Finding Suitable Variability Abstractions for Family-Based Analysis *

Aleksandar S. Dimovski adim@itu.dk, Claus Brabrand, and Andrzej Wąsowski

IT University of Copenhagen, Denmark

Abstract. For program families (Software Product Lines), specially designed *variability-aware* static (dataflow) analyses allow analyzing all variants (products) of the family, simultaneously, in a single run without generating any of the variants explicitly. They are also known as *lifted* or *family-based* analyses. The variability-aware analyses may be too costly or even infeasible for families with a large number of variants. In order to make them computationally cheaper, we can apply variability abstractions which aim to tame the combinatorial explosion of the number of variants (configurations) and reduce it to something more tractable. However, the number of possible abstractions is still intractably large to search naively, with most abstractions being too imprecise or too costly.
In this work, we propose a technique to efficiently find suitable variability abstractions from a large family of abstractions for a variability-aware static analysis. The idea is to use a *pre-analysis* to estimate the impact of variability-specific parts of the program family on the analysis's precision. Then we use the pre-analysis results to find out when and where the analysis should turn off or on its variability-awareness. We demonstrate the practicality of this approach on several Java benchmarks.

1 Introduction

Software Product Lines (SPLs) [7] appear in many application areas and for many reasons. They use features to control presence and absence of software functionality in a product family. Different family members, called *variants*, are derived by switching features on and off, while reuse of the common code is maximized. SPLs are commonly seen in development of embedded software (e.g., cars and phones), system level software (e.g., the Linux kernel), etc. While there are many implementation strategies, many popular industrial SPLs are implemented using annotative approaches such as conditional compilation.

One challenge in development of SPLs is their formal analysis and verification [26]. *Variability-aware* (lifted, family-based) dataflow analysis takes as input only the common code base, which encodes all variants of a program family (SPL), and produces precise analysis results corresponding to all variants. Variability-aware analysis can be significantly faster than the naive "brute-force" approach, which generates and analyzes all variants one by one [3]. However, the computational

* Supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE.

cost of the variability-aware analysis still depends on the number of variants, which is in the worst case exponential in the number of features. To speed-up variability-aware analysis, a range of abstractions at the variability level can be introduced [14]. They aim to abstract the configuration space (number of variants) of the given family. Each *variability abstraction* expresses a compromise between precision and speed in the induced abstract variability-aware analysis. Thus, we obtain a range of (abstract) variability-aware analysis *parameterized* by the choice of abstraction we use. The abstractions are chosen from a large family (calculus) that allows abstracting different variability-specific parts (features, variants, and preprocessor `#ifdef` statements) of a family with varying precision. This poses a hard search problem in practice. The number of possible abstractions is intractably large to search naively, with most abstractions being too imprecise or too costly to show the analysis's ultimate goal.

In this paper, we propose an efficient method to address the above search problem. We present a method for performing selective (abstract) variability-aware analysis, which uses variability-awareness only when and where doing so is likely to improve the analysis precision. The method consists of two phases. The first phase is a *pre-analysis* which aims only to estimate the impact of variability on the main analysis. Hence, it aggressively abstracts the semantic aspects of the analysis that are not relevant for its ultimate goal. The second phase is the main analysis with selective variability-awareness, i.e. the abstract variability-aware analysis, which uses the results of pre-analysis, selects influential features and variants for precision, and selectively applies variability-awareness only to those features and variants. The pre-analysis represents an over-approximation of the main analysis. However, it uses very simple abstract domain and transfer functions, so it can be efficiently run even with full variability-awareness. The pre-analysis and the resulting abstract variability-aware main analysis are different: the pre-analysis is more precise in terms of variability-awareness, but it is worse in tracking non-variability specific parts (i.e. language specific parts that operate on the program state) of the program family. We aim to use the pre-analysis results in order to construct an abstraction which is effective at slicing away (discarding) variability-specific program details (features and variants) that are irrelevant for showing the analysis's goal. The experiments show that the constructed abstract variability-aware analysis achieves competitive cost-precision tradeoffs when applied to Java SPL benchmarks.

In this work, we make the following contributions: (1) We show how to design and use a pre-analysis that estimates the impact of variability on a client (main) analysis; (2) We present a method for constructing a suitable abstract variability-aware analysis that receives guidance from the pre-analysis; (3) We experimentally show the effectiveness of our method using Java program families.

2 Motivating Example

We illustrate our approach using the interval analysis and the program family P:

```
1 x := 0;                          3 #if (B) y := y+2 #endif;
2 #if (A) x := x+2 #endif;         4 #if (¬A) x := x-2 #endif
```

The set of (Boolean) features in the above program family P is $\mathbb{F} = \{A, B\}$, and we assume the set of valid configurations is $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. Note that the variable `y` is (deliberately) uninitialized in P. For each configuration a different variant (single program) can be generated by appropriately resolving `#if` statements. For example, the variant corresponding to the configuration $A \wedge B$ will have both features A and B enabled (set to true), thus yielding the single program: `x := 0; x := x+2; y := y+2`. The variant for $\neg A \wedge \neg B$ is: `x := 0; x := x-2`. The *interval analysis* computes for every variable a lower and an upper bound for its possible values at each program point. The basic properties are of the form: $[l, h]$, where $l \in \mathbb{Z} \cup \{-\infty\}$, $h \in \mathbb{Z} \cup \{+\infty\}$, and $l \leq h$. The coarsest property is $\top = [-\infty, +\infty]$. We want to check the following two *queries* on P: "find all configurations for which `x` and `y` are non-negative at the end of P, and determine *accurately* the corresponding intervals".

Full variability-aware analysis. Full variability-aware (lifted) analysis operates on lifted stores, $\overline{a}$, that contain one component for every valid configuration from $\mathbb{K}$. For the "`#if` (θ) s" statement, lifted analysis checks for each configuration $k \in \mathbb{K}$ whether the feature constraint θ is satisfied by k and, if so, it updates the corresponding component of the lifted store by the effect of analyzing s. Otherwise, the corresponding component of the lifted store is not updated. We assume that the initial lifted store consists of uninitialized `x` and `y`, i.e. they have the initial property $\top$. We use a convention here that the first component of the lifted store corresponds to configuration $A \wedge B$, the second to $A \wedge \neg B$, the third to $\neg A \wedge B$, and the fourth to $\neg A \wedge \neg B$. We write $\overline{a} \overset{\texttt{stm n}}{\longmapsto} \overline{a'}$ when the lifted store $\overline{a'}$ is the result of analyzing the statement "n" at the input lifted store $\overline{a}$.

$$\big([\texttt{x} \mapsto \top, \texttt{y} \mapsto \top], [\texttt{x} \mapsto \top, \texttt{y} \mapsto \top], [\texttt{x} \mapsto \top, \texttt{y} \mapsto \top], [\texttt{x} \mapsto \top, \texttt{y} \mapsto \top]\big)$$

$$\overset{\text{stm } 1}{\mapsto} ([\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top])$$

$$\overset{\text{stm } 2}{\mapsto} ([\texttt{x} \mapsto [2,2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [2,2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top])$$

$$\overset{\text{stm } 3}{\mapsto} ([\texttt{x} \mapsto [2,2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [2,2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [0,0], \texttt{y} \mapsto \top])$$

$$\overset{\text{stm } 4}{\mapsto} ([\texttt{x} \mapsto [2,2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [2,2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [\texttt{-}2,\texttt{-}2], \texttt{y} \mapsto \top], [\texttt{x} \mapsto [\texttt{-}2,\texttt{-}2], \texttt{y} \mapsto \top])$$

As the result of analysis, we can deduce that at the end of P, `x` is non-negative (the exact interval is $[2, 2]$) for configurations that satisfy A (that is, $A \wedge B$ and $A \wedge \neg B$), whereas `x` is negative for configurations that satisfy $\neg A$ (that is, $\neg A \wedge B$ and $\neg A \wedge \neg B$). But, `y` is always $\top$ so we cannot prove any query for it.

