Finding suitable variability abstractions for lifted analysis

Article *in* Formal Aspects of Computing · March 2019

DOI: 10.1007/s00165-019-00479-y

CITATION
1

READS
32

Under consideration for publication in Formal Aspects of Computing

Finding Suitable Variability Abstractions for Lifted Analysis

Aleksandar S. Dimovski[1], Claus Brabrand[2], and Andrzej Wąsowski[2]

[1] Mother Teresa University, 12 Udarna Brigada 2a, 1000 Skopje, Makedonija
[2] IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen, Denmark

Abstract. Many software systems are today variational: they are built as program families or Software Product Lines. They can produce a potentially huge number of related programs, known as products or variants, by selecting suitable configuration options (features) at compile time. Many such program families are safety critical, yet the appropriate tools only rarely are able to analyze them effeciently. Researchers have addressed this problem by designing specialized *variability-aware* static (dataflow) analyses, which allow analyzing all variants 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. They take as input the common code base, which encodes all variants of a program family, and produce precise analysis results corresponding to all variants. These analyses scale much better than "brute force" approach, where all individual variants are analyzed in isolation, one-by-one, using off-the-shelf single-program analyzers. Nevertheless, the computational cost of lifted analyses still greatly depends on the number of features and variants (which is often huge). For families with a large number of features and variants, the lifted analyses may be too costly or even infeasible . In order to speed up lifted analyses and make them computationally cheaper, variability abstractions which simplify variability away from program families and lifted analyses have been introduced. However, the space of possible variability abstractions is still intractably large to search naively, with most abstractions being either too imprecise or too costly.

We introduce here a method to efficiently find suitable variability abstractions from a large space of possible abstractions for a lifted static analysis. The main idea is to use a *pre-analysis* to estimate the impact of variability-specific parts of the program family on the analysis's precision. The pre-analysis is fully variability-aware while it aggressively abstracts the other semantics aspects. Then we use the pre-analysis results to find out when and where the subsequent abstract lifted analysis should turn off or on its variability-awareness. The abstraction constructed in this way is effective in discarding variability-specific program details that are irrelevant for showing the analysis's ultimate goal. We formalize this approach and we illustrate its effectiveness on several Java case studies. The evaluation shows that our approach which consists of running a pre-analysis followed by a subsequent abstract lifted analysis achieves competitive the precision-speed tradeoff compared to the standard lifted analysis.

Keywords: Program Families, Lifted Static Analysis, Variability Abstractions, Abstract Interpretation

Correspondence and offprint requests to: Christiane Notarmarco, Springer-Verlag London Limited, Sweetapple House, Catteshall Road, Godalming, Surrey GU7 3DJ, UK. e-mail: chris@svl.co.uk

1. Introduction

The strong trend for customization in modern economy leads to construction of many variational (highly configurable) systems. Efficient methods to achieve customization, such as *Software Product Lines* (SPLs) [CN01], use *features* (statically configured options) to control presence and absence of software functionality in a product family. Different family members, called *variants* or *valid products*, are derived by switching features on and off, while reuse of the common code is maximized. SPLs are particularly popular in commercial embedded software (e.g., cars, phones, avionics, health-care), system level software (e.g., the Linux kernel), frameworks and development platforms (e.g., plugins provide features in Eclipse), as well as in many web solutions (e.g., Wordpress). Although there are different strategies for implementing product lines [TAK$^+$14], many popular industrial product lines are implemented using annotative approaches such as conditional compilation (e.g., `#ifdef` construct from the C-Preprocessor [KAK08]). They enable a simple form of two staged computation in preprocessor style. At build time, the program family is first configured and a variant describing a particular product is derived by selecting a set of features relevant for it, and only then the derived variant is compiled or interpreted.

Unfortunately, SPLs are difficult to design, and difficult to verify. Therefore, their formal analysis and verification represents a very important challenge in development of SPLs [TAK$^+$14]. In this work, we study sound static analysis techniques based on abstract interpretation [CC77, CC79, NNH99]. The simplest *"brute-force" approach* to verify such program families is to generate all valid variants of a family using a preprocessor, and then apply an existing single-program analysis technique to each resulting variant [BRT$^+$13]. However, this approach does not scale up in practice for SPLs with high variability since the number of possible variants (i.e. the size of configuration space) is *exponential* in the number of features. All variants will be analyzed independently one by one despite of their great similarity. *Variability-aware* (lifted, family-based) 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. This is the case especially for families with higher variability. However, the computational cost of the variability-aware analysis still depends on the size of the configuration space (the number of variants). In order to further scale the variability-aware analysis, we follow the classic route in the analysis space: loosing precision in favour of gaining performance, by means of abstraction. The so-called variability abstractions [DBW15] aim to abstract from certain aspects of the configuration space, so that many of the configurations (variants) become indistinguishable and can be collapsed into a single abstract configuration. This results in smaller abstract families with a smaller number of abstract configurations, such that each abstract configuration corresponds to some subset of concrete configurations. Each *variability abstraction* expresses a compromise between precision and speed in the induced abstract variability-aware analysis. At one end of the spectrum, we find *identity* abstraction which is fully precise, but slow. At the other extreme, we have the *join-everything* abstraction which is fast, but not very precise. 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 space (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 either too imprecise or too costly to show the analysis's ultimate goal.

Here, we introduce an efficient method to address the above search problem by finding an abstraction that discards program details that are unnecessary for proving an individual query. 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 that performs the requested client analysis. This analysis uses the results of pre-analysis, selects influential features and variants for precision, and selectively applies variability-awareness only to those features and variants. All other features and variants are abstracted away.

The pre-analysis represents an over-approximation of the main analysis. However, the pre-analysis uses very simple abstract domain and transfer functions, so it can be efficiently run even with full variability-awareness. The soundness condition says that all components of the pre-analysis have to over-approximate the corresponding ones of the main (variability-aware) analysis. This is identical to the standard soundness requirement of a static program analysis, except that the condition is not stated over the concrete semantics

of a program, but over the main variability-aware analysis. There is a difference between the pre-analysis and the resulting abstract variability-aware main analysis. The pre-analysis is more precise in terms of variability-awareness (it is full variability-aware with no variability abstraction), while the resulting abstract variability-aware analysis is more precise in tracking non-variability specific parts of the program family (i.e. language specific parts that operate on the program state). Our construction aims to identify which features and variants the pre-analysis will distinguish in order to apply variability awareness on the main analysis only when it is likely to benefit the final analysis result. Thus, we construct an abstraction which is effective at slicing away (discarding) variability-specific program details that are irrelevant for showing the analysis's goal. Our method ensures that for a given set of queries the resulting abstract variability-aware analysis is at least as precise as the fully variability-aware pre-analysis. However, the resulting abstract variability-aware analysis may lose some precision with respect to the fully variability-aware main analysis and therefore may not prove some query, when the pre-analysis returns too coarse answer for the query due to its own over-approximation. We evaluate our approach by comparing the constructed abstract variability-aware analysis and the standard variability-aware analysis with no abstraction, when applied to several Java SPL case studies. We quantify to what extent precision can be traded for speed in those cases. Note that our approach is general and applicable to any main static analysis chosen as a client. We demonstrate the applicability of our approach to the interval analysis.

In this work, we make the following contributions:

- We show how to design and use a pre-analysis that estimates the impact of variability on a client (main) analysis.
- We present a technique for constructing a suitable abstract variability-aware analysis that receives guidance from the pre-analysis, such that its results are at least as precise as the pre-analysis results.
- We experimentally show the effectiveness of the abstract variability-aware analysis designed using our technique, when applied to three Java benchmarks with different sizes and different variability usage.

This work represents an extended and revised version of the conference paper [DBW16]. Compared to the earlier work, we make the following extensions here. We motivate the practicality of finding suitable variability abstractions for lifted analysis. We expand and elaborate the examples as well as the discussion on how this approach works. We provide correctness proofs for all main results (Theorem 6.1). We show how to generalize the definition of lifted analysis domain as a binary decision diagram, so that there is an explicit interaction (sharing) between analysis results corresponding to different configurations. Finally, we augment the evaluation of the approach by defining precise objectives and experimental setup, providing more performance results, and considering a possible application scenario for this approach.

We proceed by giving a motivating example for finding suitable variability abstractions for lifted analysis in Section 2. In Section 3 we introduce the language for writing program families. The basics of lifted analysis based on abstract interpretation are introduced in Section 4, whereas variability abstractions and abstract lifted analyses derived using them are introduced in Section 5. Section 6 explains how to construct an appropriate abstraction for the lifted analysis based on a specially designed pre-analysis. In Section 7, we show how our lifted analysis domain can be generalized to a binary decision diagram domain, and describe the induced lifted analysis based on such lifted domains. Section 8 presents the evaluation on three Java SPLs. Finally, we discuss the relation to other works, conclude, and then present a recap table to repeat the commonly used notations and symbols in this paper.

2. Motivating Example

To better illustrate the issues we are addressing in this work, we now present a motivating example. Consider the following program family P:

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

The set of available (Boolean) features is $\mathbb{F} = \{A, B\}$, and 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. The family P contains

```
int x := 0;
x := x+2;
y := y+2
```
(a) Variant for $A \wedge B$.

```
int x := 0;
x := x+2;
```
(b) Variant for $A \wedge \neg B$.

```
int x := 0;
y := y+2
x := x-2
```
(c) Variant for $\neg A \wedge B$.

```
int x := 0;
x := x-2
```
(d) Variant for $\neg A \wedge \neg B$.

Table 1. Variants derived from the program family P.

three `#if` statements, which increase and decrease the variables `x` and `y`, depending on which features from $\mathbb{F}$ are enabled. 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), which will make both assignments in lines 2 and 3 to appear in the variant. On the other hand, the variant for the configuration $\neg A \wedge \neg B$ will have both features A and B disabled (set to false), so that only the assignment in line 4 will appear in this variant. Variants corresponding to all configurations are illustrated in Table 1.

The *interval analysis* computes the set of possible values for each variable as an interval. The basic properties are intervals of the form: $[l, h]$, where $l \in \mathbb{Z} \cup \{-\infty\}$, $h \in \mathbb{Z} \cup \{+\infty\}$, and $l \leq h$. Intuitively, $[l, h]$ is the interval of values from l to h including the end points when they are in $\mathbb{Z}$. The coarsest property is $\top = [-\infty, +\infty]$.

We want to check the following two *queries* on P: "find all configurations for which variables `x` and `y` are non-negative in the final point of P, and in those cases determine *accurately* the corresponding non-negative intervals".

