logical reasoning.

Listing 4.5: Presence Condition (Prop) creation

```
1 -- Associates a BDD node with a human-readable feature name
2 newBDD :: DDNode -> String -> Prop
3 newBDD node name = Prop node name
4
```

```haskell
-- Creates a new presence condition (Prop) for a given feature
mkBDDVar :: String -> Prop
mkBDDVar name =
  let i = lookupVar name              -- Ensures uniqueness of variable index
      r = ithVar manager i            -- Gets the corresponding BDD node
   in newBDD r name                   -- Constructs the Prop
```

For example, Listing 4.6 shows the creation of a presence condition named `UNSAFE_DIV`, which represents a particular variant of the SPL where an unsafe division (e.g., division by zero) may occur.

Listing 4.6: Prop UNSAFE_DIV

```haskell
propUnsafeDiv :: Prop
propUnsafeDiv = mkBDDVar "UNSAFE_DIV"
```

This presence condition is then used in the base program (Listing 3.3) to guard specific assignments—such as setting $pwd := sanitized\_input$ under SANITIZE and *skip* under its negation—enabling the encoding of mutually exclusive variants within a single unified representation.

Key operations on `Prop`:

- **Logical Conjunction** (`andBDD`): Combines two presence conditions.
- **Logical Disjunction** (`orBDD`): Merges two presence conditions.
- **Negation** (`notBDD`): Computes the complement of a presence condition.
- **Satisfiability** (`sat`): Checks if a presence condition is satisfiable.
- **Unsatisfiability** (`unsat`): Determines if a presence condition is always false.

4.2.3 Key Functions

- `union :: Var t -> Var t -> Var t`: Merges two `Var` instances, preserving all possible values and their presence conditions.
- `compact :: Var t -> Var t`: Groups values with the same presence conditions to minimize redundancy.

- `apply :: Var (a -> b) -> Var a -> Var b`: Applies a function over variational values, preserving presence conditions.
- `(+++): VarValor -> VarValor -> VarValor`: Performs a union-like operation for variational values, ensuring consistency in presence conditions.

4.3 Translator: Encoding Variability in the WHILE Language

The Adapted WHILE Language introduces variability through conditional compilation directives (`#IFDEF`, `#ELSE`, `#ENDIF`). These directives enable the representation of multiple program variants within a single codebase, allowing different configurations to be analyzed efficiently.

The translator phase encodes the Adapted WHILE programs into an intermediate data represented by a `VarValor` structure that preserves presence conditions and enables simultaneous evaluation of multiple configurations.

Statements are translated into a hierarchical structure composed of nested pairs and strings (`VarPair` and `VarString`). This section illustrates the encoding process using the problem introduced in Section 3.4.

4.3.1 Translating Programs

In the WHILE Language, a program consists of a single statement, which can be a composition of multiple sub-statements (e.g., sequences, conditionals, loops). To enable variability-aware analysis, we need to translate this program into a representation that explicitly encodes presence conditions. This translation is performed by the function `encodeStmt`, which produces a `VarValor` representation of the program, enriched with variability information.

Listing 4.7: encodeStm

```
1 encodeStmt :: Stmt -> VarValor
2 encodeStmt stmt = encodeStmtPC' (stmtToStmtPC ttPC stmt)
```

Here, `ttPC` denotes the presence condition that is always true—i.e., the program is initially assumed to be fully present under all configurations. The `stmtToStmtPC` function is responsible for converting the original `Stmt` into an intermediate form, `StmtPC`, in which each statement is annotated with a presence condition that reflects under which configurations it is active.

Listing 4.8: stmtToStmtPC

```
stmtToStmtPC :: PresenceCondition -> Stmt -> StmtPC
```

This transformation is particularly important for handling variability constructs such as Variant, which are used to represent `#IFDEF` directives. When a $Variantpcs1s2$ is encountered, `stmtToStmtPC` eliminates it by recursively translating both alternatives: the first with the presence condition pc and the second with its negation $notBDDpc$. In other words, the Variant is removed, and its presence condition is propagated into the respective branches. This propagation ensures that each part of the program retains precise presence information, enabling accurate analysis per configuration.

For example, the structure:

Listing 4.9: Variant

```
Variant pc s1 s2
```

is translated into:

Listing 4.10: Variant Translation

```
SeqPC (stmtToStmtPC (andBDD pc0 pc) s1)
      (stmtToStmtPC (andBDD pc0 (notBDD pc)) s2)
```

where $pc0$ is the outer (inherited) presence condition. When a statement is inactive under a certain condition, it is replaced by a `Skip` statement with a negated label, preserving structure but indicating that the statement is not executed in that configuration.

The full StmtPC datatype is defined as:

Listing 4.11: StmtPC

```
data StmtPC
    = AssignmentPC Id AExp Label PresenceCondition
    | SkipPC Label PresenceCondition
    | SeqPC StmtPC StmtPC
    | IfThenElsePC TestExp StmtPC StmtPC PresenceCondition
    | WhilePC TestExp StmtPC PresenceCondition
    deriving (Eq, Ord, Show)
```

Each constructor explicitly carries a `PresenceCondition`, capturing the variability of the program structure.

Once a statement is converted into `StmtPC`, it is further encoded into `VarValor` through `encodeStmtPC' :: StmtPC -> VarValor`.

Translating `#IFDEF` directives (Variants)

Consider the following code fragment from Listing 3.3, which introduces a variability point controlled by the presence condition `SANITIZE`:

#IFDEF SANITIZE
  $[pwd := sanitized\_input]^2$
#ELSE
  $[skip]^3$
#ENDIF

Listing 4.12: Sanitization variant example

As with other `#IFDEF` directives, this construct is translated by removing the explicit Variant node and encoding each alternative with its corresponding presence condition. The sanitizing assignment is guarded by the condition `SANITIZE`, and the fallback `skip` instruction is guarded by the negation ¬`SANITIZE`. The inactive branch under each configuration is replaced with a no-op (`SKIP`) labeled with a negative identifier, such as `-2` or `-3`, to distinguish it from semantically meaningful statements.

The result of encoding this fragment into VarValor is shown in Listing 4.13:

Listing 4.13: VarValor encoding of the `SANITIZE` variability

```
variantUnsafeDiv = VarPair (
  VarString (Var [("SEQ", ttPC)]),
  VarPair (
    VarPair (
      VarString (Var [("ASGN", propSanitize),("SKIP", notBdd
        ↪ propSanitize)]),
      VarPair (
        VarString (Var [("2", propSanitize),("-2", notBdd
          ↪ propSanitize)]),
        VarPair (
          VarString (Var [("pwd", propSanitize)]),
          VarPair (
            VarString (Var [("VAR", propSanitize)]),
            VarString (Var [("sanitized_input", propSanitize)
              ↪ ])
            )))),
    VarPair (
```

```
15        VarString (Var [("SKIP", notBdd propSanitize),("SKIP",
             ↪ propSanitize)]),
16        VarPair (
17          VarString (Var [("3", notBdd propSanitize),("-3",
               ↪ propSanitize)]),
18          VarString (Var [("", notBdd propSanitize)])))
19  )
```

Listing 4.14: encodeStmtPC'

In this encoding:

- `"ASGN"` denotes the assignment present in the sanitizing variant.
- `"SKIP"` is used to maintain structural consistency in the non-sanitizing branch.
- The label `"2"` corresponds to the original line of the sanitizing assignment; `"-2"` marks the placeholder instruction in the alternate branch.
- `propSanitize` encodes the presence condition `SANITIZE`.
- `notBDD propSanitize` expresses the condition where `SANITIZE` is absent.

This translation preserves both behavioral branches within the variational representation and enables the analysis engine to reason about their effects jointly. The use of labeled `SKIP` nodes ensures that control-flow and labeling information remain consistent across configurations, even when certain branches are inactive.

Each branch of the Variant contributes to the final `VarValor`, making the variability explicit and analyzable.

Translating Sequences

Sequences are translated by recursively encoding each body of the sequence $S_1; S_2$:

This structure is transformed into the following `VarValor` representation:

```
1  encodeStmtPC' (SeqPC s1 s2) =
2      VarPair (VarString (Var (presencePairsStmt "SEQ" ttPC)),
3               VarPair (encodeStmtPC' s1, encodeStmtPC' s2))
```

Listing 4.14: encodeStmtPC'

Each s_1 and s_2 are recursively encoded while preserving its presence conditions.