Need for abstraction. However, using full variability-aware analysis is not always the best solution. It is often too expensive to run such an analysis with large number of configurations. More importantly, in many cases, full variability-awareness does not help, i.e. either it does not improve some analysis results or the full precision is not useful for establishing some facts. For example, full variability-awareness is not helpful to establish the interval of `y`. Also, we can ignore variants that satisfy $\neg A$ (the last two components) if we only want to

establish the exact interval when x is non-negative. Moreover, we can see that analyzing the feature B is unnecessary for establishing the interval of x.

A family of abstractions. We consider a range of variability abstractions [14] which aim to reduce the size of configuration space. In effect, we obtain computationally cheaper but less precise abstract variability-aware analyses. The three basic abstractions are: (1) to confound (join) all valid variants into one single program with over-approximated control-flow, denoted $\boldsymbol{\alpha}^{\text{join}}$; (2) to project (divide-and-conquer) the configuration space onto a certain subset of variants that satisfy some constraint ϕ, denoted $\boldsymbol{\alpha}^{\text{proj}}_{\phi}$; (3) to ignore a feature, $A \in \mathbb{F}$, deemed as not relevant for the current problem, denoted $\boldsymbol{\alpha}^{\text{fignore}}_{A}$. We also use sequential composition, denoted $\circ$, and product, denoted $\otimes$. Any abstraction α induces an abstract variability-aware analysis, denoted $\overline{\mathcal{A}}_{\alpha}$, which is derived in [14]. Since variability abstractions affect only the variability-specific aspect of the variability-aware analysis (i.e. the transfer function of `#if` statement), it was shown in [14] that they can be also defined as source-to-source transformations. More specifically, for each program family P and abstraction α, we can define an abstract program $\alpha(P)$ such that $\overline{\mathcal{A}}_{\alpha}[\![P]\!] = \overline{\mathcal{A}}[\![\alpha(P)]\!]$, where $\overline{\mathcal{A}}$ represents (unabstracted) variability-aware analysis.

The coarsest abstraction. If we apply the coarsest abstraction $\boldsymbol{\alpha}^{\text{join}}$, which confounds control-flow of all valid configurations into a single program with over-approximated control-flow, we will obtain the following program $\boldsymbol{\alpha}^{\text{join}}(P)$:

```
1 x = 0;                                  3 if (*) then y:=y+2 else skip;
2 if (*) then x:=x+2 else skip;           4 if (*) then x:=x-2 else skip
```

where $*$ models an arbitrary integer. Note that $\boldsymbol{\alpha}^{\text{join}}(P)$ is a single program with no variability in it. When $\boldsymbol{\alpha}^{\text{join}}(P)$ is analyzed using the standard (single-program) interval analysis we obtain the same analysis results as analyzing P with abstract lifted analysis $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{join}}}$. As result of the above analysis, at the end of P we obtain the output store: $\big([\mathtt{x} \mapsto [-2, +2], \mathtt{y} \mapsto \top]\big)$. These estimations are not strong enough to show any of our queries for x and y.

Finding suitable abstractions. The abstract variability-aware analysis aims at analyzing families with only needed variability-awareness. It takes into account only those features and configurations that are likely to improve the precision of the analysis. For the family P, our method should predict that increasing variability-awareness is likely to help answer the first query about the non-negative interval of x, but the second query about the non-negative interval of y will not benefit. Next, our method should find out that we can bring the full benefit of variability-awareness for the first query by taking into account only variants that satisfy A. This abstraction is denoted $\boldsymbol{\alpha}^{\text{proj}}_{(A \wedge B) \vee (A \wedge \neg B)}$, or $\boldsymbol{\alpha}^{\text{proj}}_{A}$ for short. Also, the feature B does not influence the final value of x so we can ignore it obtaining the abstraction $\boldsymbol{\alpha}^{\text{fignore}}_{B} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}$. The abstract program $\boldsymbol{\alpha}^{\text{fignore}}_{B} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(P)$ is:

```
1 x := 0;                                 3 if (*) then y := y+2 else skip;
2 x := x+2;                               4 skip
```

The single-program interval analysis of the above program produces the store: $\big([\mathtt{x} \mapsto [2,2], \mathtt{y} \mapsto \top]\big)$. In this way, we can successfully prove that the first query holds for all configurations that satisfy A since the analysis always analyzes the statement "2", and skips the statement "4".

Pre-analysis. The key idea is to use a pre-analysis and estimate the impact of variability on the most precise main analysis. The pre-analysis uses a simple abstract domain and simple transfer functions, and can be run efficiently even with full variability-awareness. For example, we approximate the interval analysis using a pre-analysis with the abstract domain: $Var \to \{\bigstar, \top\}$, where $\bigstar$ means a non-negative interval, i.e. $[0, +\infty]$. This simple abstract domain of the pre-analysis is chosen because we are interested in showing queries that some variables are non-negative. We run this pre-analysis under full variability-awareness for P:

$$
\begin{aligned}
&\big([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]\big) \\
&\overset{\text{stm 1}}{\longmapsto} \big([\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top]\big) \\
&\overset{\text{stm 2}}{\longmapsto} \big([\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top]\big) \\
&\overset{\text{stm 3}}{\longmapsto} \big([\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top]\big) \\
&\overset{\text{stm 4}}{\longmapsto} \big([\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]\big)
\end{aligned}
$$

The pre-analysis in this case precisely estimates the impact of variability: it identifies where the interval analysis accurately tracks the possible (non-negative) values of `x`. In general, our pre-analysis might lose precision and use $\top$ more often than in the ideal case. However, it does so only in a sound manner.

Constructing an abstraction out of pre-analysis. From the pre-analysis results, we can select those features and configurations that help improve precision regarding given queries. We first identify queries whose variables are assigned with $\bigstar$ in the pre-analysis run. Then, for each query that is judged promising, we find variability-specific parts of the program family that contribute to the query. In our example, pre-analysis assigns $\bigstar$ to `x` in two valid configurations, $A \wedge B$ and $A \wedge \neg B$, which is a good indication that fully variability-aware interval analysis is likely to answer the first query accurately. We keep precision with respect to these two configurations by calculating the abstraction $\boldsymbol{\alpha}^{\text{proj}}_{(A \wedge B) \vee (A \wedge \neg B)}$. We can also see that the feature B does not affect the possible values of `x` at all. Thus, we can ignore the feature B obtaining $\boldsymbol{\alpha}^{\text{fignore}}_{B} \circ \boldsymbol{\alpha}^{\text{proj}}_{(A \wedge B) \vee (A \wedge \neg B)}$. For the second query that `y` is non-negative, we obtain that `y` is $\top$ for all configurations. This is indication that we cannot prove this query even with full variability-aware analysis. Our method guarantees that if the pre-analysis calculates $\bigstar$ for a variable, then the constructed abstract variability-aware analysis will compute an accurate non-negative interval for that variable. However, it is possible that the pre-analysis returns $\top$ for a query due to its own over-approximation, and not because the main analysis cannot prove the query. In this case, our approach will miss the possibility to use variability-awareness to improve the analysis precision.

3 A Language for Program Families

A finite set of Boolean variables $\mathbb{F} = \{A_1, \ldots, A_n\}$ describes the set of available *features* in the family. Each feature may be *enabled* or *disabled* in a particular variant. A *configuration* k is a truth assignment or a valuation which gives a truth value to each feature, i.e. k is a mapping from $\mathbb{F}$ to $\{\text{true}, \text{false}\}$. If a feature $A \in \mathbb{F}$ is enabled for the configuration k then $k(A) = \text{true}$, otherwise $k(A) = \text{false}$. Any configuration k can also be encoded as a conjunction of literals: $k(A_1) \cdot A_1 \wedge \cdots \wedge k(A_n) \cdot A_n$, where $\text{true} \cdot A = A$ and $\text{false} \cdot A = \neg A$. We write $\mathbb{K}$ for the set of all *valid* configurations defined over $\mathbb{F}$ for a family. Note that $|\mathbb{K}| \leq 2^{|\mathbb{F}|}$, since in general not every combination of features yields a valid configuration. We define *feature expressions*, denoted *FeatExp*, as the set of well-formed propositional logic formulas over $\mathbb{F}$ generated using the grammar: $\phi ::= \text{true} \mid A \in \mathbb{F} \mid \neg\phi \mid \phi_1 \wedge \phi_2$.