Full variability-aware analysis Full variability-aware analysis, or simply lifted analysis, operates on lifted stores, $\overline{a}$, that contain one separate component for every valid configuration from $\mathbb{K}$. For the compile-time conditional statement "`#if` (θ) s", lifted analysis checks for each $k \in \mathbb{K}$ 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, if θ is not satisfied by k, the corresponding component of the lifted store is not updated. We assume that the initial lifted store consists of uninitialized `x` and `y`, thus they have the coarsest property $\top$ in the initial point. 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 $\neg A \wedge \neg B$. For clarity, we will often explicitly write the configurations over the individual components of lifted stores. We write $\overline{a} \overset{\text{line n}}{\longmapsto} \overline{a'}$ when the lifted store $\overline{a'}$ is the result of analyzing the statement in "line n" at the input lifted store $\overline{a}$.

$$\big(\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{A\wedge B},\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{A\wedge\neg B},\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{\neg A\wedge B},\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{\neg A\wedge\neg B}\big)$$
$$\overset{\text{line 1}}{\longmapsto} \big([\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 2}}{\longmapsto} \big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 3}}{\longmapsto} \big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 4}}{\longmapsto} \big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[\text{-}2,\text{-}2],\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto[\text{-}2,\text{-}2],\mathtt{y}\mapsto\top]\big)$$

As the result of lifted analysis of P, we can successfully answer the first query by deducing that `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 $\neg A \wedge \neg B$). On the other hand, `y` is always $\top = [-\infty, +\infty]$ so we cannot answer the second query regarding `y`.

Need for abstraction Despite its preciseness, using full variability-aware analysis is not always the best solution. It is often computationally very expensive to run such an analysis with high number of configurations. More importantly, in many cases, full variability-awareness does not help, that is 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`, so we still cannot answer the second query. 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, that is, if we only want to answer the first query. Moreover, we can see that analyzing the feature B is unnecessary for establishing the exact interval of x.

Variability abstractions We introduce a range of variability abstractions [DBW15] which aim to reduce the size of configuration space to something more tractable. In effect, we obtain computationally cheaper but less precise abstract variability-aware analyses, since an over-approximation is introduced in them. The three basic abstractions we consider are: (1) to *confound* (join) configurations into a single abstract one, denoted $\boldsymbol{\alpha}^{\text{join}}$, (2) to project (*divide-and-conquer*) the configuration space onto a subset satisfying the constraint φ, denoted $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}$, and (3) to ignore a feature (or a set of features), $A \in \mathbb{F}$, deemed as not relevant for the current problem, denoted $\boldsymbol{\alpha}^{\text{fignore}}_{A}$. With join abstraction, all variants (configurations) are merged into a single abstract one, so that the obtained analysis has no variability to consider, at the cost of introducing additional control flow obtained by confounding the control flow of all individual variants. With divide-and-conquer we analyze only a subset of configurations at a time. With feature ignore the differences in behaviour caused by the ignored features are effectively annihilated, so that the resulting abstract analysis has less variability to consider. We also use sequential composition, denoted $\circ$, and product, denoted $\otimes$, to build other more sophisticated compound abstractions out of these three basic ones. Any such constructed abstraction α induces an abstract variability-aware analysis, denoted $\overline{\mathcal{A}}_{\alpha}$, which is derived in [DBW15] using the calculational approach of abstract interpretation developed in [Cou99]. Since variability abstractions affect only the variability-specific aspect of the variability-aware analysis (i.e. the transfer function of `#if` statement), they can be also defined as source-to-source transformations [DBW15]. In particular, for each program family P and abstraction α, we can define an abstract program $\alpha(P)$ with less number of abstract configurations, such that $\overline{\mathcal{A}}_{\alpha}[\![P]\!] = \overline{\mathcal{A}}[\![\alpha(P)]\!]$ where $\overline{\mathcal{A}}$ represents (unabstracted) full variability-aware analysis.

The coarsest abstraction If we apply the coarsest join abstraction $\boldsymbol{\alpha}^{\text{join}}$, which confounds control-flow of all valid configurations into a single program with over-approximated control-flow and no variability into it, the result is the following single program $\boldsymbol{\alpha}^{\text{join}}(P)$:

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

where $*$ models an arbitrary integer. The single abstract configuration corresponding to $\boldsymbol{\alpha}^{\text{join}}(P)$ is $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\text{true}\}$ (that is, true $\equiv (A \wedge B) \vee (A \wedge \neg B) \vee (\neg A \wedge B) \vee (\neg A \wedge \neg B)$). 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 $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{join}}}$. Note that $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{join}}}$ operates on stores that are 1-sized tuples. If we apply $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{join}}}$ to P, we obtain:

$$\big(\overbrace{[\mathtt{x}\mapsto\top, \mathtt{y}\mapsto\top]}^{\text{true}}\big) \overset{\text{line 1}}{\longmapsto} \big([\mathtt{x}\mapsto[0,0], \mathtt{y}\mapsto\top]\big) \overset{\text{line 2}}{\longmapsto} \big([\mathtt{x}\mapsto[0,2], \mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 3}}{\longmapsto} \big([\mathtt{x}\mapsto[0,2], \mathtt{y}\mapsto\top]\big) \overset{\text{line 4}}{\longmapsto} \big([\mathtt{x}\mapsto[-2,2], \mathtt{y}\mapsto\top]\big)$$

As result of the above $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{join}}}$ analysis, in the final point of P we obtain the output store $\big([\mathtt{x}\mapsto[\text{-}2,2], \mathtt{y}\mapsto\top]\big)$. However, these analysis results are not strong enough to prove any of our two 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 and help to successfully answer the given queries. For the example program 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. Moreover, our method should detect that we can bring the full benefit of variability-awareness for the first query by taking into account only valid configurations that satisfy A (that is, $A \wedge B$ and $A \wedge \neg B$). 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, thus obtaining the final abstraction $\boldsymbol{\alpha}^{\text{fignore}}_{B} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}$. Since A and B are the only features in our family, the last abstraction is equivalent to $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}$ obtained by confounding the

control-flow of configurations $A \wedge B$ and $A \wedge \neg B$. Thus, we obtain the following abstract (single) program $\boldsymbol{\alpha}_B^{\text{fignore}} \circ \boldsymbol{\alpha}_A^{\text{proj}}(P)$:

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

The abstract configuration corresponding to this program is $\boldsymbol{\alpha}_B^{\text{fignore}} \circ \boldsymbol{\alpha}_A^{\text{proj}}(\mathbb{K}) = \{A\}$. The single-program interval analysis of the above program proceeds as:

$$\big(\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{A}\big) \overset{\text{line 1}}{\longmapsto} \big([\mathtt{x}\mapsto[0,0],\mathtt{y}\mapsto\top]\big) \overset{\text{line 2}}{\longmapsto} \big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 3}}{\longmapsto} \big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top]\big) \overset{\text{line 4}}{\longmapsto} \big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top]\big)$$

From the output store $\big([\mathtt{x}\mapsto[2,2],\mathtt{y}\mapsto\top]\big)$ in the final point of P, we can establish that the first query holds for all configurations that satisfy A. Therefore, the analysis correctly infers that `x` is positive after line 4 for configurations that satisfy A.

Pre-analysis The key idea is to estimate the impact of variability on the main analysis by using a specially designed pre-analysis. The pre-analysis is designed to use a simple abstract domain and simple transfer functions, and so it can be run efficiently even with full variability-awareness. For example, we design a pre-analysis that over-approximates the interval analysis, such that its abstract domain is: $Var \to \{\bigstar, \top\}$, where $\bigstar$ denotes all non-negative intervals, that is any sub-interval of $[0, +\infty]$. This simple abstract domain of the pre-analysis is chosen because we are interested in showing queries that some variables are non-negative. This pre-analysis is run under full variability-awareness, i.e. for all configurations $k \in \mathbb{K}$ simultaneously. For our example program family P, we obtain:

$$\big(\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{A\wedge B},\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{A\wedge\neg B},\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{\neg A\wedge B},\overbrace{[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]}^{\neg A\wedge\neg B}\big)$$
$$\overset{\text{line 1}}{\longmapsto} \big([\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 2}}{\longmapsto} \big([\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 3}}{\longmapsto} \big([\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top]\big)$$
$$\overset{\text{line 4}}{\longmapsto} \big([\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\bigstar,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top]\big)$$

Note that the pre-analysis precisely estimates the impact of variability: it identifies where the interval analysis accurately tracks the possible (non-negative) values of `x` under full variability-awareness. In general, our pre-analysis might lose precision and produce $\top$ more often than in the ideal case. However, it does so only in a sound manner. If the pre-analysis computes $\bigstar$ for a given variable, then the full variability-aware interval analysis is guaranteed to compute an accurate non-negative interval as well.

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 results. In our example, pre-analysis assigns $\bigstar$ to `x` in two valid configurations. We regard this as a good indication that fully variability-aware interval analysis is likely to answer the first query (estimate the non-negative possible values of `x`) accurately. Then, for each query that is judged promising, we find variability-specific parts of the program family that contribute to the query. All the found variability-specific parts should be tracked precisely. First, we determine configurations for which the query is proven correct. In our example, those are configurations $A \wedge B$ and $A \wedge \neg B$. Therefore, we want to 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 establish that the feature B does not affect the possible values of `x` at all. Therefore, we can ignore the feature B thus obtaining the final abstraction $\boldsymbol{\alpha}_B^{\text{fignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{(A\wedge B)\vee(A\wedge\neg B)}$. For the second query that `y` is non-negative, we obtain that `y` is $\top$ for all configurations in the output lifted store. This is a

good indication that we cannot prove this query for any variant even with full variability-aware analysis. Our method guarantees that if the pre-analysis calculates ★ for a variable, then the constructed abstract variability-aware analysis will assign a 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 and answer correctly some queries, so there will be a precision loss in the constructed abstract variability-aware analysis.

3. Program Families

We start by defining features, configurations, and feature expressions. Then, we describe a simple imperative language $\overline{\text{IMP}}$ for writing program families.

Features, configurations, and feature expressions The available *features* are given by a set of Boolean variables $\mathbb{F} = \{A_1, \ldots, A_n\}$. Each feature may be *enabled* or *disabled* in a particular variant, thus controlling the presence and absence of software functionality. A *configuration* k is a truth assignment (a mapping from $\mathbb{F}$ to $\{\text{true}, \text{false}\}$) which gives a truth value to any feature. If a feature $A \in \mathbb{F}$ is enabled (included) 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 $true \cdot A = A$ and $false \cdot A = \neg A$. We write $\mathbb{K}$ for the set of all *valid* configurations defined over $\mathbb{F}$ for a program family. The set of valid configurations is typically described by a feature model [Bat05], but in this work we disregard syntactic representations of the set $\mathbb{K}$. In general, not every combination of features yields a valid configuration, thus $|\mathbb{K}| \leq 2^{|\mathbb{F}|}$.