Translating assignments

Assignments in the variability-aware representation are expressed using the constructor `AssignmentPC`, which extends a standard assignment with a presence condition. The translation into `VarValor` is performed by the following clause of the `encodeStmtPC'` function:

Listing 4.15: encodeStmtPC' for Assignment

```
encodeStmtPC' (AssignmentPC v e l pc) =
    VarPair (VarString (Var (presencePairsStmt "ASGN" pc)),
             VarPair (VarString (Var (presencePairsLabel (
               ↪ show l) pc)),
                      VarPair (VarString (Var [(v, pc)]),
                        ↪ encodeAExpToVarValor e pc))))
```

The function `presencePairsStmt` is used to enrich the `"ASGN"` tag with the presence condition pc, and additionally includes a `"SKIP"` tag for the negated condition ¬ `pc` (i.e., `notBDD pc`). This allows the representation to explicitly indicate that the assignment does not occur in configurations where the presence condition does not hold. Similarly, the `presencePairsLabel` function appends a negated label (e.g., `"-3"`) for the skip branch, clearly signaling that this line is inactive in that configuration. In Listing 4.13 we can see how the statements were translated.

Translating While Loops

Loops statements (*While*) are translated in a similar manner, maintaining the presence conditions:

$$\textbf{while } [cond]^l \textbf{ do } s$$

This structure is transformed into the following `VarValor` representation:

Listing 4.16: encodeStmtPC' for While loops

```
encodeStmtPC' (WhilePC (cond, l) body pc) =
    VarPair (VarString (Var (presencePairsStmt "WHILE" pc)),
             VarPair (VarString (Var (presencePairsLabel (
               ↪ show l) pc)),
                      VarPair (encodeBExpToVarValor cond pc,
                        ↪ encodeStmtPC' body)))
```

Conditionals are translated in the same way as loops.

4.4 The PCF+ Interpreter

The PCF+ interpreter serves as the execution engine for analysis written in the PCF+ language. It is designed to support both variability-aware computation and efficient memoization. This dual capability attempts to enable analyses that are both configuration-sensitive and performance-efficient, which is essential in the context of Software Product Lines (SPLs).

4.4.1 Overview

The interpreter evaluates expressions (*Exp*) in the context of a runtime environment (*RContext*) and an evolving memory (*Mem*) that stores intermediate and memoized results. The interpreter supports a variety of language constructs, such as integers, booleans, strings, lists, pairs, conditionals, and user-defined functions.

The entry point for program execution is defined by the `evalP` function:

Listing 4.17: entry point of execution

```
1 evalP :: Program -> [String] -> (VarValor -> Mem -> (VarValor
    ↪ , Mem))
2 evalP (Prog fs) memoizedFunctionNames =
3   \input mem ->
4     let memoizedIdents = map Ident memoizedFunctionNames
5         initialFContext = updatecF ([(Ident "n", input)], [],
            ↪  memoizedIdents) fs
6         context = initialFContext
7     in eval context (Call (Ident "main") [EVar (Ident "n")])
        ↪ mem
```

This function takes as input a program (composed of a set of function definitions) and a list of function names to be memoized. It produces a function that receives an initial input value and memory state and returns the result of evaluating the program starting from the `main` function.

The list `memoizedFunctionNames` plays a key role in controlling which function calls will benefit from memoization during execution. Each function identifier in this list is included in the interpreter context, so that the evaluation engine can distinguish memoized from regular function calls. This enables selective memoization: only functions explicitly listed will have their results cached and reused, while others are evaluated normally.

The memoization mechanism is discussed in more detail in the Section 4.4.3.

The core evaluation function is defined by the `eval` function:

Listing 4.18: eval function

```
1 eval :: RContext -> Exp -> Mem -> (VarValor, Mem)
```

This function takes a runtime context, an expression to be evaluated, and the current memory state. It returns a pair with the evaluated value (as a `VarValor`) and an updated memory.

4.4.2 Variational Execution

The interpreter models variability using the `VarValor` type, which wraps values together with associated *presence conditions*. These presence conditions represent under which feature configurations a value is valid. Variational values are propagated and combined throughout the evaluation, according to the lifted PCF+ semantics proposed by Shahin and Chechik [12].

For instance, conditional branching is implemented with presence condition partitioning:

Listing 4.19: EIf

```
1 EIf eCond eThen eElse
```

The condition expression is evaluated, and its result is used to partition the execution context. The branches are then evaluated under the corresponding presence conditions and recombined accordingly.

4.4.3 Memoized Execution

To improve performance and avoid redundant computations, the interpreter supports function-level memoization. Function calls whose identifiers appear in a predefined memoization list are stored and reused using a key-based caching mechanism.

The core logic of memoization is implemented in the `memoizedCall` function (Listing 4.20), which ensures that repeated calls to the same function with the same input are not recomputed.

Listing 4.20: memoizedCall

```
1 memoizedCall :: RContext -> Ident -> [VarValor] -> Mem -> (
    ↪ VarValor, Mem)
2 memoizedCall context fId paramValues mem =
```

```
  let (Fun _ _ decls fExp) = fromJust $ lookup fcontext fId
      name = getIdentString fId
      paramBindings = zip decls paramValues
      funcKey = createFuncKey name paramValues
  in runState (do
      retrieveOrRun name funcKey (\() -> do
        let (evalResult, mem') = eval (paramBindings,
          ↪ fcontext, memoizedFunctionNames) fExp mem
        return evalResult)
    ) mem
```

Internally, `memoizedCall` generates a cache key using the function name and a hash of the arguments. This is done by the helper `createFuncKey` function:

Listing 4.21: Creating the memoization key

```
data FuncKey = FuncKey
  { funcName :: String
  , funcArgsHash :: Int
  } deriving (Eq, Show, Read)

createFuncKey :: String -> [VarValor] -> FuncKey
createFuncKey name args = FuncKey
  { funcName = name
  , funcArgsHash = hash args
  }
```

This hash-based key ensures that each distinct function call is uniquely identified. The evaluation is then delegated to the memoization engine via `retrieveOrRun`:

Listing 4.22: retrieveOrRun operator

```
retrieveOrRun :: (KeyMemory k v m, Show v, Show k)
              => String -> k -> (() -> State m v) -> State m
  ↪ v
retrieveOrRun name x t = do
  mlookup x >>= \case
    Just v -> return v
    Nothing -> do
      v <- t ()
      mupdate x v
```

```
9       return v
```

If a match is found in memory, the stored result is returned; otherwise, the function is evaluated and its result stored for future reuse.

Memoization Strategy and Cache Validity

A key advantage of our memoization strategy is that explicit cache invalidation is unnecessary. This is because the interpreter operates under a pure functional programming model, where a function's result depends solely on its input and has no side effects.

Whenever the analyzed program changes, it naturally alters the input to affected function calls. Since the cache key is based on a hash of the input arguments, any change in structure, content, or variability of the input automatically produces a different key. As a result, the interpreter will not find a matching entry in the cache and will recompute the result as needed.

This property ensures that:

- Cached results are reused only when inputs are exactly the same.
- Modifications in the program's structure or content naturally lead to recomputation.
- The cache remains correct and consistent without requiring manual invalidation logic.

Memoization System

The caching mechanism is implemented via the `KeyMemory` type class, which abstracts key-value stores in a stateful monad:

Listing 4.23: Memoization type class

```
1 class KeyMemory k v m where
2   mlookup :: k -> State m (Maybe v)
3   mupdate :: k -> v -> State m ()
```

A simple instance is provided using an associative list of entries, where each value includes a reuse counter:

Listing 4.24: Key-value memory with reuse counters

```
1 type KeyValueArray k v = [(k, (v, Int))]
2
3 instance (Eq k, Show k, Show v) => KeyMemory k v (
    ↪ KeyValueArray k v) where
```

```
4   mlookup a = State (\s ->
5     case lookup a s of
6       Just (v, count) ->
7         let updated = map (\(k', (v', c)) -> if k' == a then
              ↪ (k', (v', c + 1)) else (k', (v', c))) s
8          in (Just v, updated)
9       Nothing -> (Nothing, s))
10  mupdate a v = State (\s ->
11    let newS = case lookup a s of
12          Just (_, count) -> (a, (v, count)) : filter (\(k',
              ↪ _) -> k' /= a) s
13          Nothing         -> (a, (v, 0)) : s
14     in ((), newS))
```

Tracking Memoization Efficiency The memory system also includes instrumentation utilities to monitor reuse statistics:

- `resetCounters :: State m ()` — resets all lookup counters.
- `showCounters :: State m [(k, Int)]` — returns keys with reuse counts.
- `sumCounters :: State m Int` — returns total number of reuses.

These functions are useful for assessing the benefits of memoization during performance benchmarks, as discussed in Section 5.

4.4.4 Built-in Functions and List Processing

The interpreter includes support for several built-in functions for list manipulation and other utility operations. These include:

- `head, tail, isNil, length`
- `sortList, isMember, union, intersection, difference`
- `isEqual, lt, isPair, fst, snd`

Each of these functions is handled as a pattern in the *Call* case of the *eval* function and delegates execution to a helper such as $applyUnion$, $applyLength$, $applyIsMember$, or $applySortList$. These helper functions deal with variability, combining satisfiables presence conditions.