We use the language $\overline{\text{IMP}}$ for writing program families. $\overline{\text{IMP}}$ is an extension of the imperative language IMP [22] often used in semantic studies. $\overline{\text{IMP}}$ adds a compile-time conditional statement for encoding multiple variants of a program. The new statement "`#if` (θ) s" contains a feature expression $\theta \in \mathit{FeatExp}$ as a presence condition, such that only if θ is satisfied by a configuration $k \in \mathbb{K}$ then the statement s will be included in the variant corresponding to k. The syntax is:

$$s ::= \texttt{skip} \mid \texttt{x:=}e \mid s; s \mid \texttt{if}\,(e)\,\texttt{then}\,s\,\texttt{else}\,s \mid \texttt{while}\,(e)\,\texttt{do}\,s \mid \texttt{\#if}\,(\theta)\,s, \quad e ::= n \mid \texttt{x} \mid e \oplus e$$

where n ranges over integers, `x` ranges over variable names *Var*, and $\oplus$ over binary arithmetic operators. The set of all generated statements s is denoted by *Stm*, whereas the set of all expressions e is denoted by *Exp*. Notice that $\overline{\text{IMP}}$ is only used for presentation purposes as a well established minimal language. The introduced methodology is not limited to $\overline{\text{IMP}}$ or its features.

The semantics of $\overline{\text{IMP}}$ has two stages: first, given a configuration $k \in \mathbb{K}$ compute an IMP single program without `#if`-s; second, evaluate the obtained variant using the standard IMP semantics [22]. The first stage is a simple *preprocessor* which takes as input an $\overline{\text{IMP}}$ program and a configuration $k \in \mathbb{K}$, and outputs a variant corresponding to k. The preprocessor copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP, and recursively pre-processes all sub-statements of compound statements. The interesting case is the "`#if` (θ) s" statement, where the statement s is included in the resulting variant iff $k \models \theta$ (means: k entails θ), otherwise the statement s is removed.

4 Parametric (Abstract) Variability-Aware Analysis

Variability-aware (lifted) analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. In this section, we first briefly explain the process of "lifting" introduced in [21]. Then, we recall the calculus of variability abstractions defined in [14] for reducing the configuration space. Finally, we present the induced abstract variability-aware (lifted) analysis [14], whose transfer functions are parametric in the choice of abstraction.

Lifting Single-program Analysis. Suppose that we have a monotone dataflow analysis for IMP phrased in the abstract interpretation framework [9, 22]. Such an analysis is specified by the following data. A complete lattice $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$ for describing the *properties* of the analysis. A domain $\mathbb{A} = \mathit{Var} \to \mathbb{P}$ of abstract stores, ranged over by a, which associates properties from $\mathbb{P}$ to the program variables *Var*. The *analysis domain* is $\langle \mathbb{A}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, which inherits the lattice structure from $\mathbb{P}$ in a point-wise manner. There are also *transfer functions* for expressions $\mathcal{A}'[\![e]\!] : \mathbb{A} \to \mathbb{P}$ and for statements $\mathcal{A}[\![s]\!] : \mathbb{A} \to \mathbb{A}$, which describe the effect of analyzing expressions and statements in an abstract store.

By using variational abstract interpretation [21], we can lift any single-program analysis defined as above to the corresponding *variability-aware (lifted) analysis* for $\overline{\text{IMP}}$, which is specified as follows. Given a set of valid configurations $\mathbb{K}$, the *lifted analysis domain* is $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, which inherits the lattice structure of $\mathbb{A}$ in a configuration-wise manner. Here $\mathbb{A}^{\mathbb{K}}$ is shorthand for the $|\mathbb{K}|$-fold product $\prod_{k \in \mathbb{K}} \mathbb{A}$, and so in the lifted domain there is one separate copy of $\mathbb{A}$ for each configuration of $\mathbb{K}$. For $\overline{a}, \overline{a}' \in \mathbb{A}^{\mathbb{K}}$, the lifted ordering $\dot{\sqsubseteq}$ is defined as: $\overline{a} \mathrel{\dot{\sqsubseteq}} \overline{a}'$ iff $\pi_k(\overline{a}) \sqsubseteq \pi_k(\overline{a}')$ for all $k \in \mathbb{K}$. The projection π_k selects the k^{th} component of a tuple. Similarly, all other elements of the lattice $\mathbb{A}$ are lifted, thus obtaining $\dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top}$. As an example, $\dot{\top} = \prod_{k \in \mathbb{K}} \top = (\top, \ldots, \top)$, where $\top \in \mathbb{A}$.

The lifted transfer function for statements $\overline{\mathcal{A}}[\![s]\!]$ (resp., for expressions $\overline{\mathcal{A}'}[\![e]\!]$) is a function from $\mathbb{A}^{\mathbb{K}}$ to $\mathbb{A}^{\mathbb{K}}$ (resp., from $\mathbb{A}^{\mathbb{K}}$ to $\mathbb{P}^{\mathbb{K}}$). However in practice, using a tuple of $|\mathbb{K}|$ independent simple functions of type $\mathbb{A} \to \mathbb{A}$ (resp., $\mathbb{A} \to \mathbb{P}$) is sufficient, since lifting corresponds to running $|\mathbb{K}|$ independent analyses in parallel. Therefore, the *lifted transfer functions* are given by the functions $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}}$. The k-th component of the above functions defines the analysis corresponding to the configuration $k \in \mathbb{K}$.

Interval analysis. In the following, we will use the interval analysis to demonstrate this method. The interval analysis is based on the property domain $\langle \mathit{Interval}, \sqsubseteq_I \rangle$: $\mathit{Interval} = \{\bot_I\} \cup \{[l, h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$, where $\bot_I$ denotes the empty interval, and $\top_I = [-\infty, +\infty]$. The partial ordering $\sqsubseteq_I$ is: $[l_1, h_1] \sqsubseteq_I [l_2, h_2]$ iff $l_2 \leq l_1 \wedge h_1 \leq h_2$. The partial ordering $\sqsubseteq_I$ induces the definitions for $\sqcup_I$ and $\sqcap_I$. For each arithmetic operator $\oplus$, we have the corresponding $\widehat{\oplus}$ defined on properties from *Interval* [9]:

$$[l_1, h_1] \widehat{\oplus} [l_2, h_2] = [\min_{x \in [l_1, h_1],\, y \in [l_2, h_2]} \{x \oplus y\}, \max_{x \in [l_1, h_1],\, y \in [l_2, h_2]} \{x \oplus y\}] \quad (1)$$

Thus, we have: $[l_1, h_1] \mathbin{\widehat{+}} [l_2, h_2] = [l_1 + l_2, h_1 + h_2]$ and $[l_1, h_1] \mathbin{\widehat{-}} [l_2, h_2] = [l_1 - h_2, h_1 - l_2]$. For example, $[2, 2] \mathbin{\widehat{+}} [1, 2] = [3, 4]$ and $[2, 2] \mathbin{\widehat{-}} [1, 2] = [0, 1]$.