We add a new syntactic category of *feature expressions*, denoted *FeatExp*($\mathbb{F}$), as the set of well-formed propositional logic formulas over $\mathbb{F}$, defined inductively as:

$$\theta ::= \text{true} \mid A \in \mathbb{F} \mid \neg\theta \mid \theta_1 \wedge \theta_2$$

We will use $\theta \in \textit{FeatExp}(\mathbb{F})$ to write presence conditions over features $\mathbb{F}$ in program families. We can also use $\psi \in \textit{FeatExp}(\mathbb{F})$ to represent a set of valid configurations. We write $\mathbb{K}_{[\![\psi]\!]}$ to denote the set of valid configurations described by ψ, such that each satisfiable valuation k of ψ (ψ evaluates to true under the valuation k) corresponds to a valid configuration. For example, let the set of features $\mathbb{F}$ be $\{A, B\}$. Some features expressions defined over $\mathbb{F}$ are: $A \vee B$, $A \wedge \neg B$, $\neg A$, etc. The feature expressions $A \vee B$ and true yield the following sets of valid configurations: $\mathbb{K}_{[\![A \vee B]\!]} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$ and $\mathbb{K}_{[\![true]\!]} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.

The programming language We consider the language $\overline{\text{IMP}}$ for writing program families. $\overline{\text{IMP}}$ extends the imperative language IMP [Win93] with a compile-time conditional statement for encoding multiple variants of a program. The new statement "`#if` (θ) s" contains a feature expression $\theta \in \textit{FeatExp}(\mathbb{F})$ 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 of the language is given by the following grammar:

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

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*. We use $\overline{\text{IMP}}$ only for presentational purposes as a well established minimal language. Still, the introduced methodology is not limited to $\overline{\text{IMP}}$ or its features. In fact, we evaluate our approach on program families written in Java.

The $\overline{\text{IMP}}$ programs are evaluated in two stages. First, a *preprocessor* takes as input an $\overline{\text{IMP}}$ program and a configuration $k \in \mathbb{K}$, and outputs a variant, i.e. a single IMP program without `#if`-s, corresponding to k. Second, the obtained variant is evaluated using the standard IMP semantics [Win93]. The first stage is specified by the projection function $\texttt{P}_k$, which copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP and recursively pre-processes all sub-statements of compound statements. Hence, we have $\texttt{P}_k(\texttt{skip}) = \texttt{skip}$ and $\texttt{P}_k(s;s') = \texttt{P}_k(s);\texttt{P}_k(s')$. The interesting case is "`#if` (θ) s" statement, where the statement s is included in the resulting variant iff $k \models \theta$ (where $\models$ denotes the standard satisfaction relation of propositional logic),

otherwise the statement s is removed. That is,

$$\mathtt{P}_k(\texttt{\#if } (\theta)\ s) = \begin{cases} \mathtt{P}_k(s) & \text{if } k \models \theta \\ \texttt{skip} & \text{if } k \not\models \theta \end{cases}$$

For example, the variants $\mathtt{P}_{A\wedge B}(P)$, $\mathtt{P}_{A\wedge \neg B}(P)$, $\mathtt{P}_{\neg A\wedge B}(P)$, $\mathtt{P}_{\neg A\wedge \neg B}(P)$ shown in Tables 1a, 1b, 1c, 1d, respectively, are derived from the program family P defined in Section 2.

4. Background: Lifted Analysis

Variability-aware (lifted) analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. In our case, the process of "lifting" means to take a static analysis that works on IMP programs, and to transform it into an analysis that works on $\overline{\text{IMP}}$ program families, without preprocessing them.

We start by summarizing the existing background for our work. We give an overview of the basic ideas and concepts of abstract interpretation. Then, we briefly sketch the process of "lifting" analysis introduced in [MDBW15]. Here, the focus is on interval analysis for presentation purposes; but our method is generically applicable to any (monotone) static analysis phrased as an abstract interpretation.

4.1. Abstract Interpretation

A *partial order* [NNH99] is a mathematical structure, $\langle L, \leq_L \rangle$, where L is a set equipped with a binary order relation, $\leq_L$, which is reflexive, antisymmetric, and transitive. Let $X \subseteq L$. We say that $u \in L$ is an *upper bound* for X, written $X \leq_L u$, if we have $\forall x \in X : x \leq_L u$. A *least upper bound*, written $\sqcup X$, is defined by: $\forall x \in X : x \leq_L \sqcup X \wedge \forall u \in L : X \leq_L u \implies \sqcup X \leq_L u$. (Similarly, *lower bound* and *greatest lower bound*, written $\sqcap X$, may be defined.) A *complete lattice* is a partial order for which $\sqcup X$ and $\sqcap X$ exist for all subsets $X \subseteq S$. As a consequence, a complete lattice will always have a *unique largest element*, $\top$, and a *unique smallest element*, $\bot$, defined as: $\top = \sqcup L$ and $\bot = \sqcap L$.

We consider the standard Galois connection based abstract interpretation [CC77]. A *Galois connection* (GC) is a pair of total functions, $\alpha : L \to M$ and $\gamma : M \to L$ (respectively known as the *abstraction* and *concretization* functions), connecting two complete lattices, $\langle L, \leq_L \rangle$ and $\langle M, \leq_M \rangle$, such that:

$$\forall l \in L, m \in M : \quad \alpha(l) \leq_M m \iff l \leq_L \gamma(m) \tag{1}$$

which is often typeset as: $\langle L, \leq_L \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leq_M \rangle$. For a concrete domain L, we define *abstraction* and *concretization* functions to and from a more abstract domain M, where information has been abstracted away. We will use Galois connections to approximate a computationally expensive (or uncomputable) analysis formulated over L with a computationally cheaper analysis formulated over M.

GCs have a number of important properties [CC92]:

- α and γ are *monotone*;
- $\alpha \circ \gamma$ is *reductive*, i.e. $(\alpha \circ \gamma)(m) \leq_M m$, for all $m \in M$;
- α is a *complete join morphism* (CJM), i.e. $\alpha(\bigcup_{l \in L} l) = \bigsqcup_{l \in L} \alpha(l)$, where $\cup$ and $\sqcup$ represent least upper bounds in L and M, respectively.
- The composition of Galois connections is a Galois connection. If $\langle L, \leq_L \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \leq_M \rangle$ and $\langle M, \leq_M \rangle \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \langle N, \leq_N \rangle$ then $\langle L, \leq_L \rangle \underset{\alpha' \circ \alpha}{\overset{\gamma \circ \gamma'}{\leftrightarrows}} \langle N, \leq_N \rangle$.

4.2. Lifting Single-program Analysis

We now specify a (monotone) static analysis for IMP phrased in the abstract interpretation framework [CC77, CC79, NNH99]. There is a complete lattice $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}}, \sqcup_{\mathbb{P}}, \sqcap_{\mathbb{P}}, \bot_{\mathbb{P}}, \top_{\mathbb{P}} \rangle$ for describing the *properties* of the analysis. More specifically, $\mathbb{P}$ is a set equipped with a partial order relation $\sqsubseteq_{\mathbb{P}}$, a least upper bound (join)

$\sqcup_{\mathbb{P}}$, a greatest lower bound (meet) $\sqcap_{\mathbb{P}}$, a least element (bottom) $\bot_{\mathbb{P}}$, and a greatest element (top) $\top_{\mathbb{P}}$. Then, we have an analysis 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* $\langle \mathbb{A}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ inherits the lattice structure from $\mathbb{P}$ in a point-wise manner. For example, $a \sqsubseteq a'$ iff $\forall \mathtt{x} \in \mathit{Var}$, $a(\mathtt{x}) \sqsubseteq_{\mathbb{P}} a'(\mathtt{x})$; $a \sqcup a' = \lambda \mathtt{x}.a(\mathtt{x}) \sqcup_{\mathbb{P}} a'(\mathtt{x})$; $a \sqcap a' = \lambda \mathtt{x}.a(\mathtt{x}) \sqcap_{\mathbb{P}} a'(\mathtt{x})$; $\bot = \lambda \mathtt{x}.\bot_{\mathbb{P}}$; and $\top = \lambda \mathtt{x}.\top_{\mathbb{P}}$. We also have *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.

By using variational abstract interpretation [MDBW15], we can lift any single-program analysis for IMP defined as above to the corresponding *variability-aware (lifted) analysis* for $\overline{\text{IMP}}$. The *lifted analysis domain* is $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, where $\mathbb{K}$ a set of valid configurations and $\mathbb{A}^{\mathbb{K}}$ is shorthand for the $|\mathbb{K}|$-fold product $\prod_{k \in \mathbb{K}} \mathbb{A}$, i.e. there is one separate copy of $\mathbb{A}$ for each configuration of $\mathbb{K}$. The lifted domain $\mathbb{A}^{\mathbb{K}}$ inherits the lattice structure of $\mathbb{A}$ in a product-wise manner. Given $\overline{a}, \overline{a}' \in \mathbb{A}^{\mathbb{K}}$, the lifted ordering $\dot{\sqsubseteq}$ is defined as: $\overline{a} \,\dot{\sqsubseteq}\, \overline{a}'$ 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 $\dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top}$. For example, $\overline{a} \,\dot{\sqcup}\, \overline{a}' = \prod_{k \in \mathbb{K}} \pi_k(\overline{a}) \sqcup \pi_k(\overline{a}')$; $\overline{a} \,\dot{\sqcap}\, \overline{a}' = \prod_{k \in \mathbb{K}} \pi_k(\overline{a}) \sqcap \pi_k(\overline{a}')$; $\dot{\bot} = \prod_{k \in \mathbb{K}} \bot = (\bot, \ldots, \bot)$ for $\bot \in \mathbb{A}$; and $\dot{\top} = \prod_{k \in \mathbb{K}} \top = (\top, \ldots, \top)$, for $\top \in \mathbb{A}$.

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. Thus, the *lifted transfer functions* are given by the functions: $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}}$, which represent tuples of $|\mathbb{K}|$ functions. The k-th component of the above functions defines the analysis corresponding to the configuration $k \in \mathbb{K}$. The definitions of $\overline{\mathcal{A}}[\![s]\!]$ and $\overline{\mathcal{A}'}[\![e]\!]$ are given in Fig. 1. 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 $\prod_{k \in \mathbb{K}} \pi_k(\overline{f})(\pi_k(\overline{a}))$.