4.5 Static Analysis as Programs

In this work, static analysis is not implemented as a separate compiler pass or a dedicated algorithm, but rather as a program itself, written in PCF+ and executed by the interpreter. This design offers several advantages: it promotes modularity, reusability, and extensibility, allowing analyses to be easily modified, composed, and applied to variational inputs without requiring changes to the interpreter. Furthermore, this approach greatly facilitates experimentation and preliminary evaluation, as different analyses or analysis strategies can be quickly prototyped, tested, and compared simply by changing the analysis program, rather than modifying the underlying execution engine.

Each analysis receives a representation of the program, encoded as variational data structures using `VarValor`, and computes results such as sets of definitions, expressions, or variables.

4.5.1 Data Flow Analyses

The following Data Flow Analyses (DFA) were implemented and expressed as standalone PCF+ programs, leveraging the modular and reusable nature of the interpreter-based approach described earlier. These analyses were already introduced and formally defined in Section 2.4, and are briefly revisited here for completeness:

- **Reaching Definitions**: Determines which variable definitions may reach a given program point.
- **Live Variables**: Identifies variables that are needed in the future before being redefined.
- **Available Expressions**: Tracks expressions that have been computed and not subsequently invalidated.
- **Very Busy Expressions**: Determines which expressions are guaranteed to be used before any possible redefinition.

These analyses follow the classic fixed-point approach based on chaotic iteration over CFG nodes. All are evaluated using the PCF+ interpreter, which supports both variability-aware and memoized execution.

4.5.2 Analysis Infrastructure and Key Functions

All analyses share a common structure built upon core higher-order components implemented in PCF+, following the foundational design presented in the book Principles of Program Analysis [6].

`chaoticIterations:` This is the central fixed-point iterator used to evaluate the analysis over the control flow graph. The program to be analyzed (as VarValor) and returns a pair of entry and exit (mappings of label and facts generated by each analysis).

It applies transfer functions to each node in the CFG until convergence is reached (i.e., no changes in the mapping of facts). Internally, it re-evaluates the mapping on each iteration and uses union or comparison operators to detect stabilization.

`updateMappings:` The `updateMappings` function is responsible for updating the current mapping with new abstract facts generated by the transfer function at each CFG node.

This function applies the changes calculated at a node to the overall mapping. In some analyses (e.g., Reaching Definitions), it performs a union of the new facts with the existing ones. In others (e.g., Available Expressions, Very Busy Expressions), it may use intersection or complete replacement based on analysis semantics.

`flow:` The flow relation determines the successor or predecessor nodes for each node in the CFG, depending on the direction of analysis:

- Forward analyses (e.g., Reaching Definitions, Available Expressions) use successors.
- Backward analyses (e.g., Live Variables, Very Busy Expressions) use predecessors.

Other auxiliary functions are commonly used across all analyses to manipulate and extract structural information from the control flow graph and its encoding. For instance, the function **`init`** returns the label of the initial node in the CFG, while **`final`** identifies the set of final (exit) nodes. The function **`filterFlow`** filters the flow relation to relevant edges for a given node, and **`labels`** retrieves all node labels present in the graph.

In addition, several helper functions are used to interpret the program's encoding and extract semantic information, such as:

- **`labelFromAsgn`** — retrieves the label associated with an assignment statement.
- **`varFromAsgn`** — extracts the variable defined in an assignment.
- **`checkType`** — identifies the type of a node (e.g., assignment, conditional, expression).
- **`getCondFromIf`** — extracts the condition expression from a conditional statement.

These functions provide an abstraction layer over the low-level encoding of the control flow graph in PCF+, allowing analyses to focus on the logical structure and transfer semantics rather than syntactic details.

4.5.3 Reaching Definitions Analysis

Reaching Definitions is a forward Data Flow Analysis whose objective is to determine, for each program point, which variable assignments (definitions) may reach that point without being subsequently redefined along any execution path. This analysis is crucial in identifying the provenance of variable values and enabling optimizations such as dead code elimination and constant propagation.

In the implementation developed for this work, the Reaching Definitions analysis is expressed in PCF+ using both general-purpose and analysis-specific functions. The key functions used to implement this analysis are described below:

Listing 4.25: rdEntry

```
rdEntry :: String -> Any -> [(String, [(String, String)])] ->
  ↪ [(String, String)]
rdEntry (l, prog, exit) {
    if (isEqual (l, init(prog)))
        then makeSetOfFV(fv(prog))
        else findWithUnion(filterFlow(l, flow(prog)), exit)
}
```

The `rdEntry` function determines the set of definitions that reach the entry of a given `Block` with label l. If l corresponds to the initial label of the program, the analysis computes the free variables of the program. Otherwise, it combines the results of predecessors in the control flow graph by filtering the relevant flow entries and applying a union operation.

Listing 4.26: rdExit

```
rdExit :: String -> Any -> [(String, [(String, String)])] ->
  ↪ [(String, String)]
rdExit (l, prog, entry) {
    rdExitWithBlock(l, prog, entry, head(findBlock(l, prog)))
}
```

The `rdExit` function computes the set of definitions that reach the exit of a `Block` with label l, by invoking `rdExitWithBlock`, which applies the transfer function associated with the corresponding program block.

Listing 4.27: killRD

```
killRD :: Any -> [(String, String)] -> [(String, String)]
killRD (block, assignments) {
```

```
    if (checkType(block, "ASGN"))
        then union([((varFromAsgn(getStmt(block))), "?")],
                filterRD(varFromAsgn(getStmt(block)),
                  ↪ labelFromAsgn(getStmt(block)),
                  ↪ assignments))
        else []
}
```

The `killRD` function identifies the definitions that are invalidated (killed) by a given block. In the case of assignment statements (`"ASGN"`), it filters out all previous definitions of the assigned variable and replaces them with a placeholder indicating redefinition.

Listing 4.28: genRD

```
genRD :: Any -> [(String, String)]
genRD (prog) {
    if (checkType(prog, "ASGN"))
        then [(varFromAsgn(getStmt(prog)), labelFromAsgn(
          ↪ getStmt(prog)))]
        else []
}
```

The `genRD` function extracts the new definitions generated by a program block. For assignment statements, it returns a pair consisting of the assigned variable and the label of the assignment.

Together, these functions implement the classic Data Flow Analysis pattern: the entry set is computed by collecting and propagating definitions from predecessors, and the exit set is calculated by applying the kill and gen sets to the entry set. This modular approach allows the analysis to be applied compositionally over variational programs and reused across different interpreters.

Example. Consider the Variant `SANITIZE` from the 3.3, previously presented in Section 3:

$$[pwd := input]^1;$$
$$[pwd := sanitized\_input]^2;$$
$$[result := pwd]^4$$

Listing 4.29: Variant `SANITIZE`

The results of the Reaching Definitions analysis for this SPL program—considering presence conditions—are shown in Table 4.1.

Table 4.1: Reaching Definitions for Variant `SANITIZE`

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\{(input, ?),\ (pwd, ?),$ $(result, ?),\ (sanitized\_input, ?)\}$	$\{(input, ?),\ (pwd, 1),$ $(result, ?),\ (sanitized\_input, ?)\}$
2	$\{(input, ?),\ (pwd, 1),$ $(result, ?),\ (sanitized\_input, ?)\}$	$\{(input, ?),\ (pwd, 2),$ $(result, ?),\ (sanitized\_input, ?)\}$
4	$\{(input, ?),\ (pwd, 2),$ $(result, ?),\ (sanitized\_input, ?)\}$	$\{(input, ?),\ (pwd, 2),$ $(result, 4),\ (sanitized\_input, ?)\}$

Now, consider the alternative variant ¬`SANITIZE`, in which there is no sanitization:

$$[pwd := input]^1;$$
$$[skip]^3;$$
$$[result := pwd]^4$$

Listing 4.30: Variant ¬ `SANITIZE`

Table 4.2: Reaching Definitions for Variant ¬ `SANITIZE`

ℓ	$RD_{entry}(\ell)$	$RD_{exit}(\ell)$
1	$\{(input, ?),\ (pwd, ?),$ $(result, ?)\}$	$\{(input, ?),\ (pwd, 1),$ $(result, ?)\}$
3	$\{(input, ?),\ (pwd, 1),$ $(result, ?)\}$	$\{(input, ?),\ (pwd, 1),$ $(result, ?)\}$
4	$\{(input, ?),\ (pwd, 1),$ $(result, ?)\}$	$\{(input, ?),\ (pwd, 1),$ $(result, 4)\}$

As shown in Table 4.2, the variable `sanitized_input` does not appear in the results, as it is only present in the `SANITIZE` variant and is absent from this particular configuration.

When the Reaching Definitions analysis is applied to the **entire program with variability as input**, the resulting analysis output is also **variational**. That is, the analysis

result encodes the presence conditions under which each definition is valid. When projecting this variational result onto specific configurations (using the associated presence conditions), we recover the same results as in Tables 4.1 and 4.2.