The *single-program transfer function* for constants is: $\mathcal{A}'[\![\mathtt{n}]\!] = \lambda a.\mathit{abst}_{\mathbb{Z}}(\mathtt{n})$, where $a \in \mathbb{A} = \mathit{Var} \to \mathit{Interval}$, and $\mathit{abst}_{\mathbb{Z}} : \mathbb{Z} \to \mathit{Interval}$ is a function for turning values to properties defined as: $\mathit{abst}_{\mathbb{Z}}(n) = [n, n]$. The corresponding *lifted transfer function* becomes $\overline{\mathcal{A}'}[\![\mathtt{n}]\!] = \lambda \overline{a}. \prod_{k \in \mathbb{K}} \mathit{abst}_{\mathbb{Z}}(\mathtt{n})$, where $\overline{a} \in \mathbb{A}^{\mathbb{K}}$. The complete list of definitions is given in Fig. 1, where for full variability-aware analysis the parameter α is instantiated with the identity abstraction $\boldsymbol{\alpha}^{\text{id}}$. Note that for simplicity, here we overload the λ-abstraction notation, so creating a

tuple of functions looks like a function on tuples: we write $\lambda\overline{a}.\prod_{k\in\mathbb{K}} f_k(\pi_k(\overline{a}))$ to mean $\prod_{k\in\mathbb{K}} \lambda a_k.f_k(a_k)$. Similarly, if $\overline{f} : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ and $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, then we write $\overline{f}(\overline{a})$ to mean $\prod_{k\in\mathbb{K}} \pi_k(\overline{f})(\pi_k(\overline{a}))$.

Variability Abstractions. We now introduce abstractions for reducing the lifted analysis domain $\mathbb{A}^{\mathbb{K}}$. The set *Abs* of abstractions is given by [14]:

$$\alpha \ ::= \ \boldsymbol{\alpha}^{\mathrm{id}} \mid \boldsymbol{\alpha}^{\mathrm{join}} \mid \boldsymbol{\alpha}^{\mathrm{proj}}_{\phi} \mid \boldsymbol{\alpha}^{\mathrm{fignore}}_{A} \mid \alpha \circ \alpha \mid \alpha \otimes \alpha$$

where $\phi \in FeatExp$, and $A \in \mathbb{F}$. For each abstraction α, we define the effect of applying α on sets of configurations $\mathbb{K}$, and on domain elements $\overline{a} \in \mathbb{A}^{\mathbb{K}}$. Note that, the set of features is fixed, i.e. we have $\alpha(\mathbb{F}) = \mathbb{F}$ for any α.

The $\boldsymbol{\alpha}^{\mathrm{id}}$ is an identity on $\mathbb{K}$ and $\overline{a} \in \mathbb{A}^{\mathbb{K}}$. So, $\boldsymbol{\alpha}^{\mathrm{id}}(\mathbb{K}) = \mathbb{K}$ and the abstraction and concretization functions: $\boldsymbol{\alpha}^{\mathrm{id}}(\overline{a}) = \overline{a}$, $\boldsymbol{\gamma}^{\mathrm{id}}(\overline{a}) = \overline{a}$, form a Galois connection [1].

The *join* abstraction, $\boldsymbol{\alpha}^{\mathrm{join}}$, gathers (joins) the information about all configurations $k \in \mathbb{K}$ into one (over-approximated) value of $\mathbb{A}$. We have $\boldsymbol{\alpha}^{\mathrm{join}}(\mathbb{K}) = \{\bigvee_{k\in\mathbb{K}} k\}$, i.e. after abstraction we obtain a single valid configuration denoted by the compound formula $\bigvee_{k\in\mathbb{K}} k$. The abstraction and concretization functions between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\{\bigvee_{k\in\mathbb{K}} k\}} \equiv \mathbb{A}^1$, which form a Galois connection [14], are: $\boldsymbol{\alpha}^{\mathrm{join}}(\overline{a}) = \big(\bigsqcup_{k\in\mathbb{K}} \pi_k(\overline{a})\big)$, and $\boldsymbol{\gamma}^{\mathrm{join}}(a) = \prod_{k\in\mathbb{K}} a$.

The *projection* abstraction, $\boldsymbol{\alpha}^{\mathrm{proj}}_{\phi}$, preserves only the values corresponding to configurations from $\mathbb{K}$ that satisfy $\phi \in FeatExp$. The information about configurations violating ϕ is disregarded. We have $\boldsymbol{\alpha}^{\mathrm{proj}}_{\phi}(\mathbb{K}) = \{k \in \mathbb{K} \mid k \models \phi\}$, and the Galois connection [14] between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\{k\in\mathbb{K}|k\models\phi\}}$ is defined as: $\boldsymbol{\alpha}^{\mathrm{proj}}_{\phi}(\overline{a}) = \prod_{k\in\mathbb{K},k\models\phi} \pi_k(\overline{a})$, and $\boldsymbol{\gamma}^{\mathrm{proj}}_{\phi}(\overline{a}') = \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{a}') & \text{if } k \models \phi \\ \top & \text{if } k \not\models \phi \end{cases}$.

The abstraction $\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}$ ignores a single feature $A \in \mathbb{F}$ that is not directly relevant for the current analysis. It merges configurations that only differ with regard to A, and are identical with regard to remaining features, $\mathbb{F}\backslash\{A\}$. Given $\phi \in FeatExp$, we write $\phi\backslash_A$ for a formula obtained by eliminating the feature A from ϕ (see [14] for details). For each formula $k' \equiv k\backslash_A$ where $k \in \mathbb{K}$, there will be one configuration in $\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}(\mathbb{K})$ determined by the formula $\bigvee_{k\in\mathbb{K},k\backslash_A\equiv k'} k$. Therefore, we have $\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}(\mathbb{K}) = \{\bigvee_{k\in\mathbb{K},k\backslash_A\equiv k'} k \mid k' \in \{k\backslash_A \mid k \in \mathbb{K}\}\}$. The Galois connection [14] between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}(\mathbb{K})}$ is defined as: $\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}(\overline{a}) = \prod_{k'\in\boldsymbol{\alpha}^{\mathrm{fignore}}_{A}(\mathbb{K})} \bigsqcup_{k\in\mathbb{K},k\models k'} \pi_k(\overline{a})$, and $\boldsymbol{\gamma}^{\mathrm{fignore}}_{A}(\overline{a}') = \prod_{k\in\mathbb{K}} \pi_{k'}(\overline{a}')$ if $k \models k'$.

We also have two compositional operators: *sequential composition* $\alpha_2 \circ \alpha_1$, which will run two abstractions α_1 and α_2 in sequence; and *product* $\alpha_1 \otimes \alpha_2$, which will run both abstractions α_1 and α_2 in parallel ("side-by-side"). For precise definitions of $\alpha_2 \circ \alpha_1$ and $\alpha_1 \otimes \alpha_2$, the reader is referred to [14]. In the following, we will simply write $(\alpha, \gamma) \in Abs$ for any $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}^{\alpha(\mathbb{K})}, \dot{\sqsubseteq} \rangle$, which is constructed using the operators presented in this section.

[1] $\langle L, \leq_L \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leq_M \rangle$ is a *Galois connection* between lattices L and M iff α and γ are total functions that satisfy: $\alpha(l) \leq_M m \iff l \leq_L \gamma(m)$ for all $l \in L, m \in M$.

Example 1. Consider the lifted interval analysis and $\overline{a} = \big([\mathtt{x} \mapsto [2,2]], [\mathtt{x} \mapsto [2,2]], [\mathtt{x} \mapsto [0,0]], [\mathtt{x} \mapsto [\mathtt{-2},\mathtt{-2}]]\big)$, where $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.