The function $\overline{\mathcal{A}}[\![s]\!]$ (resp., $\overline{\mathcal{A}'}[\![e]\!]$) captures the effect of analysing the statement s (resp., expression e) in a lifted store $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ by computing an output lifted store $\overline{a'} \in \mathbb{A}^{\mathbb{K}}$ (resp., property $\overline{p} \in \mathbb{P}^{\mathbb{K}}$). 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 analysis of `if` statement results in the least upper bound (join) of the effects from the two corresponding branches. For the `while` statement, we compute the least fixed point (lfp) of a functional, $\lambda \overline{\Phi}.\, \lambda \overline{a}.\, \overline{a} \,\dot{\sqcup}\, \overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\, \overline{a})$, in order to capture the effect of running all possible iterations of the `while` loop. For complete lattices with finite height this fixed point exists and is computable by Kleene's fixed point theorem [CC79]. If the lattice is with infinite height and infinite ascending chains (e.g. the interval analysis domain), then the computation of the fixed point can be achieved by using widening operators [CC77, NNH99]. The analysis of "`#if` (θ) s" checks the relation between each valid configuration $k \in \mathbb{K}$ [1] whether the feature constraint θ is satisfied and, if so, it updates the corresponding component of the input store by the effect of evaluating the statement s. Otherwise, the corresponding component of the store is not updated. For expressions, we need a way of turning values into properties, which is specified by a function $\mathit{abst}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{P}$. Thus, the rule for constants n becomes: $\overline{\mathcal{A}'}[\![n]\!] = \lambda \overline{a}. \prod_{k \in \mathbb{K}} \mathit{abst}_{\mathbb{Z}}(n)$. For each (binary) operator $\oplus$, we assume that there is a corresponding abstract operator $\widehat{\oplus} : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$, which describes how the operator is defined on values from $\mathbb{P}$. The monotonicity of $\overline{\mathcal{A}}[\![s]\!]$ and $\overline{\mathcal{A}'}[\![e]\!]$ was shown in [MDBW15].

4.3. Lifted Interval Analysis

We will now use the interval analysis to illustrate our method for lifting. For each program point, the interval analysis identifies the range of possible values for every variable. The property domain $\langle \mathit{Interval}, \sqsubseteq_I \rangle$ is:

$$\mathit{Interval} = \{\bot_I\} \cup \{[l, h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$$

where $\bot_I$ denotes the empty interval, and $\top_I = [-\infty, +\infty]$. The partial ordering $\sqsubseteq_I$ is:

$$[l_1, h_1] \sqsubseteq_I [l_2, h_2] \text{ iff } l_2 \leq l_1 \wedge h_1 \leq h_2$$

[1] Since any $k \in \mathbb{K}$ is a valuation, we have that $k \not\models \theta$ and $k \models \neg\theta$ are equivalent for any $\theta \in \mathit{FeatExp}(\mathbb{F})$.

$$\overline{\mathcal{A}}[\![\texttt{skip}]\!] = \lambda\overline{a}.\,\overline{a}$$

$$\overline{\mathcal{A}}[\![\texttt{x := } e]\!] = \lambda\overline{a}.\prod_{k\in\mathbb{K}}(\pi_k(\overline{a}))[\texttt{x} \mapsto \pi_k(\overline{\mathcal{A}'}[\![e]\!]\overline{a})]$$

$$\overline{\mathcal{A}}[\![s_0 \texttt{ ; } s_1]\!] = \overline{\mathcal{A}}[\![s_1]\!] \circ \overline{\mathcal{A}}[\![s_0]\!]$$

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

$$\overline{\mathcal{A}}[\![\texttt{while } e \texttt{ do } s]\!] = \mathrm{lfp}\,\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a} \dot{\sqcup} \overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\,\overline{a})$$

$$\overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!] = \lambda\overline{a}.\prod_{k\in\mathbb{K}}\begin{cases}\pi_k(\overline{\mathcal{A}}[\![s]\!]\overline{a}) & \text{if } k \models \theta\\ \pi_k(\overline{a}) & \text{if } k \not\models \theta\end{cases}$$

$$\overline{\mathcal{A}'}[\![n]\!] = \lambda\overline{a}.\prod_{k\in\mathbb{K}} abst_{\mathbb{Z}}(n)$$

$$\overline{\mathcal{A}'}[\![\texttt{x}]\!] = \lambda\overline{a}.\prod_{k\in\mathbb{K}}\pi_k(\overline{a})(\texttt{x})$$

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

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

The partial ordering $\sqsubseteq_I$ induces a least upper bound, $\sqcup_I$, and a greatest lower bound operator, $\sqcap_I$. The function $abst_{\mathbb{Z}} : \mathbb{Z} \to \mathit{Interval}$ for turning values to properties is defined as:

$$abst_{\mathbb{Z}}(\texttt{n}) = [\texttt{n}, \texttt{n}] \tag{2}$$

Thus, the lifted transfer function for constants n becomes:

$$\overline{\mathcal{A}'}[\![\texttt{n}]\!] = \lambda\overline{a}.\prod_{k\in\mathbb{K}}[\texttt{n}, \texttt{n}]$$

For each binary operator $\oplus$, we have the operator $\widehat{\oplus}$ defined on properties from *Interval* [CC77, NNH99]:

$$[l_1, h_1]\widehat{\oplus}[l_2, h_2] = [\min_{\substack{x\in[l_1,h_1]\\ y\in[l_2,h_2]}}\{x \oplus y\}, \max_{\substack{x\in[l_1,h_1]\\ y\in[l_2,h_2]}}\{x \oplus y\}] \tag{3}$$

Thus, for the addition and subtraction operators, we have:

$$[l_1, h_1]\,\widehat{+}\,[l_2, h_2] = [l_1 + l_2, h_1 + h_2], \quad \text{and} \quad [l_1, h_1]\,\widehat{-}\,[l_2, h_2] = [l_1 - h_2, h_1 - l_2]$$

For example, $[2,2]\,\widehat{+}\,[1,2] = [3,4]$ and $[2,2]\,\widehat{-}\,[1,2] = [0,1]$, etc.

Example 4.1. Consider the $\overline{\text{IMP}}$ program P':

$$\texttt{x := 0; \#if } (A) \texttt{ x := x+2; \#if } (\neg A) \texttt{ x := x-2;}$$

where $\mathbb{K}_{[\![\text{true}]\!]} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. We write s_1 for the first statement x:=0; s_2 for #if (A) x := x+2, and s_3 for #if $(\neg A)$ x := x-2. We apply the lifted interval analysis on the family P'. We use an initial lifted store where x is uninitialized, i.e. it has the value $\top = [-\infty, +\infty]$. We write $\overline{a_0} \overset{\overline{\mathcal{A}}[\![s]\!]}{\longmapsto} \overline{a_1}$ when $\overline{\mathcal{A}}[\![s]\!]\overline{a_0} = \overline{a_1}$.

$$\big(\overbrace{[\texttt{x}\mapsto\top]}^{A\wedge B},\overbrace{[\texttt{x}\mapsto\top]}^{A\wedge\neg B},\overbrace{[\texttt{x}\mapsto\top]}^{\neg A\wedge B},\overbrace{[\texttt{x}\mapsto\top]}^{\neg A\wedge\neg B}\big)$$

$$\overset{\overline{\mathcal{A}}[\![s_1]\!]}{\longmapsto}\big([\texttt{x}\mapsto[0,0]],[\texttt{x}\mapsto[0,0]],[\texttt{x}\mapsto[0,0]],[\texttt{x}\mapsto[0,0]]\big)$$

$$\overset{\overline{\mathcal{A}}[\![s_2]\!]}{\longmapsto}\big([\texttt{x}\mapsto[2,2]],[\texttt{x}\mapsto[2,2]],[\texttt{x}\mapsto[0,0]],[\texttt{x}\mapsto[0,0]]\big)$$

$$\overset{\overline{\mathcal{A}}[\![s_3]\!]}{\longmapsto}\big([\texttt{x}\mapsto[2,2]],[\texttt{x}\mapsto[2,2]],[\texttt{x}\mapsto[-2,-2]],[\texttt{x}\mapsto[-2,-2]]\big)$$

When we analyze an `#if` with the presence condition A (resp. $\neg A$), the input store is updated only for those components $A \wedge B$, $A \wedge \neg B$ (resp., $\neg A \wedge B$, $\neg A \wedge \neg B$) which satisfy that presence condition by the effect of analyzing the statement associated with the given `#if`. □

Example 4.2. Consider the following $\overline{\text{IMP}}$ program P'' with nested `#if`-s:

$$\texttt{\#if } (A) \ \{\texttt{if (y>0) then \#if } (B) \texttt{ x++ else \#if } (\neg B) \texttt{ x--}\}$$

where we use $\mathbb{K}_{[\![\text{true}]\!]}$ and an initial lifted store with $\texttt{x} \mapsto [0,0]$ and $\texttt{y} \mapsto \top$ for all configurations. The effect of the outer annotation "`#if`(A)" is that only analysis results of configurations $A \wedge B$ and $A \wedge \neg B$ will be updated based on the analysis of the "`if`$(\texttt{y} > 0)$" statement. For "`#if` (B) `x++`" the analysis result with respect to $A \wedge B$ and $A \wedge \neg B$ will be $\texttt{x} \mapsto [1,1]$ and $\texttt{x} \mapsto [0,0]$ respectively, whereas for "`#if` $(\neg B)$ `x--`" the analysis result with respect to $A \wedge B$ and $A \wedge \neg B$ will be $\texttt{x} \mapsto [0,0]$ and $\texttt{x} \mapsto [-1,-1]$ respectively. Their least upper bound will give the result for the "`if`$(\texttt{y} > 0)$" statement. Thus, the final analysis result of the above program P'' is:

$$\big(\overbrace{[\texttt{x} \mapsto [0,1]]}^{A \wedge B}, \overbrace{[\texttt{x} \mapsto [-1,0]]}^{A \wedge \neg B}, \overbrace{[\texttt{x} \mapsto [0,0]]}^{\neg A \wedge B}, \overbrace{[\texttt{x} \mapsto [0,0]]}^{\neg A \wedge \neg B}\big) \qquad \square$$

5. Parametric Abstract Lifted Analysis

In this section, we first recall the calculus of variability abstractions defined in [DBW15] for reducing the configuration space. Then, we present the induced abstract variability-aware (lifted) analysis [DBW15], whose transfer functions are parametric in the choice of abstraction. A parametric lifted analysis analyzes a program family in a standard way except that it requires an abstraction to be provided as parameter before the analysis starts.