For instance, the value of $RD_{exit}(4)$ in the variational analysis is represented as follows:

Listing 4.31: RDExit(4)

```
analyze :: CFG -> [(String, Integer)]
VarList[
    VarPair(VarString (Var [("input", ttPC)]), VarString (Var
      ↪  [("?", ttPC)])),
    VarPair(VarString (Var [("pwd", notBDD propSanitize)]),
      ↪ VarString (Var [("1", notBDD propSanitize)])),
    VarPair(VarString (Var [("pwd", propSanitize)]),
      ↪ VarString (Var [("2", propSanitize)])),
    VarPair(VarString (Var [("result", ttPC)]), VarString (
      ↪ Var [("4", ttPC)])),
    VarPair(VarString (Var [("sanitized_input", propSanitize)
      ↪ ]), VarString (Var [("?", propSanitize)])) ]
```

This representation demonstrates how each definition is annotated with a presence condition, preserving both variants within a single unified analysis result.

4.5.4 Other Analyses and Key Differences

The remaining analyses follow the same architectural pattern but differ in their transfer functions, directions, and semantic interpretations:

Live Variables (Backward Analysis)

- Goal: Identify variables that will be used later before being redefined.
- Transfer function propagates used variables backward.
- Kill set: Variables redefined at the node.
- Gen set: Variables used in the expression.

Available Expressions (Forward Analysis)

- Goal: Identify expressions that have been computed and not invalidated.
- Kill set: Expressions involving redefined variables.

- Gen set: Expressions computed at the node.
- Merge operation: Intersection instead of union (expressions must be available along all paths).

Very Busy Expressions (Backward Analysis)

- Goal: Expressions guaranteed to be used on all paths before any redefinition.
- Gen set: Expressions used at the node.
- Kill set: Expressions invalidated by assignments.
- Merge operation: Intersection (similar to available expressions but in reverse direction).

4.5.5 Correctness and Testing Strategy

Unlike previous work [14], this work does not include a formal verification of the correctness of the proposed memoization strategy, nor of its interaction with the variational lifting mechanism. The absence of formal guarantees means that, at this stage, it is not possible to rigorously assert that the memoization technique is correct or that it always behaves properly when combined with lifted analyses.

Nevertheless, a comprehensive suite of tests was developed to build practical confidence in the implementation. These tests include:

- Step-by-step validation of the intermediate and final results produced by each data-flow analysis (e.g., sets such as `init`, `final`, `kill`, `gen`, etc.), ensuring they match the expected outcomes for each example.
- Unit tests of auxiliary functions used by the interpreter, including presence propagation (`apply`, `union`, `substitute`), Boolean operations on BDDs, and the representation of variational values.
- Cross-validation between interpreter configurations: results produced by memoized and non-memoized versions were systematically compared to ensure consistency.

In addition, the memory system is built on a purely functional programming model, where function outputs depend solely on their inputs and produce no side effects. The memoization key is computed as a hash of the function's name and its arguments. Consequently, when any part of the analyzed program changes, the input to the affected

function changes as well, yielding a different hash. As a result, the interpreter naturally bypasses outdated cached results without requiring explicit cache invalidation.

While this empirical testing provides a strong basis for confidence in the technique, a formal treatment of correctness—especially to ensure sound reuse of memoized results in variational contexts—is left as future work.

Chapter 5

Empirical Evaluation

This chapter presents an empirical evaluation of the proposed method, reporting the observed results and discussions. Section 5.1 defines the scope of the evaluation, while Section 5.2 details the experiment setup. The results are presented in Section 5.3, followed by an analysis and discussion in Section 5.4. Finally, potential threats to validity are discussed in Section 5.5.

5.1 Definition

To define the scope of the evaluation, we use the Goal-Question-Metric (GQM) methodology [21]. The primary goal is to assess the impact of variability-aware execution and memoization in static analysis.

Table 5.1 summarized the evaluation goal.

Table 5.1: GQM Goal

Purpose	Assess
Issue	Performance
Object	Memoized and Variability-aware control-flow-based static analysis
Viewpoint	Quality Assurance
Context	Evolving SPL

The following methods and questions are presented to help us achieve the goal.

- **Q1:** How does the use of memoization and variability-aware techniques in the interpreter affect runtime?
  - **M1.1:** Execution time per analysis.
- **Q2:** How effective is memoization in reusing previously computed results?

- **M2.1:** Number of entries stored in memory.
- **M2.2:** Number of reused computations.

5.2 Experiment Setup

To conduct this evaluation, we compare the use of the Variational and Memoized Interpreter against the Base Interpreter. Additionally, by including the use of the only Variational Interpreter and the only Memoized Interpreter, we aim to provide a clearer understanding of the overall results.

The empirical evaluation is conducted using a set of benchmark programs analyzed under four interpreters:

1. **Base Interpreter**: Standard execution without variability or memoization.
2. **Variational Interpreter**: Handles variability using presence conditions.
3. **Memoized Interpreter**: Implements caching to avoid redundant computations.
4. **Variational and Memoized Interpreter**: Combines variability handling with memoization.

If the chosen interpreter is one that does not handle variability (Base or Memoized), the Interpreter step is executed multiple times—once per variation of the SPL.

The final analysis result is produced:

- A `VarValor` result if a Variability-aware interpreter is used (Variational or Variational + Memoized).
- A list of `Valor` results if a non-variability-aware interpreter is used (Base or Memoized). Valor is a Data Structure similar to VarValor, without PresenceConditions.

5.2.1 Subject System Selection

Due to the limitations of the Adapted WHILE Language as the target language for programs, it was not feasible to employ real-world Software Product Lines (SPLs) directly. Instead, we designed a controlled set of benchmark programs that aim to reflect a variety of computational patterns and types of variability, inspired by common variability scenarios studied in SPL evolution literature.

This set of programs was developed with the intention of illustrating different forms of variability and to allow a controlled evaluation of the analysis techniques under different conditions. While not based on a specific established benchmark suite, the design of the

subject systems focus on variability aspects observed in real-world SPLs, as discussed by Kröher et al. (2023) [8] and further explored in the analysis of LPS evolution by Hubner [22].

For each benchmark program, an evolved version (denoted as _v2) was also developed. Each _v2 version introduces a different type of change—either structural or functional—to simulate SPL evolution scenarios. These changes vary across programs and aim to emulate some patterns of software evolution, rather than applying a uniform transformation across all benchmarks.

The programs used in our evaluation are described below.

- **Deep Loop Computation** (`deep_loop`)
  Computes a complex arithmetic function iteratively, simulating real-world deep control flow. The loop runs for a large number of iterations, and the presence condition alters the loop depth, affecting execution time. The evolution includes an extra computation.

- **Nested Loop with Variability** (`nested_variability`)
  Implements two nested loops with presence conditions that alter the number of iterations. This program has a computational complexity of $O(n^2)$ and tests how variability impacts nested control flow structures. The evolution increases inner loop bound.

- **Interprocedural Simulation** (`interprocedural_sim`)
  Simulates function calls using While Language semantics. It processes values using an accumulator variable, with variability affecting function logic and computational paths. The evolution adds new conditional branch.

- **Recursion Simulation (Factorial)** (`factorial_rec_sim`)
  Simulates recursion using loops instead of function calls, computing the factorial of a number. Variability determines whether additional computations, such as power calculations, are performed. The evolution extends iteration with power computation.

- **Arithmetic-Heavy Computation** (`arithmetic_heavy`)
  A program with intensive arithmetic operations, including multiplication, division, addition, and subtraction. Presence conditions influence whether additional computations are executed, affecting execution time. The evolution adds another arithmetic operation.

- **Variational Initialization** (`init_variability`)
  A simple program with three presence conditions affecting initial variable assignments. It evaluates how combinations of initial configurations impact the program behavior. The evolution adds a new computation based on previous initialization results.

- **Loop with Multiple Variants** (`loop_multi_variant`)
  A loop whose body is gradually modified by a cascade of presence conditions. This example tests how variability can impact loop behavior incrementally. The evolution includes a post-loop computation based on the accumulated result.

- **Deeply Nested Variants** (`deep_nested_variants`)
  A synthetic example built to stress-test nested `Variant` structures with five presence conditions. It shows how complex variability expressions can control assignment paths. The evolution introduces an additional computation derived from the final assigned value.

5.2.2 Experiment Design and Analysis Procedures

For each program and its revised version, the following Data Flow Analyses (DFAs) were employed in the empirical evaluation. In each case, a set of relevant functions was memoized to explore reuse opportunities during program evolution:

- **Reaching Definitions**: labels, flow, fv, assignments, init, final, findBlock, makeSetOfFV, killRD, genRD, filterFlow.

- **Live Variables**: getVarFromAexp, getVarFromBexp, labels, flowR, flow, fv, init, final, findBlock, killLV, genLV, filterFlow.