We have $\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{A\wedge\neg B}(\overline{a}) \sqcup \pi_{\neg A\wedge B}(\overline{a}) \sqcup \pi_{\neg A\wedge\neg B}(\overline{a})) = ([\mathtt{x} \mapsto [\mathtt{-2},2]])$. Thus, the state is significantly decreased to only one component, but the abstraction $\boldsymbol{\alpha}^{\text{join}}$ loses precision by saying that `x` can have any value between -2 and 2. Then, we have $\boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}), \pi_{A\wedge\neg B}(\overline{a})) = ([\mathtt{x} \mapsto [2,2]], [\mathtt{x} \mapsto [2,2]])$. Now the state is decreased to two components that satisfy A. Also, $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{A\wedge\neg B}(\overline{a})) = ([\mathtt{x} \mapsto [2,2]])$. We have $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}) = \{(A \wedge B) \vee (\neg A \wedge B) \equiv B, (A \wedge \neg B) \vee (\neg A \wedge \neg B) \equiv \neg B\}$, and so $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{\neg A\wedge B}(\overline{a}), \pi_{A\wedge\neg B}(\overline{a}) \sqcup \pi_{\neg A\wedge\neg B}(\overline{a})) = ([\mathtt{x} \mapsto [0,2]], [\mathtt{x} \mapsto [\mathtt{-2},2]])$. □

Induced Abstract Lifted Analysis. Recall that any analysis phrased in the abstract interpretation framework can be lifted to the corresponding variability-aware analysis [21], which is specified by the domain $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle$, and lifted transfer functions $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}}$. Given a Galois connection $(\alpha, \gamma) \in \mathit{Abs}$, the abstract lifted analyses induced by (α, γ) has been derived algorithmically in [14]. The derivation finds an over-approximation of $\alpha \circ \overline{\mathcal{A}}[\![s]\!] \circ \gamma$ obtaining a new abstract statement transfer function $\overline{\mathcal{A}}_{\alpha}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K})}$. Also, a new abstract expression transfer function $\overline{\mathcal{A}'}_{\alpha}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\alpha(\mathbb{K})}$ is derived, which over-approximates $\alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma$. Note that full variability-aware analysis $\overline{\mathcal{A}'}[\![e]\!]$ and $\overline{\mathcal{A}}[\![s]\!]$ are included as a special case, i.e. they coincide with $\overline{\mathcal{A}'}_{\boldsymbol{\alpha}^{\text{id}}}[\![e]\!]$ and $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{id}}}[\![s]\!]$. The derivation of $\overline{\mathcal{A}'}_{\alpha}[\![e]\!]$ and $\overline{\mathcal{A}}_{\alpha}[\![s]\!]$ is based on the calculational approach to abstract interpretation [8], which advocates simple algebraic manipulation to obtain a *direct expression* for the abstract transfer functions.

The definitions of $\overline{\mathcal{A}}_{\alpha}[\![s]\!]$ and $\overline{\mathcal{A}'}_{\alpha}[\![e]\!]$ are given in Fig. 1. The function $\overline{\mathcal{A}}_{\alpha}[\![s]\!]$ (resp. $\overline{\mathcal{A}'}_{\alpha}[\![e]\!]$) captures the effect of analysing the statement s (resp., expression e) in a lifted store $\overline{a} \in \mathbb{A}^{\alpha(\mathbb{K})}$ by computing an output lifted store $\overline{a'} \in \mathbb{A}^{\alpha(\mathbb{K})}$ (resp, property $\overline{p} \in \mathbb{P}^{\alpha(\mathbb{K})}$). For "`x := e`", the value of `x` is updated in every component of the input lifted store $\overline{a}$ by the value of the expression e evaluated in the corresponding component of $\overline{a}$. The most interesting case is the analysis of "`#if` (θ) s", which checks the relation between each abstract configuration $k' \in \alpha(\mathbb{K})$ and the presence condition θ. Since k' can be any compound formula, not only a valuation formula as in $\mathbb{K}$, there are three possible cases: (1) if $k' \models \theta$, the corresponding component of the input store is updated by the effect of evaluating the statement s; (2) if $k' \models \neg\theta$, the corresponding component of the store is not updated; (3) if $(k' \wedge \theta)$ and $(k' \wedge \neg\theta)$ are both satisfiable, then the component is updated by the least upper bound of its initial value and the effect of s. For example, when $k' = A$, we obtain: the case (1) if $\theta = A$, the case (2) if $\theta = \neg A$, and the case (3) if $\theta = B$. Note that for $\boldsymbol{\alpha}^{\text{id}}$, since all configurations k in $\mathbb{K}$ are valuation formulas (i.e. either $k \models \theta$ or $k \models \neg\theta$), only the first two cases are possible. Note that, only definitions for constants n and binary operators $\oplus$ are analysis-dependent. So our approach is general and applicable to any static dataflow analysis chosen as a client. The monotonicity and the soundness (i.e., $\alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma \dot{\sqsubseteq} \overline{\mathcal{A}'}_{\alpha}[\![e]\!]$ and $\alpha \circ \overline{\mathcal{A}}[\![s]\!] \circ \gamma \dot{\sqsubseteq} \overline{\mathcal{A}}_{\alpha}[\![s]\!]$) of the abstract lifted analysis follows by construction as shown in [14].

$$\overline{\mathcal{A}}_\alpha[\![\texttt{skip}]\!] = \lambda\overline{a}.\,\overline{a}, \qquad \overline{\mathcal{A}}_\alpha[\![\texttt{x := } e]\!] = \lambda\overline{a}.\prod_{k'\in\alpha(\mathbb{K})} \pi_{k'}(\overline{a})[\texttt{x} \mapsto \pi_{k'}(\overline{\mathcal{A}'}_\alpha[\![e]\!]\overline{a})]$$

$$\overline{\mathcal{A}}_\alpha[\![s_0 \texttt{ ; } s_1]\!] = \overline{\mathcal{A}}_\alpha[\![s_1]\!] \circ \overline{\mathcal{A}}_\alpha[\![s_0]\!], \qquad \overline{\mathcal{A}}_\alpha[\![\texttt{while } e \texttt{ do } s]\!] = \mathrm{lfp}\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a} \dot{\sqcup} \overline{\Phi}(\overline{\mathcal{A}}_\alpha[\![s]\!]\,\overline{a})$$

$$\overline{\mathcal{A}}_\alpha[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] = \lambda\overline{a}.\,\overline{\mathcal{A}}_\alpha[\![s_0]\!]\overline{a} \dot{\sqcup} \overline{\mathcal{A}}_\alpha[\![s_1]\!]\overline{a}$$

$$\overline{\mathcal{A}}_\alpha[\![\texttt{\#if}(\theta)s]\!] = \lambda\overline{a}.\prod_{k'\in\alpha(\mathbb{K})} \begin{cases} \pi_{k'}(\overline{\mathcal{A}}_\alpha[\![s]\!]\overline{a}) & \text{if } k' \models \theta \\ \pi_{k'}(\overline{a}) & \text{if } k' \models \neg\theta \\ \pi_{k'}(\overline{a}) \sqcup \pi_{k'}(\overline{\mathcal{A}}_\alpha[\![s]\!]\overline{a}) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \end{cases}$$

$$\overline{\mathcal{A}'}_\alpha[\![n]\!] = \lambda\overline{a}.\prod_{k'\in\alpha(\mathbb{K})} abst_{\mathbb{Z}}(\mathtt{n}), \qquad \overline{\mathcal{A}'}_\alpha[\![\texttt{x}]\!] = \lambda\overline{a}.\prod_{k'\in\alpha(\mathbb{K})} \pi_{k'}(\overline{a})(\texttt{x})$$

$$\overline{\mathcal{A}'}_\alpha[\![e_0 \oplus e_1]\!] = \lambda\overline{a}.\prod_{k'\in\alpha(\mathbb{K})} \pi_{k'}(\overline{\mathcal{A}'}_\alpha[\![e_0]\!]\overline{a}) \mathbin{\widehat{\oplus}} \pi_{k'}(\overline{\mathcal{A}'}_\alpha[\![e_1]\!]\overline{a})$$

Fig. 1: Definitions of $\overline{\mathcal{A}}_\alpha[\![\overline{s}]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K})}$ and $\overline{\mathcal{A}'}_\alpha[\![\overline{e}]\!] : (\mathbb{A} \to \mathbb{P})^{\alpha(\mathbb{K})}$.

5 Pre-analysis for Finding α

Given a program family and a set of queries, we want to find a good abstraction α for a variability-aware (main) analysis defined by: the domain $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle$, where $\mathbb{A} = Var \to \mathbb{P}$, and the transfer functions $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}}$, $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$. In this section, we first present how to design a pre-analysis, then we describe how we can construct an appropriate abstraction α for the main analysis based on the pre-analysis results.