5.1. Variability Abstractions

We shall now define abstractions for reducing the lifted analysis domain $\mathbb{A}^{\mathbb{K}}$. The set *Abs* of well-formed abstractions is generated by the following grammar [DBW15]:

$$\alpha \ ::= \ \boldsymbol{\alpha}^{\text{id}} \mid \boldsymbol{\alpha}^{\text{join}} \mid \boldsymbol{\alpha}_{\varphi}^{\text{proj}} \mid \boldsymbol{\alpha}_{A}^{\text{fignore}} \mid \alpha \circ \alpha$$

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

The $\boldsymbol{\alpha}^{\text{id}}$ represents an identity function on $\mathbb{K}$ and $\overline{a} \in \mathbb{A}^{\mathbb{K}}$. Hence, we have $\boldsymbol{\alpha}^{\text{id}}(\mathbb{K}) = \mathbb{K}$. We also have a pair of abstraction and concretization functions: $\boldsymbol{\alpha}^{\text{id}}(\overline{a}) = \overline{a}$ and $\boldsymbol{\gamma}^{\text{id}}(\overline{a}) = \overline{a}$, which forms a Galois connection.

The *join* abstraction, $\boldsymbol{\alpha}^{\text{join}}$, gathers (joins) the information about all configurations $k \in \mathbb{K}$ into one (over-approximated) value of $\mathbb{A}$. After applying the $\boldsymbol{\alpha}^{\text{join}}$, we obtain only one valid abstract configuration, that is $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\bigvee_{k \in \mathbb{K}} k\}$. Note that this means that the obtained abstract domain is effectively $\mathbb{A}^1$. The abstraction and concretization functions between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\{\bigvee_{k \in \mathbb{K}} k\}}$, which form a Galois connection [DBW15], are:

$$\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = \left(\bigsqcup_{k \in \mathbb{K}} \pi_k(\overline{a})\right), \quad \boldsymbol{\gamma}^{\text{join}}(a) = \prod_{k \in \mathbb{K}} a$$

where $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $a \in \mathbb{A}$.

The *projection* abstraction, $\boldsymbol{\alpha}_{\varphi}^{\text{proj}}$, where $\varphi \in \mathit{FeatExp}(\mathbb{F})$, preserves only the values corresponding to configurations from $\mathbb{K}$ that satisfy φ. The information about configurations violating φ is disregarded. The set of abstract configurations is $\boldsymbol{\alpha}_{\varphi}^{\text{proj}}(\mathbb{K}) = \{k \in \mathbb{K} \mid k \models \varphi\}$. The abstraction and concretization functions between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\{k \in \mathbb{K} \mid k \models \varphi\}}$, which form a Galois connection [DBW15], are:

$$\boldsymbol{\alpha}_{\varphi}^{\text{proj}}(\overline{a}) = \prod_{k \in \mathbb{K}, k \models \varphi} \pi_k(\overline{a}), \quad \boldsymbol{\gamma}_{\varphi}^{\text{proj}}(\overline{a}') = \prod_{k \in \mathbb{K}} \begin{cases} \pi_k(\overline{a}') & \text{if } k \models \varphi \\ \top & \text{if } k \not\models \varphi \end{cases}$$

The abstraction $\boldsymbol{\alpha}_{A}^{\text{fignore}}$ 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 $\varphi \in FeatExp(\mathbb{F})$, we write $\varphi\backslash_A$ for a formula obtained by eliminating the feature A from φ in the following way. First, we convert φ into NNF (negation normal form), which contains only $\neg$, $\wedge$, $\vee$ connectives and $\neg$ appears only in literals. Then, $\varphi\backslash_A$ is the formula φ where literals A and $\neg A$ are replaced with true. Note that valuation formulas $k \in \mathbb{K}$ are already in NNF. For each formula $k' \equiv k\backslash_A$ where $k \in \mathbb{K}$, there will be one configuration in $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})$ determined by the formula $\bigvee_{k\in\mathbb{K},k\backslash_A\equiv k'} k$. Therefore, the set of abstract configurations is $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) = \{\bigvee_{k\in\mathbb{K},k\backslash_A\equiv k'} k \mid k' \in \{k\backslash_A \mid k \in \mathbb{K}\}\}$. The abstraction and concretization functions between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})}$, which form a Galois connection [DBW15], are:

$$\boldsymbol{\alpha}_A^{\text{fignore}}(\overline{a}) = \prod_{k'\in\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})} \bigsqcup_{k\in\mathbb{K},k\models k'} \pi_k(\overline{a}), \quad \boldsymbol{\gamma}_A^{\text{ignore}}(\overline{a}') = \prod_{k\in\mathbb{K}} \pi_{k'}(\overline{a}') \text{ if } k \models k'$$

We can similarly define $\boldsymbol{\alpha}^{\text{fignore}}_{\{A_1,\ldots,A_k\}}$ and $\boldsymbol{\gamma}^{\text{ignore}}_{\{A_1,\ldots,A_k\}}$, which ignore a set of features $\{A_1,\ldots,A_k\}$.

We also have *sequential composition* $\alpha_2 \circ \alpha_1$, which will run two abstractions α_1 and α_2 in sequence. Given Galois connections $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq}\rangle$ and $\langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq}\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_2(\alpha_1(\mathbb{K}))}, \dot{\sqsubseteq}\rangle$, we define their *sequential composition* $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\alpha_2\circ\alpha_1}{\overset{\gamma_1\circ\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_2\circ\alpha_1)(\mathbb{K})}, \dot{\sqsubseteq}\rangle$ as follows. The abstract set of configurations is $(\alpha_2 \circ \alpha_1)(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$, and we have:

$$(\alpha_2 \circ \alpha_1)(\overline{a}) = \alpha_2(\alpha_1(\overline{a})), \qquad (\gamma_1 \circ \gamma_2)(\overline{a}') = \gamma_1(\gamma_2(\overline{a}'))$$

for $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $\overline{a}' \in \mathbb{A}^{(\alpha_2\circ\alpha_1)(\mathbb{K})}$.

Remark. Note that, $\boldsymbol{\alpha}^{\text{join}}$ can be derived using $\boldsymbol{\alpha}_A^{\text{fignore}}$ and $\alpha_2 \circ \alpha_1$ by ignoring all features from $\mathbb{F}$ one by one. However, we keep $\boldsymbol{\alpha}^{\text{join}}$ in our set of abstractions, since its direct implementation as syntactic transformation of program families is much more efficient (see [DBW18, Sect.6]).

From now on, we will simply write $(\alpha, \gamma) \in Abs$ for any Galois connection $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}^{\alpha(\mathbb{K})}, \dot{\sqsubseteq}\rangle$, which is constructed using the operators presented in this section.

Example 5.1. Suppose that we have the following lifted store

$$\overline{a} = (\overbrace{[\mathtt{x}\mapsto[2,2]]}^{A\wedge B}, \overbrace{[\mathtt{x}\mapsto[2,2]]}^{A\wedge\neg B}, \overbrace{[\mathtt{x}\mapsto[0,0]]}^{\neg A\wedge B}, \overbrace{[\mathtt{x}\mapsto[\mathtt{-2},\mathtt{-2}]]}^{\neg A\wedge\neg B})$$

We have $\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{A\wedge\neg B}(\overline{a}) \sqcup \pi_{\neg A\wedge B}(\overline{a}) \sqcup \pi_{\neg A\wedge\neg B}(\overline{a})) = (\overbrace{[\mathtt{x}\mapsto[\mathtt{-2},2]]}^{\text{true}})$. Thus, the state is significantly decreased to only one component, but the abstraction $\boldsymbol{\alpha}^{\text{join}}$ losses precision by saying that $\mathtt{x}$ can have any value between -2 and 2.

Next, we have $\boldsymbol{\alpha}_A^{\text{proj}}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}), \pi_{A\wedge\neg B}(\overline{a})) = (\overbrace{[\mathtt{x}\mapsto[2,2]]}^{A\wedge B}, \overbrace{[\mathtt{x}\mapsto[2,2]]}^{A\wedge\neg B})$. Now the state is decreased to two components corresponding to configurations that satisfy A. We can also calculate $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_A^{\text{proj}}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{A\wedge\neg B}(\overline{a})) = (\overbrace{[\mathtt{x}\mapsto[2,2]]}^{A})$.

On the other hand, we have $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) = \{(A\wedge B)\vee(\neg A\wedge B) \equiv B, (A\wedge\neg B)\vee(\neg A\wedge\neg B) \equiv \neg B\}$, and so $\boldsymbol{\alpha}_A^{\text{fignore}}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{\neg A\wedge B}(\overline{a}), \pi_{A\wedge\neg B}(\overline{a}) \sqcup \pi_{\neg A\wedge\neg B}(\overline{a})) = (\overbrace{[\mathtt{x}\mapsto[0,2]]}^{B}, \overbrace{[\mathtt{x}\mapsto[\mathtt{-2},2]]}^{\neg B})$. Similarly, we can obtain $\boldsymbol{\alpha}_B^{\text{fignore}}(\overline{a}) = (\pi_{A\wedge B}(\overline{a}) \sqcup \pi_{A\wedge\neg B}(\overline{a}), \pi_{\neg A\wedge B}(\overline{a}) \sqcup \pi_{\neg A\wedge\neg B}(\overline{a})) = (\overbrace{[\mathtt{x}\mapsto[2,2]]}^{A}, \overbrace{[\mathtt{x}\mapsto[\mathtt{-2},0]]}^{\neg A})$. We can check that $\boldsymbol{\gamma}_A^{\text{fignore}}([\mathtt{x}\mapsto[0,2]], [\mathtt{x}\mapsto[\mathtt{-2},2]]) = ([\mathtt{x}\mapsto[0,2]], [\mathtt{x}\mapsto[\mathtt{-2},2]], [\mathtt{x}\mapsto[0,2]], [\mathtt{x}\mapsto[\mathtt{-2},2]]) \dot{\sqsupseteq} \overline{a}$, while $\boldsymbol{\gamma}_B^{\text{fignore}}([\mathtt{x}\mapsto[2,2]], [\mathtt{x}\mapsto[\mathtt{-2},0]]) = ([\mathtt{x}\mapsto[2,2]], [\mathtt{x}\mapsto[2,2]], [\mathtt{x}\mapsto[\mathtt{-2},0]], [\mathtt{x}\mapsto[\mathtt{-2},0]]) \dot{\sqsupseteq} \overline{a}$. □

$$\overline{\mathcal{A}}_\alpha[\![\texttt{skip}]\!] = \lambda\overline{a}.\,\overline{a}$$

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

$$\overline{\mathcal{A}}_\alpha[\![s_0 \texttt{ ; } s_1]\!] = \overline{\mathcal{A}}_\alpha[\![s_1]\!] \circ \overline{\mathcal{A}}_\alpha[\![s_0]\!]$$

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

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

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

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

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

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

Fig. 2. Definitions of abstract lifted transfer functions $\overline{\mathcal{A}}_\alpha[\![\overline{s}]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K})}$ and $\overline{\mathcal{A}'}_\alpha[\![\overline{e}]\!] : (\mathbb{A} \to \mathbb{P})^{\alpha(\mathbb{K})}$.