- **Available Expressions**: nonTrivialExpression, labels, flow, fv, init, final, findBlock, killAE, genAE, filterFlow.

- **Very Busy Expressions**: labels, flow, flowR, fv, init, final, findBlock, killVB, genVB, filterFlow.

The selection of functions for memoization was guided by their computational specificity and likelihood of being reused across program versions. These functions typically operate over program structure (e.g., control flow graph traversal or syntactic analysis), produce intermediate data reused in multiple steps of the analysis pipeline, and are agnostic to the overall program context. For example, functions like `flow`, `findBlock`, and `fv` are invoked repeatedly in the fixpoint computations of multiple DFAs and exhibit stable

input-output behavior when unaffected parts of the program remain unchanged. Memoizing these components offers high reuse potential with minimal risk of stale data, as any structural or semantic change in their inputs will automatically produce a different hash and bypass the cache. Additionally, memoized functions are often small in number but dominate execution time due to their repeated invocation, making them ideal targets for memoization from a cost-benefit perspective.

Each analysis was executed using all four interpreter configurations: Base, Variational, Memoized, and Variational and Memoized. The benchmark programs were used as inputs, and each program was evaluated in both its original and evolved version (`v1` and `v2`).

For interpreters supporting memoization, a serialized memory state was produced after analyzing `v1`. This memory file was subsequently used to initialize the analysis of the corresponding `v2` program, allowing reuse of prior computations and simulating incremental analysis in practice.

We used the Haskell library *criterion*[1] to benchmark the time spent computing analyses with each interpreter configuration. Criterion performs statistically robust benchmarking by executing each benchmark multiple times (often dozens or hundreds of iterations), discarding warm-up phases, and applying techniques such as bootstrapping and linear regression modeling to estimate mean runtime, standard deviation, and confidence intervals. It also detects and classifies outliers to improve result accuracy.

All benchmarks were automatically analyzed using these statistical methods, with runtime data exported in JSON format and post-processed to derive aggregated metrics. This procedure ensures reliable and reproducible measurements while minimizing variance introduced by system-level fluctuations such as CPU scheduling or background processes.

This benchmarking process was specifically designed to collect high-confidence runtime metrics for addressing our first research question: ***Q1*** – *Compared to the baseline, how much faster are memoized lifted analyses?*

From these benchmarks, we extracted the average execution time and standard deviation per file. To compute aggregate metrics for an analysis, individual means of each program were summed, and the overall standard deviation was calculated assuming independent measurements (i.e., square root of the sum of variances).

The benchmark generates two output files:

- `runtime_metrics.csv`: Contains the execution time statistics per program and analysis, including mean time, standard deviation, and outlier variance.
- `cache_metrics.csv`: Contains cache-related statistics, including serialized memory size (`CacheSize`), cache misses (`CacheMiss`), and cache hits (`CacheHits`). A cache

[1] `https://hackage.haskell.org/package/criterion`

miss is recorded when a memory lookup fails to retrieve a cached result, requiring recomputation. A cache hit occurs when a previously memoized function result is reused.

These metrics provide a systematic basis for evaluating the efficiency of memoization and its impact on both runtime performance and cache reuse during SPL evolution.

5.2.3 Instrumentation and Operation

The experiments were conducted on a 2017 MacBook Pro equipped with 8 GB of RAM, a 3.1 GHz Intel Core i5 dual-core processor, and a 512 GB SSD. Correctness was verified through manual comparison of the textual outputs produced by each analysis across all interpreters. A reproducibility package containing all benchmark programs, input data, and evaluation scripts is available in our source code repository[2].

5.3 Results

Following the evaluation plan outlined in Section 5.2, the results for each analysis and interpreter are presented below, covering runtime performance (Section 5.3.1) and memoization efficiency (Section 5.3.2).

Table 5.2 presents the benchmark programs used in this work along with their corresponding abbreviations (short labels). These labels are used throughout the figures and graphs to improve readability and reduce visual clutter.

Program	**Label**
Arithmetic Heavy	AH
Deep Loop	DL
Deep Nested Variants	DNV
Interprocedural	I
Loop with Multiple Variants	LMV
Nested Loop	NL
Recursion Sim	RS
Variational Initialization	VI

Table 5.2: Short labels for benchmark programs

5.3.1 Analysis Runtime

Figures 5.1–5.4 present the cumulative execution time (in milliseconds) for four data-flow analyses — Reaching Definitions, Live Variables, Available Expressions and Very Busy

[2] `https://github.com/fischertayna/lifting-framework/tree/main/benchmarks`

Expressions — across different interpreter variants: `Base`, `Memoized`, `Variational` and `VMemoized` — Variational and Memoized. Each bar is divided into two stacked segments representing program versions V1 and V2, which correspond to successive variants of the same program and reflect the effects of software evolution. The height of each segment indicates the aggregated mean runtime for that version, while the error margins correspond to the combined standard deviation, assuming independent measurements. These visualizations highlight how each interpreter handles changes across program versions and how optimizations such as memoization and variability awareness impact performance.