Definition of Pre-Analysis. We replace the property domain $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$ from the main analysis with a suitable abstract property domain $\langle \mathbb{P}^\#, \sqsubseteq_{\mathbb{P}^\#} \rangle$, from which the pre-analysis is induced. The pre-analysis is fully variability-aware and is specified by the following domains: $\langle \mathbb{A}^\# = Var \to \mathbb{P}^\#, \sqsubseteq \rangle$, $\langle \mathbb{A}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$; and transfer functions: $\overline{\mathcal{A}'^\#}[\![e]\!] : (\mathbb{A}^\# \to \mathbb{P}^\#)^{\mathbb{K}}$, $\overline{\mathcal{A}^\#}[\![s]\!] : (\mathbb{A}^\# \to \mathbb{A}^\#)^{\mathbb{K}}$. Any designed pre-analysis should fulfill two conditions: *soundness* and *computational efficiency*.

Soundness. We design the pre-analysis which runs with full variability-awareness but with a simpler abstract domain and simpler abstract transfer functions than those of the main analysis.
First, there should be a pair of abstraction $\widehat{\alpha}^\# : \mathbb{P} \to \mathbb{P}^\#$ and concretization functions $\widehat{\gamma}^\# : \mathbb{P}^\# \to \mathbb{P}$ forming a Galois connection $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle \underset{\widehat{\alpha}^\#}{\overset{\widehat{\gamma}^\#}{\leftrightarrows}} \langle \mathbb{P}^\#, \sqsubseteq_{\mathbb{P}^\#} \rangle$. These functions formalize the fact that an abstract property from $\mathbb{P}^\#$ in the pre-analysis means a set of properties from $\mathbb{P}$ in the main analysis. By point-wise lifting we obtain the Galois connection $\langle \mathbb{A}, \sqsubseteq \rangle \underset{\alpha^\#}{\overset{\gamma^\#}{\leftrightarrows}} \langle \mathbb{A}^\#, \sqsubseteq \rangle$ by taking: $\alpha^\#(a) = \lambda x.\widehat{\alpha}^\#(a(x))$

and $\gamma^{\#}(a^{\#}) = \lambda x.\widehat{\gamma}^{\#}(a^{\#}(x))$. By configuration-wise lifting we obtain the Galois connection $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\overline{\alpha}^{\#}}{\overset{\overline{\gamma}^{\#}}{\leftrightarrows}} \langle \mathbb{A}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$ by: $\overline{\alpha}^{\#}(\overline{a}) = \prod_{k\in\mathbb{K}} \alpha^{\#}(\pi_k(\overline{a}))$ and $\overline{\gamma}^{\#}(\overline{a^{\#}}) = \prod_{k\in\mathbb{K}} \gamma^{\#}(\pi_k(\overline{a^{\#}}))$. Similarly, by configuration-wise lifting we can construct the Galois connection $\langle \mathbb{P}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\overline{\widehat{\alpha}}^{\#}}{\overset{\overline{\widehat{\gamma}}^{\#}}{\leftrightarrows}} \langle \mathbb{P}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$.

Second, the transfer functions $\overline{\mathcal{A}'^{\#}}[\![e]\!]$ and $\overline{\mathcal{A}^{\#}}[\![s]\!]$ of the pre-analysis should be *sound* with respect to those of the variability-aware main analysis: $\overline{\widehat{\alpha}}^{\#} \circ \overline{\mathcal{A}'}[\![e]\!] \circ \overline{\gamma}^{\#} \dot{\sqsubseteq} \overline{\mathcal{A}'^{\#}}[\![e]\!]$, and $\overline{\alpha}^{\#} \circ \overline{\mathcal{A}}[\![s]\!] \circ \overline{\gamma}^{\#} \dot{\sqsubseteq} \overline{\mathcal{A}^{\#}}[\![s]\!]$, for any $e \in Exp, s \in Stm$. In this way, we ensure that pre-analysis over-approximates variability-aware main analysis.

Computational efficiency. We define a query, q, to be of the form: $(s, P, \mathtt{x}) \in Stm \times \mathcal{P}(\mathbb{P}) \times Var$, which represents an assertion that after the statement s the variable $\mathtt{x}$ should always have a property value from the set $P \subseteq \mathbb{P}$. We want to design a pre-analysis, which although estimates computationally expensive main analysis, still remains computable. We achieve *computational efficiency* of the pre-analysis by choosing very simple property domain $\mathbb{P}^{\#}$. Let $\mathbb{P}^{\#} = \{\bigstar, \top_{\mathbb{P}^{\#}}\}$ be a complete lattice with $\bigstar \sqsubset \top_{\mathbb{P}^{\#}}$. Given the query $q = (s, P, \mathtt{x})$, the functions $\widehat{\alpha}^{\#} : \mathbb{P} \to \mathbb{P}^{\#}$ and $\widehat{\gamma}^{\#} : \mathbb{P}^{\#} \to \mathbb{P}$ are defined as:

$$\widehat{\alpha}^{\#}(p) = \begin{cases} \bigstar & \text{if } p \in P \\ \top_{\mathbb{P}^{\#}} & \text{otherwise} \end{cases} \qquad \widehat{\gamma}^{\#}(\bigstar) = \bigsqcup P,\ \widehat{\gamma}^{\#}(\top_{\mathbb{P}^{\#}}) = \top_{\mathbb{P}}$$

The only non-trivial case is $\bigstar$ denoting at least the properties from the set $P \subseteq \mathbb{P}$ that the given query q wants to establish after analyzing some program code. From now on, we omit to write subscripts $\mathbb{P}$ and $\mathbb{P}^{\#}$ in lattice operators whenever they are clear from the context.

The variability-aware pre-analysis with simple property domain (e.g. $\mathbb{P}^{\#} = \{\bigstar, \top\}$) can be computed by an efficient algorithm based on *sharing representation* [3], where sets of configurations with equivalent analysis information are compactly represented as bit vectors or formulae. For example, the pre-analysis with sharing for the variational program P of Section 2 runs as: $\big([\![\text{true}]\!] \mapsto [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]\big) \ldots \overset{\text{stm}\,3}{\longmapsto} \big([\![\text{true}]\!] \mapsto [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top]\big) \overset{\text{stm}\,4}{\longmapsto} \big([\![A]\!] \mapsto [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\![\neg A]\!] \mapsto [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]\big)$, where $[\![\mathit{true}]\!] = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$, $[\![A]\!] = \{A \wedge B, A \wedge \neg B\}$, and $[\![\neg A]\!] = \{\neg A \wedge B, \neg A \wedge \neg B\}$. Since the abstract domain $\mathbb{P}^{\#}$ of our pre-analysis is very small (has only 2 values), the possibilities for sharing (i.e. configurations with equivalent analysis results) are very promising.

Interval pre-analysis. We now design a pre-analysis for the interval analysis example with respect to queries that require non-negative intervals for variables. The pre-analysis aims at predicting which variables get assigned non-negative values when the program family is analyzed by the variability-aware interval analysis. Suppose that $\mathit{Interval}^{\#} = \{\bigstar, \top\}$, where $\bigstar \sqsubseteq \top$. We define $\widehat{\gamma}^{\#}(\bigstar) = [0, +\infty]$, and $\widehat{\gamma}^{\#}(\top) = [-\infty, +\infty]$. That is, $\bigstar$ denotes all non-negative intervals. Then, we have $\mathbb{A}^{\#} : Var \to \mathit{Interval}^{\#}$, and $\mathbb{A}^{\#\,\mathbb{K}} = \prod_{k\in\mathbb{K}} \mathbb{A}^{\#}$. We can calculate the transfer functions for expressions by following the sound-