5.2. Induced Abstract Lifted Analysis

As we showed in Section 4.2, a lifted analysis 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 $(\alpha, \gamma) \in Abs$, the abstract lifted analyses induced by (α, γ) has been derived algorithmically in [DBW15], using the calculational approach to abstract interpretation [Cou99], which advocates simple algebraic manipulation to obtain a *direct expression* for the abstract transfer functions. 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 $\alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma$.

Figure 2 shows the definitions of $\overline{\mathcal{A}}_\alpha[\![s]\!]$ and $\overline{\mathcal{A}'}_\alpha[\![e]\!]$. They work over abstract lifted stores $\overline{a} \in \mathbb{A}^{\alpha(\mathbb{K})}$. Note that full lifted analysis $\overline{\mathcal{A}'}[\![e]\!]$ and $\overline{\mathcal{A}}[\![s]\!]$ coincide with $\overline{\mathcal{A}'}_{\boldsymbol{\alpha}^{\mathrm{id}}}[\![e]\!]$ and $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\mathrm{id}}}[\![s]\!]$ since $\boldsymbol{\alpha}^{\mathrm{id}}$ and $\boldsymbol{\gamma}^{\mathrm{id}}$ are identity functions. Observe also that the definitions of $\overline{\mathcal{A}}_\alpha[\![s]\!]$ and $\overline{\mathcal{A}'}_\alpha[\![e]\!]$ are identical to the definitions of $\overline{\mathcal{A}}[\![s]\!]$ and $\overline{\mathcal{A}'}[\![e]\!]$ except for the case of the preprocessor statement "`#if` (θ) s". This is expected since variability abstractions only affect the variability-specific aspect of the lifted analysis. This observation is the basis for defining abstractions as source-to-source transformations [DBW15]. The analysis of "`#if` (θ) s" 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 cases: (1) if $k' \models \theta$, the component k' of the input store is updated by the effect of evaluating the statement s; (2) if $k' \models \neg\theta$, the component k' of the store is not updated; (3) if $(k' \wedge \theta)$ and $(k' \wedge \neg\theta)$ are both satisfiable, denoted $\mathrm{sat}(k'\wedge\theta)\wedge\mathrm{sat}(k'\wedge\neg\theta)$, then the component k' is updated by the least upper bound of its initial value and the effect of s.

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 [DBW15].

Example 5.2. Reconsider the $\overline{\mathrm{IMP}}$ program P' from Example 4.1, where s_1 stands for the first statement `x:=0`; s_2 for `#if` (A) `x := x+2`, and s_3 for `#if` $(\neg A)$ `x := x-2`.

For $\boldsymbol{\alpha}^{\mathrm{join}}$, $\boldsymbol{\alpha}^{\mathrm{join}}(\mathbb{K}) = \{(A \wedge B) \vee (A \wedge \neg B) \vee (\neg A \wedge B) \vee (\neg A \wedge \neg B) \equiv \mathrm{true}\}$, thus:

$$\big(\overbrace{[\texttt{x} \mapsto \top]}^{\mathrm{true}}\big) \overset{\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\mathrm{join}}}[\![s_1]\!]}{\longmapsto} \big([\texttt{x} \mapsto [0,0]]\big) \overset{\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\mathrm{join}}}[\![s_2]\!]}{\longmapsto} \big([\texttt{x} \mapsto [0,2]]\big) \overset{\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\mathrm{join}}}[\![s_3]\!]}{\longmapsto} \big([\texttt{x} \mapsto [-2,2]]\big)$$

Note that both $(\mathrm{true} \wedge A)$ and $(\mathrm{true} \wedge \neg A)$ are satisfiable, thus for $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\mathrm{join}}}[\![s_2]\!]$ and $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\mathrm{join}}}[\![s_3]\!]$ we take the least upper bound of the input store and the effects of s_2 and s_3, respectively.

For $\alpha_1 = \boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}$, there is only one abstract configuration $k' = (A \wedge B) \vee (A \wedge \neg B) \equiv A$, so:

$$\big(\overbrace{[\mathtt{x} \mapsto \top]}^{A}\big) \overset{\overline{\mathcal{A}}_{\alpha_1}[\![s_1]\!]}{\longmapsto} \big([\mathtt{x} \mapsto [0,0]]\big) \overset{\overline{\mathcal{A}}_{\alpha_1}[\![s_2]\!]}{\longmapsto} \big([\mathtt{x} \mapsto [2,2]]\big) \overset{\overline{\mathcal{A}}_{\alpha_1}[\![s_3]\!]}{\longmapsto} \big([\mathtt{x} \mapsto [2,2]]\big)$$

Note that $k' \models \neg(\neg A)$, so in the last step $\overline{\mathcal{A}}_{\alpha_1}[\![s_3]\!]$ is an identity function. Thus, we can conclude that `x` is in the range $[2,2]$ for configurations that satisfy A after analyzing P'.

For $\alpha_2 = \boldsymbol{\alpha}^{\text{fignore}}_{B}$, there are two abstract configurations: $k'_1 = (A \wedge B) \vee (A \wedge \neg B) \equiv A$ and $k'_2 = (\neg A \wedge B) \vee (\neg A \wedge \neg B) \equiv \neg A$, so

$$\big(\overbrace{[\mathtt{x} \mapsto \top]}^{A}, \overbrace{[\mathtt{x} \mapsto \top]}^{\neg A}\big) \overset{\overline{\mathcal{A}}_{\alpha_2}[\![s_1]\!]}{\longmapsto} \big([\mathtt{x} \mapsto [0,0]], [\mathtt{x} \mapsto [0,0]]\big) \overset{\overline{\mathcal{A}}_{\alpha_2}[\![s_2]\!]}{\longmapsto}$$
$$\big([\mathtt{x} \mapsto [2,2]], [\mathtt{x} \mapsto [0,0]]\big) \overset{\overline{\mathcal{A}}_{\alpha_2}[\![s_3]\!]}{\longmapsto} \big([\mathtt{x} \mapsto [2,2]], [\mathtt{x} \mapsto [-2,-2]]\big)$$

For calculating $\overline{\mathcal{A}}_{\alpha_2}[\![\#\mathsf{if}\ (A)\ \mathtt{x:=x+2}]\!]$ we use the fact that $k'_1 \models A$ and $k'_2 \models \neg A$; and for $\overline{\mathcal{A}}_{\alpha_2}[\![\#\mathsf{if}\ (\neg A)\ \mathtt{x:=x-2}]\!]$ we use the fact that $k'_1 \models \neg(\neg A)$ and $k'_2 \models \neg A$. Thus, if we ignore the feature B, we can see that `x` is non-negative for configurations that satisfy A, and `x` is negative for configurations that satisfy $\neg A$. □

6. Abstractions Designed by Pre-analysis

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 $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}}$, $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$. The goal is to automatically construct a good abstraction α out of operators in the calculus *Abs*, such that α balances the precision and cost of the induced abstract variability-aware analysis. In this section, we first present how to design a pre-analysis which estimates when and where the full variability-aware main analysis is likely to have an accurate result. Then, we describe how using the pre-analysis results we can construct an appropriate abstraction α for the lifted main analysis that balances the precision and computational cost.

6.1. Definition of Pre-Analysis

We now define the pre-analysis which aims at estimating maximal precision of the lifted main analysis. The pre-analysis is induced from a suitable abstract property domain $\langle \mathbb{P}^{\#}, \sqsubseteq_{\mathbb{P}^{\#}} \rangle$, which represents an abstraction of the property domain $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$ from the main analysis. The pre-analysis is fully variability-aware and is specified by the following (abstract) domains: $\langle \mathbb{A}^{\#} = Var \to \mathbb{P}^{\#}, \sqsubseteq \rangle$, $\langle \mathbb{A}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$; and transfer functions: $\overline{\mathcal{A}'^{\#}}[\![e]\!] : (\mathbb{A}^{\#} \to \mathbb{P}^{\#})^{\mathbb{K}}$, $\overline{\mathcal{A}^{\#}}[\![s]\!] : (\mathbb{A}^{\#} \to \mathbb{A}^{\#})^{\mathbb{K}}$. There are two requirements that any designed pre-analysis should fulfill: *soundness* and *computational efficiency*. We now explain these two requirements separately.

Soundness. The pre-analysis should be designed to run with full variability-awareness but with a simpler abstract domain and simpler abstract transfer functions than those of the main analysis. This represents the *soundness condition* for our pre-analysis, which is specified not over the concrete semantics of the language but over the semantics of lifted main analysis (that is, over $\overline{\mathcal{A}'^{\#}}[\![e]\!]$ and $\overline{\mathcal{A}^{\#}}[\![s]\!]$).

The domains $\mathbb{P}^{\#}$, $\mathbb{A}^{\#}$, and $\mathbb{A}^{\#\,\mathbb{K}}$ of the pre-analysis should be sound with respect to those of the full variability-aware main analysis. 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 $\gamma^{\#}(a^{\#}) = \lambda x.\widehat{\gamma}^{\#}(a^{\#}(x))$, where $a \in A$ and $a^{\#} \in \mathbb{A}^{\#}$. 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 $\overline{\gamma}^{\#}(\overline{a^{\#}}) = \prod_{k \in \mathbb{K}} \gamma^{\#}(\pi_k(\overline{a^{\#}}))$, where $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $\overline{a^{\#}} \in \mathbb{A}^{\#\,\mathbb{K}}$. Similarly, by configuration-wise lifting we can construct the Galois connection

$\langle \mathbb{P}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\overline{\widehat{\alpha}}^{\#}}{\overset{\overline{\widehat{\gamma}}^{\#}}{\leftrightarrows}} \langle \mathbb{P}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$ by $\overline{\widehat{\alpha}}^{\#}(\overline{p}) = \prod_{k\in\mathbb{K}} \widehat{\alpha}^{\#}(\pi_k(\overline{p}))$ and $\overline{\widehat{\gamma}}^{\#}(\overline{p^{\#}}) = \prod_{k\in\mathbb{K}} \widehat{\gamma}^{\#}(\pi_k(\overline{p^{\#}}))$, where $\overline{p} \in \mathbb{P}^{\mathbb{K}}$ and $\overline{p^{\#}} \in \mathbb{P}^{\#\,\mathbb{K}}$.