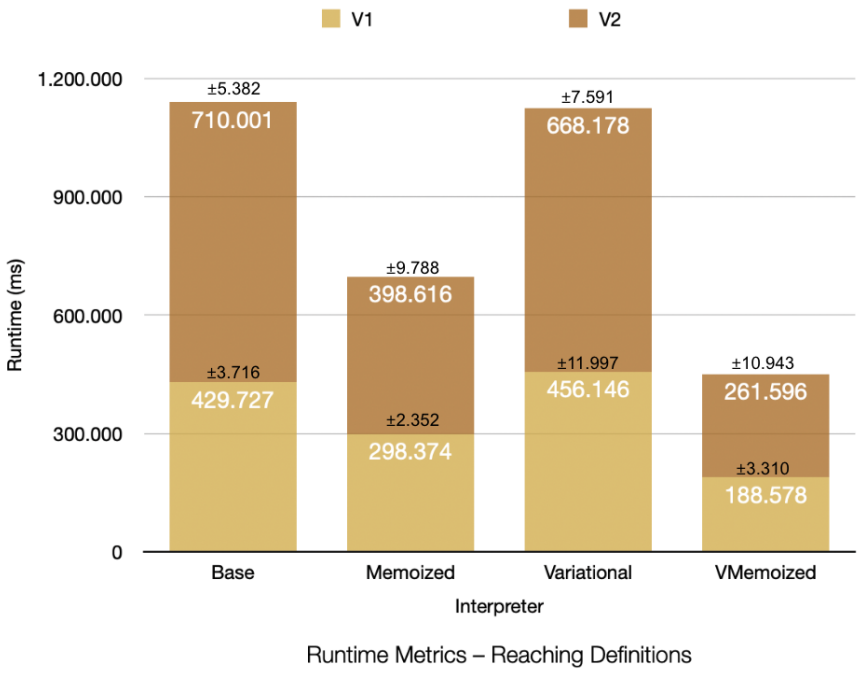

Figure 5.1: Runtime Metrics – Reaching Definitions

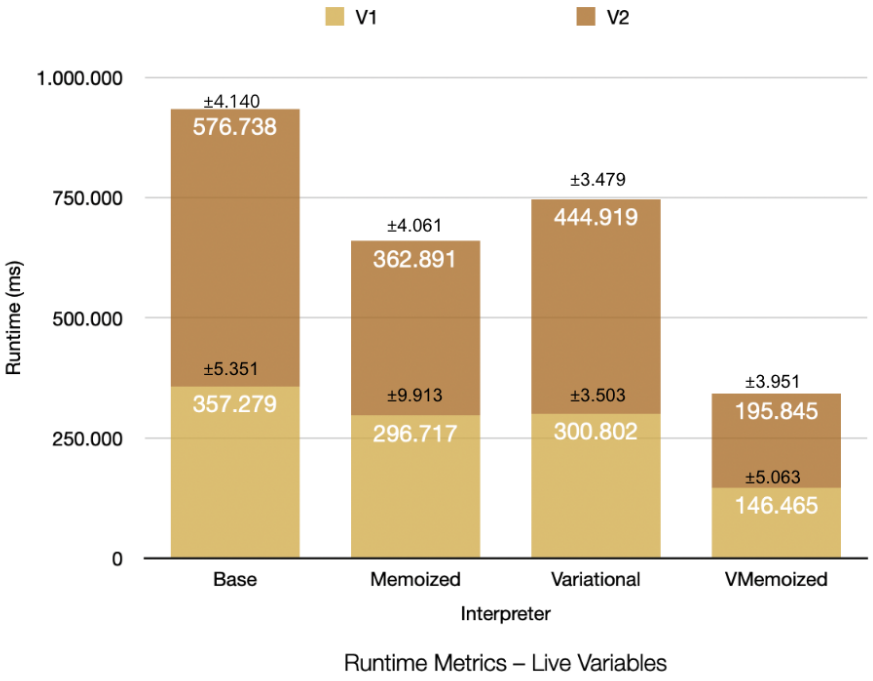

Figure 5.2: Runtime Metrics – Live Variables

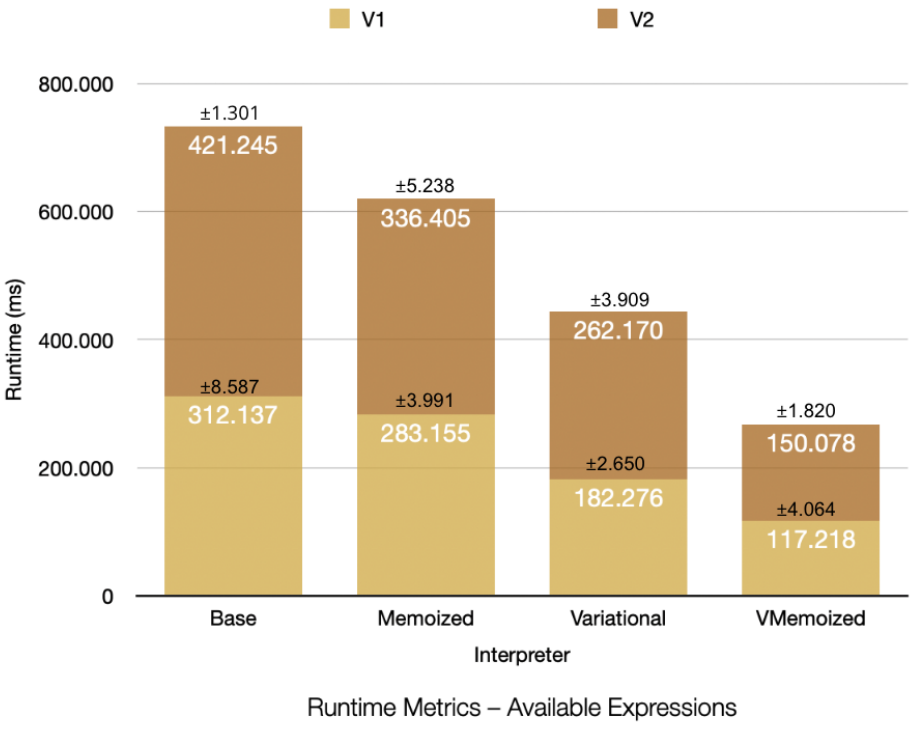

Figure 5.3: Runtime Metrics – Available Expressions

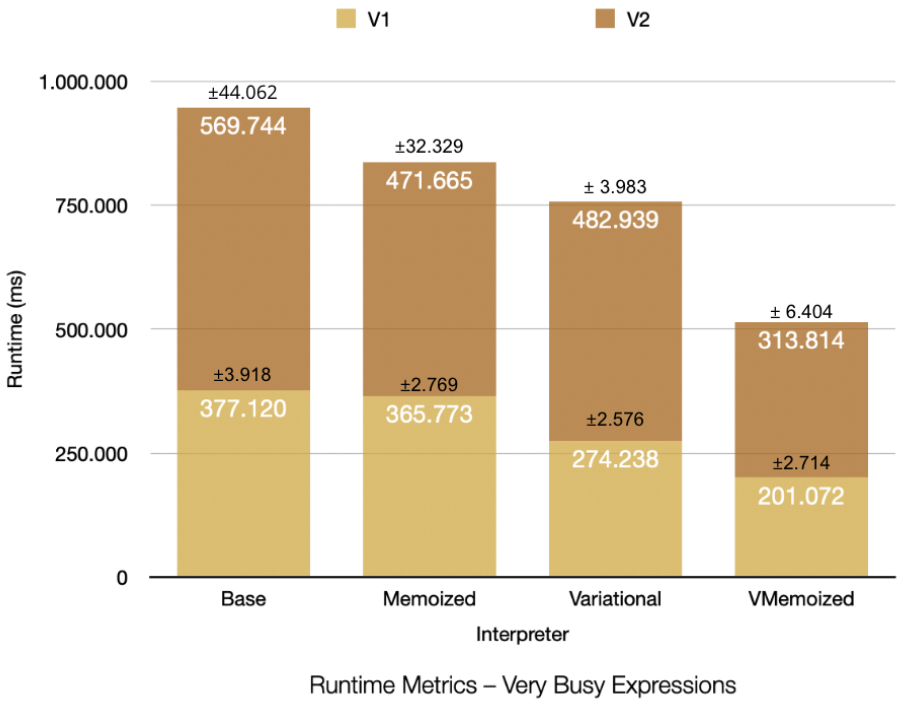

Figure 5.4: Runtime Metrics – Very Busy Expressions

5.3.2 Memoization Efficiency

Tables 5.3-5.6 presents the cache metrics collected across all programs when using Variational and Memoized Interpreter. The metrics include:

- **Cache Size**: The storage size of the file outputed with the Memory State (Bytes).
- **Cache Misses**: The number of unique memory state entries not found in cache.
- **Cache Hits**: The number of reused function results found in cache.

Program	Version	VMemoized Cache Size (B)	Miss	Hits	Memoized Cache Size (B)	Miss	Hits
AH	v1	10463	33	367	12155	53	549
AH	v2	12041	37	533	14280	60	790
DL	v1	7468	25	299	6788	36	197
DL	v2	8968	29	401	8639	42	319
DNV	v1	11435	29	272	6645	43	469
DNV	v2	12228	33	367	12806	68	1372
I	v1	8567	29	229	8627	45	387
I	v2	12163	41	471	13786	66	846
LMV	v1	11023	37	533	16101	77	1363
LMV	v2	12581	41	599	19585	88	2320
NL	v1	9643	33	367	11150	53	506
NL	v2	10695	37	533	12665	60	740
RS	v1	9376	29	401	8790	43	282
RS	v2	10929	33	467	10613	49	468
VI	v1	9521	29	272	22506	111	593
VI	v2	10617	33	367	29804	136	1024

Table 5.3: Memoization efficiency for Reaching Definitions analysis.

Program	Version	VMemoized Cache Size (B)	Miss	Hits	Memoized Cache Size (B)	Miss	Hits
AH	v1	7596	30	175	8359	42	308
AH	v2	8738	34	201	9821	48	567
DL	v1	5677	22	94	4934	28	92
DL	v2	6892	26	254	6491	34	278
DNV	v1	7637	26	44	4155	30	290
DNV	v2	8782	30	93	8643	50	1006
I	v1	6278	25	305	6353	34	236
I	v2	9185	37	473	10193	52	848
LMV	v1	8622	34	436	12504	64	1096
LMV	v2	9883	38	492	15469	74	1851
NL	v1	7520	30	257	8207	42	308
NL	v2	8613	34	295	9654	48	444
RS	v1	6883	26	149	6191	34	226
RS	v2	8168	30	298	7795	40	315
VI	v1	6583	26	324	14690	76	604
VI	v2	7699	30	380	19793	94	826

Table 5.4: Memoization efficiency for Live Variables analysis.

Program	Version	VMemoized Cache Size (B)	Miss	Hits	Memoized Cache Size (B)	Miss	Hits
AH	v1	9592	30	216	10836	47	303
AH	v2	11274	34	295	12695	54	438
DL	v1	6982	22	36	5921	30	50
DL	v2	8569	26	44	7770	37	67
DNV	v1	7332	26	44	3931	30	290
DNV	v2	10732	30	52	10038	55	649
I	v1	7681	26	79	7675	40	76
I	v2	10518	38	121	11719	61	127
LMV	v1	10311	34	60	15677	76	388
LMV	v2	12228	38	68	19682	89	471
NL	v1	8389	30	52	9612	47	93
NL	v2	9440	34	60	11126	54	110
RS	v1	8568	26	44	7441	37	67
RS	v2	10413	30	52	9485	44	84
VI	v1	9187	26	79	19066	94	178
VI	v2	11045	30	134	26357	119	433

Table 5.5: Memoization efficiency for Available Expressions analysis.

Program	Version	VMemoized Cache Size (B)	Miss	Hits	Memoized Cache Size (B)	Miss	Hits
AH	v1	9648	30	93	10873	47	163
AH	v2	11291	34	389	12733	54	479
DL	v1	7473	22	65	6202	30	96
DL	v2	9182	26	79	8158	37	125
DNV	v1	7341	26	44	4046	30	290
DNV	v2	10704	30	380	9730	50	1006
I	v1	7678	25	107	7564	38	70
I	v2	10481	37	167	11610	59	121
LMV	v1	10793	34	436	14460	68	860
LMV	v2	12856	38	492	18171	79	1041
NL	v1	8878	30	52	9881	47	93
NL	v2	9935	34	60	11392	54	110
RS	v1	9098	26	114	7716	37	125
RS	v2	11069	30	134	9872	44	189
VI	v1	9209	26	219	16583	80	328
VI	v2	11048	30	257	22265	99	453

Table 5.6: Memoization efficiency for Very Busy Expressions analysis.