ness condition: $\overline{\mathcal{A}'^{\#}}[\![e]\!] \dot{\sqsupseteq} \overline{\widehat{\alpha}}^{\#} \circ \overline{\mathcal{A}'}[\![e]\!] \circ \overline{\gamma}^{\#}$. The resulting functions can be computed effectively (in constant time) for constants and all binary operators as follows: $\overline{\mathcal{A}'^{\#}}[\![n]\!] = \lambda\overline{a^{\#}}.(\texttt{if}\ \ n \geq 0\ \texttt{then}\ \overline{\bigstar}\ \texttt{else}\ \overline{\top})$, $\overline{\mathcal{A}'^{\#}}[\![e_1 - e_2]\!] = \lambda\overline{a^{\#}}.\overline{\top}$, and $\overline{\mathcal{A}'^{\#}}[\![e_1 + e_2]\!] = \lambda\overline{a^{\#}}.\prod_{k\in\mathbb{K}} \pi_k(\overline{\mathcal{A}'^{\#}}[\![e_1]\!]\overline{a^{\#}}) \sqcup \pi_k(\overline{\mathcal{A}'^{\#}}[\![e_2]\!]\overline{a^{\#}})$, where $\overline{\bigstar} = (\bigstar, \ldots, \bigstar), \overline{\top} = (\top, \ldots, \top) \in \mathit{Interval}^{\#\,\mathbb{K}}$. The analysis approximately tracks integer constants n, i.e. non-negative values get abstracted to $\bigstar$, whereas negative values to $\top$. Note that the addition "+" operator (similarly "*" and "/") is interpreted as the least upper bound $\sqcup$, so that for a configuration $k \in \mathbb{K}$, $e_1 + e_2$ evaluates to $\bigstar$ only when both e_1 and e_2 are $\bigstar$. For the subtraction "-" operator, the analysis always produces $\top$, thus losing precision. Also note that since the pre-analysis works on a lattice with finite height ($\mathit{Interval}^{\#}$) there is no need of defining widening operators to compute the fixed point of `while` loops. In contrast, the (main) interval analysis works on a lattice with infinite ascending chains ($\mathit{Interval}$) so it needs widening operators for handling loops.

Constructing Abstractions. We can use the results obtained during the pre-analysis to: (1) find queries that are likely to benefit from increased variability-awareness of the main analysis; (2) find configurations and features that are worth being distinguished during the main analysis. The found *configurations* and *features* are used to construct an abstraction α, which instructs how much variability-awareness the main analysis should use.

First, we find whether a query can benefit from increased variability-awareness. For simplicity, we assume that there is only one query $q = (s, P, \mathtt{x}) \in \mathit{Stm} \times \mathcal{P}(\mathbb{P}) \times \mathit{Var}$. The analysis should prove the query $q = (s, P, \mathtt{x})$ by computing a lifted store $\overline{a}$ after analyzing the statement s, and checking for which $k \in \mathbb{K}$ it holds: $\pi_k(\overline{a})(\mathtt{x}) \sqsubseteq_{\mathbb{P}} \sqcup P$. To find whether the given query will benefit from increased variability-awareness, we run the variability-aware pre-analysis. Let $\overline{\mathcal{A}^{\#}}[\![s]\!]\overline{a_0^{\#}}$ be the result of the pre-analysis, where $\overline{a_0^{\#}}$ denotes the initial abstract lifted store with all variables set to $\top_{\mathbb{P}^{\#}}$. Using this result, we check if there is some $k \in \mathbb{K}$ such that: $\pi_k(\overline{\mathcal{A}^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})) = \bigstar$. Let $\mathbb{K}_{promise} \subseteq \mathbb{K}$ be the set of all *promising configurations* k that satisfy the above equation for a selected query.

We now compute the set $\mathbb{F}_{good} \subseteq \mathbb{F}$ of *necessary features* for a given query via dependency analysis, which is simultaneously done with the pre-analysis as follows. Let $\mathbb{P}'^{\#} = \mathbb{P}^{\#} \times \mathcal{P}(\mathbb{F})$. The idea is to over-approximate the set of features involved in analyzing each variable in the second component of $\mathbb{P}'^{\#}$. The abstract domain is $\mathbb{A}^{\#} = \mathit{Var} \to \mathbb{P}'^{\#}$. For lifted abstract store $\overline{a^{\#}} \in \mathbb{A}^{\#\,\mathbb{K}}$, we define $\pi_k(\overline{a^{\#}}(x))|_1 \in \mathbb{P}^{\#}$ as the property associated with the variable x in the component of $\overline{a^{\#}}$ corresponding to $k \in \mathbb{K}$; and $\pi_k(\overline{a^{\#}}(x))|_2 \in \mathcal{P}(\mathbb{F})$ as the set of features involved in producing the analysis result for x in the component of $\overline{a^{\#}}$ corresponding to $k \in \mathbb{K}$. The abstract semantics $\overline{\mathcal{A}'^{\#}}[\![e]\!]^F$ and $\overline{\mathcal{A}^{\#}}[\![s]\!]^F$ are the same as before except that they also maintain the set of involved features $F \subseteq \mathbb{F}$. The parameter $F \subseteq \mathbb{F}$ is propagated for all sub-statements of statements. For example: $\overline{\mathcal{A}^{\#}}[\![\texttt{\#if}\ (\theta)\ s]\!]^F = \lambda\overline{a^{\#}}.\prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{\mathcal{A}^{\#}}[\![s]\!]^{F\cup FV(\theta)}\overline{a^{\#}}) & \text{if } k \models \theta \\ \pi_k(\overline{a^{\#}}) & \text{if } k \not\models \theta \end{cases}$, where

$FV(\theta)$ denotes the set of features occurring in θ. For the `#if` statement, we also propagate the set of features in θ for each configuration k that satisfies θ, since the analysis result for those configurations will depend on features in θ. For "`x := e`", we record in the analysis which features have contributed for calculating the given property of `x`. We compute $\mathbb{F}_{good}$ as the union of all S, such that for some $k \in \mathbb{K}_{promise}$ we obtain: $\pi_k(\overline{\mathcal{A}^\#}[\![s]\!]^\emptyset \overline{a^\#}(\mathtt{x})) = (\bigstar, S)$. Then, we set $\mathbb{F}_{ignore} = \mathbb{F} \backslash \mathbb{F}_{good}$. The final constructed abstraction is: $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}_{ignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\vee_{k \in \mathbb{K}_{promise}} k}$.

Example 2. If we calculate $\overline{\mathcal{A}^\#}[\![P]\!]^\emptyset \overline{a_0^\#}$, where P is our example from Section 2 and $\overline{a_0^\#} = \prod_{k \in \mathbb{K}} (\lambda x.(\top, \emptyset))$ is the initial store, we obtain the final store: $([\mathtt{x} \mapsto (\bigstar, \{A\}), \mathtt{y} \mapsto (\top, \{B\})], [\mathtt{x} \mapsto (\bigstar, \{A\}), \mathtt{y} \mapsto (\top, \{B\})], [\mathtt{x} \mapsto (\top, \{A\}), \mathtt{y} \mapsto (\top, \{B\})], [\mathtt{x} \mapsto (\top, \{A\}), \mathtt{y} \mapsto (\top, \{B\})])$.

Therefore, we select the first query that asks for non-negative values of `x` as promising with $\mathbb{K}_{promise} = \{A \wedge B, A \wedge \neg B\}$ (which contains all configurations where `x` is mapped to $\bigstar$), $\mathbb{F}_{good} = \{A\}$, and $\mathbb{F}_{ignore} = \{B\}$. The abstraction regarding the first query is: $\boldsymbol{\alpha}^{\text{fignore}}_{B} \circ \boldsymbol{\alpha}^{\text{proj}}_{(A \wedge B) \vee (A \wedge \neg B)}$. But the second query that asks for non-negative values of `y` is rejected, since `y` is always mapped to $\top$. □

Finally, by using the soundness of pre-analysis, and suitability of the pre-analysis for the given query (definitions of $\mathbb{K}_{promise}$ and $\mathbb{F}_{ignore}$), we can show:

Theorem 1 (Promising Preservation). *Let $\mathbb{F}_{ignore}$ and $\mathbb{K}_{promise}$ be the sets of ignored features and promising configurations for a query $(s, P, \mathtt{x})$ defined by the result of our pre-analysis $\overline{\mathcal{A}^\#}[\![s]\!]^\emptyset \overline{a_0^\#}$. Let $\alpha = \boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}_{ignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\vee_{k \in \mathbb{K}_{promise}} k}$ and $\gamma = \boldsymbol{\gamma}^{\text{proj}}_{\vee_{k \in \mathbb{K}_{promise}} k} \circ \boldsymbol{\gamma}^{\text{fignore}}_{\mathbb{F}_{ignore}}$. Then: $\gamma\big(\overline{\mathcal{A}}_\alpha[\![s]\!]\overline{a_0}(\mathtt{x})\big) \dot{\sqsubseteq} \overline{\gamma}^\#\big(\overline{\mathcal{A}^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)$, where $\overline{a_0} \in \mathbb{A}^{\alpha(\mathbb{K})}$ and $\overline{a_0^\#} \in \mathbb{A}^{\#\,\mathbb{K}}$ are the initial (uninitialized) lifted stores.*

6 Evaluation

We now evaluate our pre-analysis guided approach for finding suitable variability abstractions for lifted analysis. For our experiments, we use SOOT's intra-procedural dataflow analysis framework [27] for analyzing Java programs and an existing SOOT extension for lifted dataflow analyses of Java program families [3]. The lifted dataflow analysis framework uses CIDE (Colored IDE) [17], which is an Eclipse plug-in, to annotate statements using background colors rather than `#ifdef` directives. Every feature in a program family is thus associated with a unique color. We consider optimized lifted intra-procedural analyses with improved representation via sharing of analysis equivalent configurations using a high-performance bit vector library [15]. Note that our pre-analysis guided approach for lifted analysis is orthogonal to the particular analysis chosen as a client, since it depends only on variability-specific constructs of the language.

First, we have implemented interval pre-analysis and interval analysis in the SOOT framework. For interval analysis, the delayed widening is implemented using the *flowThrough* method of *ForwardFlowAnalysis* class by counting the

times a node was visited and applying a widening operator once a threshold has been reached. Then, on top of a lifted dataflow analyzer [3], we have implemented variability-aware versions of interval pre-analysis and interval analysis described in Section 5 and Section 4, respectively. The pre-analysis reports a set of promising configurations and a set of features that should be ignored. This information is used to construct an abstraction, which is passed as parameter to the subsequent variability-aware interval analysis. The implemented analysis tracks the range of possible values for all integer (*int* and *long*) variables.

For our experiment, we use three Java benchmarks from the CIDE project [17]. Graph PL (GPL) is a small desktop application with intensive feature usage. It contains about 1,35 kLOC, 18 features, and 19 methods with integer variables. Prevayler is a slightly larger product line with low feature usage, which contains 8 kLOC, 5 features, and 174 methods with integer variables. BerkeleyDB is a larger database library with moderate feature usage, containing about 84 kLOC, 42 features, and 2654 methods with integer variables.

All experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU with 8 GB memory. All times are reported as averages over ten runs with the highest and lowest number removed. We report only the times needed for actual intra-procedural analyses to be performed. In experiments, to illustrate our approach we consider *queries* which ask for the exact non-negative possible values of local integer variables at the end (final nodes) of their methods.

Table 1 compares the performance of our approach based on pre-analysis followed by the corresponding abstract variability-aware interval analysis (table below) with full variability-aware interval analysis which is used as a baseline (table above). We measured the analysis precision by the number of integer variables for which our approach accurately calculates their analysis information (see full precision column, table below), which can be a non-negative interval (var $[\,]$) or the $\top$ value (var $\top$), such that the same analysis results are obtained with the full variability-aware interval analysis (analysis results column, table above). We report the number of configurations (con $[\,]$ and con $\top$) in which those precisely tracked variables occur. We also measured the number of variables and corresponding configurations where there is a precision loss (see precision loss column, table below), i.e. our approach produces the $\top$ value but the full variability-aware interval analysis can establish that their intervals are non-negative. For each of the benchmarks, we only analyze the methods that contain integer variables. We report the sum of analysis times for all such methods in a benchmark. We can see that for GPL and Prevayler there is no precision loss with our approach, but we obtain speed-ups in running times. For GPL we observe 2.2 times speed-up, whereas for Prevayler we have 1.3 times speed-up (pre-analysis+abstract vs. unabstract variability-aware analysis). For BerkeleyDB, we have precision loss for 3 variables found in 43 valid configurations (out of 7386 configurations where integer variables occur) which represents 0.58% precision loss in total, but we still keep precision for all the other 7386-43=7343 cases (configurations). Yet, we achieve 1.5 times speed-up with our approach for BerkeleyDB.

Benchmark	*Unabstract variability-aware analysis*				
	analysis results				Time
	var [ ]	con [ ]	var $\top$	con $\top$	
`GPL`	33	216	18	26	73.1
`Prevayler`	56	58	166	168	83.2
`BerkeleyDB`	1144	3197	2139	4189	2908

Benchmark	*Pre-analysis guided approach*						
	full precision				prec. loss		Time
	var [ ]	con [ ]	var $\top$	con $\top$	var $\top$	con $\top$	
`GPL`	33	216	18	26	0	0	33.4
`Prevayler`	56	58	166	168	0	0	62.3
`BerkeleyDB`	1141	3154	2137	4232	3	43	1933

Table 1: Performance results for unabstract variability-aware analysis which is used as a baseline (table above) vs. our pre-analysis guided approach which consists of running a pre-analysis followed by a subsequent abstract variability-aware analysis (table below). All times are in ms (milliseconds).

7 Related Work and Conclusion

Using pre-analysis to adjust the main analysis precision was first introduced in [23, 24]. They design pre-analysis for finding various precision parameters, such as: context sensitivity, flow sensitivity, and relational constraints between variables. In this work, we adapt this idea to the setting of variability-aware analysis for program families. In [20], machine learning is used to find a minimal abstraction that is sufficient to prove all queries provable by the most precise abstraction. The technique presented in [28] finds the optimum abstraction that proves a given query, but it is applicable only to disjunctive analysis.

The work in [3] lifts a dataflow analysis from the monotone framework, resulting in a variability-aware dataflow analysis that works on the level of families. Another efficient implementation of the lifted dataflow analysis formulated within the IFDS framework [25] was proposed in SPL$^{\text{LIFT}}$ [2]. However, this technique is limited to work only for analyses phrased within the IFDS framework [25], and many dataflow analyses, including interval, cannot be encoded in IFDS. Other approaches for lifting existing analysis techniques to work on the level of families are: lifted syntax checking [19], lifted type checking [4, 18], lifted model checking in the settings of transition systems [1, 6, 5, 12, 13] and game semantics [10, 11], lifted testing [16]. All these lifted techniques could benefit from using variability abstractions and from the present approach on finding a good abstraction.

To conclude, in this work we presented a technique for automatically finding abstractions that enable effective abstract variability-aware analysis. The suitable abstraction parameters are calculated by a pre-analysis. We demonstrate the effectiveness of our approach with experiments.

References

1. Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.
2. Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl^{lift}: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.
3. Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *T. Aspect-Oriented Software Development*, 10:73–108, 2013.
4. Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012.
5. Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.
6. Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010*, pages 335–344, 2010.
7. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
8. Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *POPL'77*, pages 238–252, Los Angeles, California, January 1977.
10. Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014.
11. Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.
12. Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Family-based model checking without a family-based model checker. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings*, volume 9232 of *LNCS*, pages 282–299. Springer, 2015.
13. Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Efficient family-based model checking via variability abstractions. *STTT*, 2016.
14. Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
15. CERN: European Organization for Nuclear Research. The colt project: Open source libraries for high performance scientific and technical computing in java. In *CERN*, 1999.

16. Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 597–607, 2015.
17. Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.
18. Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.
19. Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, pages 805–824, 2011.
20. Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 31–42, 2011.
21. Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015.
22. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Secaucus, USA, 1999.
23. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, page 49, 2014.
24. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective x-sensitive analysis guided by impact pre-analysis. *ACM Trans. Program. Lang. Syst.*, 38(2):6, 2016.
25. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. 22nd ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, POPL '95, pages 49–61, 1995.
26. Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.
27. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON'99)*, pages 13–. IBM Press, 1999.
28. Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 365–376, 2013.