Then, the transfer functions for expressions $\overline{\mathcal{A}'^{\#}}[\![e]\!]$ and statements $\overline{\mathcal{A}^{\#}}[\![s]\!]$ of the pre-analysis should be sound with respect to those of the full variability-aware main analysis:

$$\overline{\widehat{\alpha}}^{\#} \circ \overline{\mathcal{A}'}[\![e]\!] \circ \overline{\gamma}^{\#} \dot{\sqsubseteq} \overline{\mathcal{A}'^{\#}}[\![e]\!], \quad \overline{\alpha}^{\#} \circ \overline{\mathcal{A}}[\![s]\!] \circ \overline{\gamma}^{\#} \dot{\sqsubseteq} \overline{\mathcal{A}^{\#}}[\![s]\!] \tag{4}$$

for any $e \in Exp, s \in Stm$. In this way, we ensure that pre-analysis results over-approximate those of the full 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 is estimating the computationally expensive main analysis, still remains computable. We achieve *computational efficiency* of the pre-analysis by choosing a 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:

$$\widehat{\alpha}^{\#}(p) = \begin{cases} \bigstar & \text{if } p \in P \\ \top_{\mathbb{P}^{\#}} & \text{otherwise} \end{cases}, \quad \widehat{\gamma}^{\#}(\bigstar) = \bigsqcup P, \text{ and } \widehat{\gamma}^{\#}(\top_{\mathbb{P}^{\#}}) = \top_{\mathbb{P}} \tag{5}$$

The only non-trivial abstract property 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 full 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* [BRT+13], 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, where $\bigstar$ denotes all non-negative intervals, is executed as:

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

Initially, *all* configurations, $[\![true]\!] = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$, may be shared as they all have equivalent analysis information, $[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]$, associated with them. Thus, the initial lifted store $([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top])$ is represented as $([\![true]\!] \mapsto [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top])$. While we analyze the first three statements, all configurations still share the same store: $[\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top])$. After the variability statement in line 4, "`if (¬A) x:=x-2`", the four configurations get split into those for which A is enabled, $[\![A]\!] = \{A \wedge B, A \wedge \neg B\}$, that are mapped to $[\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top]$; and those for which A is disabled, $[\![\neg A]\!] = \{\neg A \wedge B, \neg A \wedge \neg B\}$ that are mapped to $[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]$. Now, the resulting lifted store $\big([\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]\big)$ is represented as $\big([\![A]\!] \mapsto [\mathtt{x} \mapsto \bigstar, \mathtt{y} \mapsto \top], [\![\neg A]\!] \mapsto [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top]\big)$. We still have some sharing; two equivalence classes of analysis information each with two configurations in them. In general, sharing is initially optimal, and then as the flow of control passes `#if` statements the configuration space is slowly split up into more and more equivalence classes. However, since the abstract domain $\mathbb{P}^{\#}$ of our pre-analysis is very small (there are only 2 values), the possibilities for sharing are very promising.

Discussion. Note that choosing a pre-analysis that under-approximates the main analysis would be too optimistic since the pre-analysis result would contain more $\bigstar$-s than the over-approximated pre-analysis and the resulting abstract lifted analysis would not be computationally cost effective. On the other hand, using an over-approximating pre-analysis allows us to construct a coarser abstraction where maximal number of configurations and features are abstracted (ignored).

Interval pre-analysis We now illustrate how design a pre-analysis for the interval analysis 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.

Let $\mathit{Interval}^{\#}$ be $\{\bigstar, \top\}$, where $\bigstar \sqsubseteq \top$. We define $\widehat{\gamma}^{\#}(\bigstar) = [0, +\infty]$ and $\widehat{\gamma}^{\#}(\top) = [-\infty, +\infty]$, which means that $\bigstar$ denotes all non-negative intervals. We then define $\mathbb{A}^{\#} : \mathit{Var} \to \mathit{Interval}^{\#}$ and $\mathbb{A}^{\#\,\mathbb{K}} = \prod_{k\in\mathbb{K}} \mathbb{A}^{\#}$. We can derive algorithmically the transfer functions for expressions by following the soundness 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:

$$\overline{\mathcal{A}'^{\#}}[\![n]\!] = \lambda\overline{a^{\#}}.(\texttt{if } n \geq 0 \texttt{ then } \overline{\bigstar} \texttt{ else } \overline{\top})$$
$$\overline{\mathcal{A}'^{\#}}[\![e_1 \oplus e_2]\!] = \lambda\overline{a^{\#}}. \prod_{k\in\mathbb{K}} \pi_k(\overline{\mathcal{A}'^{\#}}[\![e_1]\!]\overline{a^{\#}}) \;\sqcup\; \pi_k(\overline{\mathcal{A}'^{\#}}[\![e_2]\!]\overline{a^{\#}}), \text{ for } \oplus \in \{+, *, /\}$$
$$\overline{\mathcal{A}'^{\#}}[\![e_1 - e_2]\!] = \lambda\overline{a^{\#}}.\overline{\top}$$

where $\overline{\bigstar} = (\bigstar, \ldots, \bigstar), \overline{\top} = (\top, \ldots, \top) \in \mathit{Interval}^{\#\,\mathbb{K}}$. The analysis approximately tracks constants n. Non-negative values get abstracted to $\bigstar$, whereas negative values to $\top$. The binary operators "+", "*" and "/" are interpreted as the least upper bound $\sqcup$, so that for a configuration $k \in \mathbb{K}$, $e_1 + e_2$ (also, $e_1 * e_2$ and 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 information, since we do not know which one of the operands is bigger. Notice that $\overline{\mathcal{A}^{\#}}[\![s]\!]$ are identical to those of $\overline{\mathcal{A}}[\![s]\!]$ in Fig. 1. Also note that since the pre-analysis works on a lattice with finite height 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 so it needs widening operators. Such designed pre-analysis satisfies the required *soundness* and *computational efficiency* conditions.

Example 6.1. Reconsider the $\overline{\text{IMP}}$ program P' from Example 4.1. If we run the above designed pre-analysis on it, we obtain:

$$\big(\overbrace{[\mathtt{x}\mapsto\top]}^{A\wedge B},\overbrace{[\mathtt{x}\mapsto\top]}^{A\wedge\neg B},\overbrace{[\mathtt{x}\mapsto\top]}^{\neg A\wedge B},\overbrace{[\mathtt{x}\mapsto\top]}^{\neg A\wedge\neg B}\big) \quad \begin{array}{l} \overset{\overline{\mathcal{A}'^{\#}}[\![s_1]\!]}{\longmapsto}\big([\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar]\big) \\ \overset{\overline{\mathcal{A}'^{\#}}[\![s_2]\!]}{\longmapsto}\big([\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar]\big) \\ \overset{\overline{\mathcal{A}'^{\#}}[\![s_3]\!]}{\longmapsto}\big([\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\bigstar],[\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top]\big) \end{array}$$

If we use the optimized version of lifted analysis with sharing, we obtain the following calculations:

$$\big([\![\text{true}]\!] \mapsto [\mathtt{x}\mapsto\top]\big) \quad \begin{array}{l} \overset{\overline{\mathcal{A}'^{\#}}[\![s_1]\!]}{\longmapsto}\big([\![\text{true}]\!] \mapsto [\mathtt{x}\mapsto\bigstar]\big) \\ \overset{\overline{\mathcal{A}'^{\#}}[\![s_2]\!]}{\longmapsto}\big([\![\text{true}]\!] \mapsto [\mathtt{x}\mapsto\bigstar]\big) \\ \overset{\overline{\mathcal{A}'^{\#}}[\![s_3]\!]}{\longmapsto}\big([\![A]\!] \mapsto [\mathtt{x}\mapsto\bigstar],[\![\neg A]\!] \mapsto [\mathtt{x}\mapsto\top]\big) \end{array}$$

□

6.2. Constructing Abstractions

The pre-analysis results are used 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 as well as where variability-awareness should be turned on or off.

We first need to find whether a query can benefit from increased variability-awareness. Proving the correctness of queries or showing that they are likely to be violated is the goal of the analysis. For simplicity, to keep our presentation focussed 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 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 where all variables are set to $\top_{\mathbb{P}^{\#}}$, that

$$\overline{\mathcal{A}_\alpha}[\![\texttt{skip}]\!]^F = \lambda\overline{a^\#}.\,\overline{a^\#}$$

$$\overline{\mathcal{A}^\#}[\![\texttt{x := } e]\!]^F = \lambda\overline{a^\#}.\prod_{k\in\mathbb{K}}\pi_k(\overline{a^\#})[\texttt{x}\mapsto\left(\pi_k(\overline{\mathcal{A}'^\#}[\![e]\!]^F\overline{a^\#})|_1, F\cup\pi_k(\overline{\mathcal{A}'^\#}[\![e]\!]^F\overline{a^\#})|_2\right)]$$

$$\overline{\mathcal{A}^\#}[\![s_0\ \texttt{;}\ s_1]\!]^F = \lambda\overline{a^\#}.\overline{\mathcal{A}^\#}[\![s_1]\!]^F(\overline{\mathcal{A}^\#}[\![s_0]\!]^F\overline{a^\#})$$

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

$$\overline{\mathcal{A}^\#}[\![\texttt{while } e \texttt{ do } s]\!]^F = \mathrm{lfp}\lambda\overline{\Phi}.\,\lambda\overline{a^\#}.\,\overline{a^\#}\,\dot\sqcup\,\overline{\Phi}(\overline{\mathcal{A}^\#}[\![s]\!]^F\,\overline{a^\#})$$

$$\overline{\mathcal{A}^\#}[\![\texttt{\#if } (\theta)\ s]\!]^F = \lambda\overline{a^\#}.\prod_{k\in\mathbb{K}}\begin{cases}\pi_k(\overline{\mathcal{A}^\#}[\![s]\!]^{F\cup FV(\theta)}\overline{a^\#}) & \text{if } k\models\theta\\ \pi_k(\overline{a^\#}) & \text{if } k\not\models\theta\end{cases}$$

$$\overline{\mathcal{A}'}[\![n]\!]^F = \lambda\overline{a^\#}.\prod_{k'\in\mathbb{K}} abst_{\mathbb{Z}}(\mathtt{n})$$

$$\overline{\mathcal{A}'^\#}[\![\mathtt{x}]\!]^F = \lambda\overline{a^\#}.\prod_{k'\in\mathbb{K}}\pi_{k'}(\overline{a^\#})(\mathtt{x})$$

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

Fig. 3. Definitions for pre-analysis lifted transfer functions $\overline{\mathcal{A}^\#}[\![s]\!]^F : (\mathbb{A}^\# \to \mathbb{A}^\#)^{\alpha(\mathbb{K})}$ and $\overline{\mathcal{A}'^\#}[\![e]\!]^F : (\mathbb{A}^\# \to \mathbb{P}'^\#)^{\alpha(\mathbb{K})}$.

is $\overline{a_0^\#} = \prod_{k\in\mathbb{K}}(\lambda x.\top_{\mathbb{P}^\#})$. Using this result, we check if there is some $k \in \mathbb{K}$ such that:

$$\left(\pi_k(\overline{\mathcal{A}^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x}))\right) =_{\mathbb{P}} \bigstar \tag{6}$$

We select a query as worth being checked using the subsequent lifted analysis only if the pre-analysis result does not lose too much information with respect to that query.