5.4 Analysis and Discussion

This section discusses the results presented in Section 5.3, highlighting the trade-offs between variability-awareness and memoization, as well as the impacts of software product line (SPL) evolution on analysis performance.

5.4.1 Analysis Runtime

The runtime results across the four data-flow analyses — *Reaching Definitions*, *Live Variables*, *Available Expressions*, and *Very Busy Expressions* — reveal distinct performance patterns depending on the type of analysis and the optimization strategy applied. In general, both the **Memoized** and **Variational** interpreters demonstrate competitive and stable performance, while the **vMemoized** interpreter consistently achieves the best or near-best results across all scenarios.

For the *Reaching Definitions* and *Live Variables* analyses, the **impact of memoization** is notably greater than that of variability-aware execution alone. These analyses

involve extensive traversal of control-flow paths — forward in the case of Reaching Definitions and backward for Live Variables — and frequently revisit similar program states across evolving versions. In such cases, memoization enables the reuse of intermediate computations, significantly reducing redundant processing and resulting in substantial performance gains. The **vMemoized** interpreter, which combines memoization with variability-awareness, outperforms other variants by leveraging both cross-version reuse and the compact representation of configuration-specific behavior.

In contrast, the *Available Expressions* and *Very Busy Expressions* analyses exhibit a slightly different pattern. Here, **variability-aware execution** often leads to better performance compared to pure memoization. These analyses benefit from evaluating expressions in a shared context across variants, especially when common expressions appear in multiple configurations. By lifting the analysis over the entire configuration space, the **Variational** and **vMemoized** interpreters can avoid duplicated evaluation of shared subexpressions. Consequently, the Variational interpreter alone already performs competitively, and when memoization is added, **vMemoized** achieves the lowest overall runtimes, particularly in programs with reusable or persistent computations across configurations and versions.

These trends reflect the nature of the **memoized functions** implemented in each analysis. Analyses based on propagating sets (e.g., reaching definitions or live variables) tend to produce similar results across versions and thus benefit more from caching. Conversely, analyses involving the tracking of expressions and their availability across paths and configurations gain more from the sharing mechanisms enabled by variability-awareness. Ultimately, the results suggest that the effectiveness of each optimization technique is analysis-dependent, reinforcing the value of combining both strategies in a unified interpreter.

Memoization and Variability-aware Techniques Performance Impact

The combined use of memoization and variability-aware techniques reduced execution time significantly, especially in evolving scenarios, by avoiding redundant computations across configurations and software versions.

5.4.2 Memoization Efficiency

The cache metrics provide additional evidence of the effectiveness of memoization in reducing redundant computations during static analysis. Overall, the results highlight that memoization supports both *temporal reuse* (across program versions) and *spatial*

reuse (within the same execution), especially in analyses over programs with repetitive structures and control flow complexity.

A notable pattern is that the **Variational and Memoized (vMemoized)** interpreter consistently exhibits **lower cache size and fewer cache misses** than the Memoized interpreter, despite performing the same analyses. This indicates that the integration of variability-aware execution with memoization leads to more compact and efficient caching. By analyzing all variants simultaneously, the interpreter avoids redundant evaluations across configurations, leading to more opportunities to reuse cached results.

Interestingly, a cache miss count in Version 2 that is close to the count from Version 1 does not necessarily suggest a lack of reuse. On the contrary, it often reflects that the interpreter was able to continue using existing entries from the earlier version, and that relatively few new computations were required. This behavior is particularly evident in programs that evolve structurally while retaining a high degree of similarity with earlier versions.

Programs such as `Deep Nested Variants` and `Loop with Multiple Variants`, which feature a higher number of presence conditions and branching points, show particularly high cache reuse. These programs benefit from both the structural regularity and the increased opportunities for reusing previously memoized sub-computations, as many parts of the analysis logic remain unchanged across configurations and versions.

Cache hits are another critical metric. High cache hit rates indicate that the interpreter efficiently avoided re-computation by retrieving results stored in earlier steps of the chaotic iteration. This reuse occurred both within a single program execution (e.g., recursive or repetitive constructs) and across successive versions of the same program. The correlation between high cache hit counts and reduced execution time confirms that memoization not only improves performance but also ensures scalability in the presence of evolving and configurable software.

How effective is memoization in reusing previously computed results

Memoization was highly effective, with high cache hit rates, reduced memory usage, and fewer cache misses in the variational setting, enabling extensive reuse of previously computed results and minimizing redundant evaluations.

5.4.3 Summary of Findings

The **Variational and Memoized Interpreter** (vMemoized) consistently achieves the best trade-off between variability-awareness and execution performance. By combining memoization and variational execution, it enables extensive reuse of previously computed

results while preserving precision across multiple configurations. This dual optimization proves especially effective in the analysis of Software Product Lines (SPLs) with complex control flows and structural similarity across program versions.

Across the four data-flow analyses, **Reaching Definitions** and **Live Variables** exhibited the most significant gains from memoization. These analyses involve extensive traversal of the control-flow graph and generate recurring intermediate results, which are highly amenable to caching. In contrast, **Available Expressions** and **Very Busy Expressions** benefitted more from **variability-aware execution**, due to their reliance on configuration-sensitive expression reuse. This reflects a complementary relationship between the two techniques: while memoization excels in reducing recomputation over time, variability-awareness improves performance across configuration space.

Additionally, cache metrics revealed that the **vMemoized** interpreter achieved lower cache sizes and fewer misses than the Memoized interpreter alone, indicating more efficient caching when presence conditions are integrated into execution. High cache hit rates confirmed effective reuse not only across program versions (temporal reuse), but also within the same analysis execution (spatial reuse). Programs with deeper nesting and a greater number of presence conditions, such as those with multiple variability points, demonstrated particularly strong reuse behavior.

These findings reinforce the value of addressing **both spatial and temporal variability** in static analysis for SPLs. Leveraging both dimensions enables scalable, reusable, and efficient analyses—even as software evolves or expands its configuration space.

5.5 Threats to Validity

This section discusses potential threats to the validity of our empirical evaluation, structured across four categories: internal validity, external validity, construct validity, and conclusion validity.

5.5.1 Internal Validity

Internal validity refers to whether the observed results can be attributed to the factors under study rather than other confounding variables.

- **Implementation Optimizations:** Differences in implementation optimizations across interpreters may unintentionally favor one approach over another. Despite efforts to implement all interpreters consistently, low-level optimizations may have influenced performance outcomes.

- **System-level Factors:** Variability in system resources, such as CPU scheduling, memory management, or background processes, may impact performance measurements, potentially introducing noise in the collected data.

5.5.2 External Validity

External validity concerns the generalizability of the results beyond the experimental context.

- **Limited Benchmark Set:** The set of benchmark programs used in this evaluation may not be representative of all possible variational programs. As a result, the findings may not generalize to other domains or more complex real-world scenarios.
- **Program Complexity:** Many of the evaluated programs are relatively simple and may not expose the full potential benefits of variational execution and memoization. In more complex settings, different performance trade-offs may emerge.

5.5.3 Construct Validity

Construct validity reflects how well the evaluation metrics capture what they are intended to measure.

- **Selection of Memoized Functions:** The choice of which functions are subject to memoization affects the measured storage overhead and memory load times. Different choices could yield different results and affect the conclusions regarding the efficiency of memoization.

5.5.4 Conclusion Validity

Conclusion validity relates to the reliability of the conclusions drawn from the empirical data.

- **Measurement Variability:** Although multiple runs were conducted to minimize measurement bias, inherent variability in execution time or memory usage may still affect the statistical significance of observed performance differences.
- **Use of Criterion Benchmarking Framework:** To improve the reliability of runtime measurements and mitigate measurement variability, we employed the `Criterion` benchmarking library, a widely adopted framework in the Haskell ecosystem. Criterion automatically performs a large number of iterations (typically thousands),

discards initial warm-up runs to avoid cold-start effects, and applies statistical techniques such as bootstrapping and standard deviation analysis to produce robust estimates. It also calculates confidence intervals for each measurement, helping to identify outliers and reduce the impact of transient system noise. These features make Criterion particularly well-suited for precise benchmarking, contributing to the statistical soundness of the results reported in this evaluation.

Chapter 6

Conclusion

This dissertation presented a framework that combines *variational lifting* and *memoization* to support scalable and reusable static analysis in the context of evolving Software Product Lines (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 main contributions of this work are:

- The implementation of control-flow-based static analyses for an adapted While language;
- The development of a variational and memoized interpreter capable of handling both spatial and temporal variability;
- An empirical evaluation demonstrating the performance benefits of combining variational execution and memoization.

The empirical evaluation confirmed the 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 practical scalability of the proposed framework for evolving Software Product Lines.

6.1 Limitations

While the proposed framework demonstrates promising results, several limitations were identified throughout the development and evaluation process:

- **Restricted Expressiveness of the Interpreter Language**: The current interpreter is based on an extended version of PCF+, which, while sufficient for modeling core data-flow analyses, imposes limitations on the expressiveness of programs and analyses that can be encoded. More complex language constructs, such as algebraic data types, type classes, or lazy evaluation (as in Haskell), are not currently supported. This restricts the applicability of the framework to more advanced or real-world analysis scenarios involving higher-level language features.
- **Memoization Granularity**: The memoization strategy is currently applied at the level of whole function calls. Finer-grained caching mechanisms (e.g., per statement or per basic block) could offer better reuse in certain scenarios.
- **Absence of Real-world Case Studies**: Although the benchmarks were designed to simulate realistic evolution scenarios, the evaluation did not include analyses over large, real-world SPLs, which may exhibit different patterns of variability and evolution.
- **Lack of Formal Verification**: This work does not provide a formal proof of correctness for the memoization strategy or its interaction with variational lifting. While extensive empirical testing was conducted to validate the implementation, formal guarantees regarding soundness and safety of reuse remain future work.
- **Tooling and Integration**: The current framework is implemented as a standalone tool. Integration with broader analysis ecosystems or development environments would require additional effort.

Chapter 7

Future Work

Several opportunities exist to expand and improve upon this research:

- **Extend the Interpreter Language**: As a natural evolution of this work, the interpreter could be extended to support more complex languages such as Haskell. This would enable the application of the framework to more expressive analyses and real-world functional programs, promoting reuse and extensibility in advanced static analysis scenarios.
- **Fine-grained Memoization**: Exploring more granular memoization strategies could lead to better reuse and performance, especially in large analyses. One possibility is caching at the level of individual control flow blocks or variable definitions.
- **Analysis of Real-world SPLs**: Applying the framework to real SPLs (e.g., subsystems from the Linux Kernel or coreboot) would allow for a more comprehensive validation of its scalability and practical relevance.
- **Extending Analysis Capabilities**: New forms of static analysis, such as taint analysis, pointer analysis, or type inference, could be implemented within the same interpreter-based infrastructure, benefiting from the variational and memoized execution model.
- **Formal Verification of Correctness**: A promising direction is the formal verification of the correctness of the memoization strategy and its interaction with variational lifting. This would involve defining formal semantics for both mechanisms and proving that memoized results are semantically equivalent to non-memoized computations. Such a foundation would strengthen the reliability of the framework and its applicability in safety-critical domains. The formalization of variational lifting could build upon the existing formal semantics proposed by Shahin and Chechik [12]

or leverage the formal and mechanized semantics developed by Castro et al. [23], providing a solid basis for further theoretical guarantees.

- **Tool Integration and Visualization**: Integrating the framework into existing IDEs or analysis platforms, and developing visualizations for variational analysis results, could increase usability and adoption.

Overall, this dissertation lays the foundation for future research on integrated approaches to handle both dimensions of variability—space and time—in static program analysis. The proposed framework contributes toward building more modular, reusable, and efficient analysis tools for highly configurable and evolving systems.

References

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

[2] SPLC: *Software Product Line Conference - Hall of Fame*. https://splc.net/fame.html, 2024. Accessed: 2024-10-02. 1

[3] Thüm, Thomas, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer: *Towards Efficient Analysis of Variation in Time and Space*. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, pages 57–64, Paris France, September 2019. ACM, ISBN 978-1-4503-6668-7. https://dl.acm.org/doi/10.1145/3307630.3342414. 1, 2, 14

[4] Møller, Anders and Michael I. Schwartzbach: *Static program analysis*, October 2018. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/. 1

[5] Thüm, Thomas, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake: *Analysis strategies for software product lines*. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, April 2012. 1

[6] Nielson, Flemming, Hanne R. Nielson, and Chris Hankin: *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010, ISBN 3642084745. 1, 5, 10, 11, 12, 36

[7] Thüm, Thomas, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake: *A Classification and Survey of Analysis Strategies for Software Product Lines*. ACM Computing Surveys, 47(1):1–45, July 2014, ISSN 0360-0300, 1557-7341. https://dl.acm.org/doi/10.1145/2580950. 2, 13

[8] Kröher, Christian, Lea Gerling, and Klaus Schmid: *Comparing the intensity of variability changes in software product line evolution*. Journal of Systems and Software, 203:111737, 2023, ISSN 0164-1212. https://www.sciencedirect.com/science/article/pii/S0164121223001322. 2, 14, 46

[9] Bodden, Eric, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini: *Spllift: statically analyzing software product lines in minutes instead of*

years. SIGPLAN Not., 48(6):355–364, June 2013, ISSN 0362-1340. https://doi.org/10.1145/2499370.2491976. 2, 14

[10] Walkingshaw, Eric, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden: *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*, pages 213–226, Portland Oregon USA, October 2014. ACM, ISBN 978-1-4503-3210-1. https://dl.acm.org/doi/10.1145/2661136.2661143. 2, 4, 5, 13, 22

[11] Lanna, André, Thiago M. Castro, Vander Alves, Genaína Nunes Rodrigues, Pierre-Yves Schobbens, and Sven Apel: *Feature-family-based reliability analysis of software product lines*. Inf. Softw. Technol., 94:59–81, 2018. https://doi.org/10.1016/j.infsof.2017.10.001. 2

[12] Shahin, Ramy and Marsha Chechik: *Automatic and efficient variability-aware lifting of functional programs*. Proceedings of the ACM on Programming Languages, 4(OOPSLA):1–27, November 2020, ISSN 2475-1421. https://dl.acm.org/doi/10.1145/3428225. 2, 4, 14, 15, 32, 64

[13] Michie, Donald: *"memo" functions and machine learning*. Nature, 218(5136):19–22, 1968, ISSN 1476-4687. https://doi.org/10.1038/218019a0. 2

[14] Wimmer, Simon, Shuwei Hu, and Tobias Nipkow: *Verified Memoization and Dynamic Programming*. In Avigad, Jeremy and Assia Mahboubi (editors): *Interactive Theorem Proving*, volume 10895, pages 579–596. Springer International Publishing, Cham, 2018, ISBN 978-3-319-94820-1 978-3-319-94821-8. http://link.springer.com/10.1007/978-3-319-94821-8_34, visited on 2023-01-04, Series Title: Lecture Notes in Computer Science. 2, 42

[15] Plotkin, G.D.: *Lcf considered as a programming language*. Theoretical Computer Science, 5(3):223–255, 1977, ISSN 0304-3975. https://www.sciencedirect.com/science/article/pii/0304397577900445. 2, 9

[16] Erwig, Martin and Eric Walkingshaw: *The Choice Calculus: A Representation for Software Variation*. ACM Transactions on Software Engineering and Methodology, 21(1):1–27, December 2011, ISSN 1049-331X, 1557-7392. https://dl.acm.org/doi/10.1145/2063239.2063245. 6

[17] Parnas, D.L.: *On the design and development of program families*. IEEE Transactions on Software Engineering, SE-2(1):1–9, 1976. 13

[18] Brabrand, Claus, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba: *Intraprocedural dataflow analysis for software product lines*. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 13–24, New York, NY, USA, 2012. Association for Computing Machinery, ISBN 9781450310925. https://doi.org/10.1145/2162049.2162052. 14

[19] Liebig, Jörg, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer: *Scalable analysis of variable software*. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013, ISBN 978-1-4503-2237-9. https://dl.acm.org/doi/10.1145/2491411.2491437. 14

[20] Midtgaard, Jan, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski: *Systematic derivation of correct variability-aware program analyses*. Sci. Comput. Program., 105(C):145–170, July 2015, ISSN 0167-6423. https://doi.org/10.1016/j.scico.2015.04.005. 14

[21] Basili, Victor R., Gianluigi Caldiera, and H. Dieter Rombach: *The goal question metric approach*. In *The Goal Question Metric Approach*, 1994. https://api.semanticscholar.org/CorpusID:13884048. 44

[22] Hubner, Herval Alexandre Dias: *Análise de evolução de linhas de produtos de software*. Dissertação de mestrado em informática, Universidade de Brasília, Brasília, Brasil, October 2023. https://ppgi.unb.br/images/documentos/Dissertacoes/Herval_Alexandre_Dias_Hubner.pdf, Orientador: Prof. Dr. Vander Ramos Alves. 46

[23] Castro, Thiago M., Leopoldo Teixeira, Vander Alves, Sven Apel, Maxime Cordy, and Rohit Gheyi: *A formal framework of software product line analyses*. ACM Trans. Softw. Eng. Methodol., 30(3):34:1–34:37, 2021. https://doi.org/10.1145/3442389. 65