Let $\mathbb{K}_{promise} \subseteq \mathbb{K}$ represent the set of all *promising configurations* $k \in \mathbb{K}$ that satisfy Eqn. (6) for a given selected query. Then, we compute the set $\mathbb{F}_{good} \subseteq \mathbb{F}$ of *necessary features* for a given query via dependency analysis, which is simultaneously done during the run of 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 $\mathbb{A}^\# = Var \to \mathbb{P}'^\#$, and $\mathbb{A}^{\#\,\mathbb{K}} = \prod_{k\in\mathbb{K}}\mathbb{A}^\#$. 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 $F \subseteq \mathbb{F}$. Their definitions are given in Fig. 3. The parameter $F \subseteq \mathbb{F}$ is propagated for all sub-statements of statements. For the `#if` statement, we also propagate the set of features occurring in θ, denoted as $FV(\theta)$, for each configuration k that satisfies θ, since the analysis result for those configurations will depend on features in θ. The most interesting case is the assignment "`x := e`", when it is recorded explicitly in the analysis which features have contributed for calculating the given property of `x`.

We compute $\mathbb{F}_{good}$ as the union of all sets of features F, such that for some $k \in \mathbb{K}_{promise}$ we obtain: $\pi_k(\overline{\mathcal{A}^\#}[\![s]\!]^\emptyset\overline{a^\#}(\mathtt{x})) = (\bigstar, F)$. Then, we set $\mathbb{F}_{ignore} = \mathbb{F}\backslash\mathbb{F}_{good}$. The final constructed abstraction is: $\boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}_{ignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\wedge_{k\in\mathbb{K}_{promise}}k}$, and the corresponding concretization is: $\boldsymbol{\gamma}^{\text{proj}}_{\wedge_{k\in\mathbb{K}_{promise}}k} \circ \boldsymbol{\gamma}^{\text{fignore}}_{\mathbb{F}_{ignore}}$.

Example 6.2. If we calculate $\overline{\mathcal{A}^\#}[\![P]\!]^\emptyset\overline{a_0^\#}$, where P is our motivating example from Section 2 and $\overline{a_0^\#} = \prod_{k\in\mathbb{K}}(\lambda x.(\top,\emptyset))$ is the initial uninitialized lifted store, we will obtain the output store:

$$\big([\mathtt{x}\mapsto(\bigstar,\{A\}),\mathtt{y}\mapsto(\top,\{B\})],[\mathtt{x}\mapsto(\bigstar,\{A\}),\mathtt{y}\mapsto(\top,\{B\})],$$
$$[\mathtt{x}\mapsto(\top,\{A\}),\mathtt{y}\mapsto(\top,\{B\})],[\mathtt{x}\mapsto(\top,\{A\}),\mathtt{y}\mapsto(\top,\{B\})]\big)$$

Therefore, we select the first query that asks for non-negative values of x as worth being subsequently analysed 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: $\boldsymbol{\alpha}_B^{\text{fignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{(A\wedge B)\vee(A\wedge\neg B)}$. On the other hand, the second query that asks for non-negative values of y is rejected, since y is mapped to $\top$ for all configurations. □

Example 6.3. Reconsider the $\overline{\text{IMP}}$ program P' from Example 4.1. We can calculate $\overline{\mathcal{A}^\#}[\![P]\!]^{\emptyset}\overline{a_0^\#}$ as follows:

$$\big(\overbrace{[\mathtt{x}\mapsto(\top,\emptyset)]}^{A\wedge B},\overbrace{[\mathtt{x}\mapsto(\top,\emptyset)]}^{A\wedge\neg B},\overbrace{[\mathtt{x}\mapsto(\top,\emptyset)]}^{\neg A\wedge B},\overbrace{[\mathtt{x}\mapsto(\top,\emptyset)]}^{\neg A\wedge\neg B}\big) \overset{\overline{\mathcal{A}^\#}[\![s_1]\!]}{\longmapsto} \big([\mathtt{x}\mapsto(\bigstar,\emptyset)],[\mathtt{x}\mapsto(\bigstar,\emptyset)],[\mathtt{x}\mapsto(\bigstar,\emptyset)],[\mathtt{x}\mapsto(\bigstar,\emptyset)]\big)$$
$$\overset{\overline{\mathcal{A}^\#}[\![s_2]\!]}{\longmapsto}\big([\mathtt{x}\mapsto(\bigstar,\{A\})],[\mathtt{x}\mapsto(\bigstar,\{A\})],[\mathtt{x}\mapsto(\bigstar,\emptyset)],[\mathtt{x}\mapsto(\bigstar,\emptyset)]\big)$$
$$\overset{\overline{\mathcal{A}^\#}[\![s_3]\!]}{\longmapsto}\big([\mathtt{x}\mapsto(\bigstar,\{A\})],[\mathtt{x}\mapsto(\bigstar,\{A\})],[\mathtt{x}\mapsto(\top,\{A\})],[\mathtt{x}\mapsto(\top,\{A\})]\big)$$

We obtain $\mathbb{K}_{promise} = \{A \wedge B, A \wedge \neg B\}$, $\mathbb{F}_{good} = \{A\}$, and $\mathbb{F}_{ignore} = \{B\}$. The constructed abstraction is $\boldsymbol{\alpha}_B^{\text{fignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{(A\wedge B)\vee(A\wedge\neg B)}$. □

We now give an example when our pre-analysis guided abstraction loses precision compared to the full lifted analysis due to the over-approximation introduced in the pre-analysis.

Example 6.4. Consider the following $\overline{\text{IMP}}$ program P'':

x := 10; #if (A) x := x-2;

where $\mathbb{F} = \{A\}$ and $\mathbb{K} = \{A, \neg A\}$. When P'' is analyzed with the full lifted interval analysis, we obtain the output lifted store $\big(\overbrace{[\mathtt{x}\mapsto[8,8]]}^{A},\overbrace{[\mathtt{x}\mapsto[10,10]]}^{\neg A}\big)$. So, we can establish that x is non-negative after analyzing P'' in both configurations A and $\neg A$.

However, if we use our pre-analysis guided abstraction then we lose precision for the configuration A. First, the full lifted interval pre-analysis after analyzing P'' will compute the store $\big([\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\bigstar]\big)$, since $\overline{\mathcal{A}'^\#}[\![\mathtt{x}-2]\!]([\mathtt{x}\mapsto\bigstar]) = \top$. Thus, we obtain the abstraction $\boldsymbol{\alpha}^{\text{proj}}_{\neg A}$, and $\overline{\mathcal{A}}_{\boldsymbol{\alpha}^{\text{proj}}_{\neg A}}[\![P'']\!]([\mathtt{x}\mapsto\top]) = [\mathtt{x}\mapsto[10,10]]$. In this way, we show that x is non-negative after P'' only for the configuration $\neg A$, but we miss the opportunity to show that x is non-negative for the configuration A as well. □

Finally, we can show that the constructed abstract variability-aware analysis reports precise results for the given query.

Theorem 6.1 (Promising Preservation). Let $\mathbb{F}_{ignore}$ and $\mathbb{K}_{promise}$ be the sets of ignored features and good 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 $\alpha = \boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}_{ignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\wedge_{k\in\mathbb{K}_{promise}}k}$ and $\gamma = \boldsymbol{\gamma}^{\text{proj}}_{\wedge_{k\in\mathbb{K}_{promise}}k} \circ \boldsymbol{\gamma}^{\text{fignore}}_{\mathbb{F}_{ignore}}$. Then:

$$\gamma\big(\overline{\mathcal{A}}_\alpha[\![s]\!]\overline{a_0}(\mathtt{x})\big) \dot{\sqsubseteq} \overline{\widehat{\gamma}}^\#\big(\overline{\mathcal{A}^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)$$

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.

Proof. We prove this theorem in two steps. We first show that $\overline{\mathcal{A}^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x}) = \gamma\big(\overline{\mathcal{A}_\alpha^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)$ (Lemma 6.1). Then, we prove that $\overline{\mathcal{A}_\alpha}[\![s]\!]\overline{\gamma}^\#(\overline{a_0^\#})(\mathtt{x}) \dot{\sqsubseteq} \overline{\gamma}^\#\big(\overline{\mathcal{A}_\alpha^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)$ (Lemma 6.2). Using these two lemmas, we have:

$$\begin{aligned}
&\overline{\widehat{\gamma}}^\#\big(\overline{\mathcal{A}^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)\\
&= \overline{\widehat{\gamma}}^\#\big(\gamma\big(\overline{\mathcal{A}_\alpha^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)\big) && \text{(by Lemma 6.1)}\\
&= \gamma\big(\widehat{\gamma}^\#\big(\overline{\mathcal{A}_\alpha^\#}[\![s]\!]\overline{a_0^\#}(\mathtt{x})\big)\big) && \text{(by def. of } \gamma \text{ and } \overline{\gamma}^\#)\\
&\dot{\sqsupseteq} \gamma\big(\overline{\mathcal{A}_\alpha}[\![s]\!]\overline{\gamma}^\#(\overline{a_0^\#})(\mathtt{x})\big) && \text{(by Lemma 6.2)}
\end{aligned}$$

□

Lemma 6.1. $\overline{\mathcal{A}^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x}) = \gamma\big(\overline{\mathcal{A}_\alpha^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})\big)$.

Proof. Let $\overline{a^{\#}} = \overline{\mathcal{A}^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})$. Using the lifted store $\overline{a^{\#}}$, we construct $\alpha = \boldsymbol{\alpha}^{\text{fignore}}_{\mathbb{F}_{ignore}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\wedge_{k\in\mathbb{K}_{promise}}k}$ and $\gamma = \boldsymbol{\gamma}^{\text{proj}}_{\wedge_{k\in\mathbb{K}_{promise}}k} \circ \boldsymbol{\gamma}^{\text{fignore}}_{\mathbb{F}_{ignore}}$. Then, we have $\alpha(\overline{a^{\#}}) = \prod_{k\in\alpha(\mathbb{K})} \bigstar$, $\overline{\mathcal{A}_\alpha^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x}) = \prod_{k\in\alpha(\mathbb{K})} \bigstar$, and $\gamma(\alpha(\overline{a^{\#}})) = \overline{a^{\#}}$ by the definition of α and γ. Thus, we have:

$$
\begin{aligned}
&\gamma\big(\overline{\mathcal{A}_\alpha^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})\big)\\
&= \gamma\Big(\prod_{k\in\alpha(\mathbb{K})} \bigstar\Big) = \gamma(\alpha(\overline{a^{\#}})) = \overline{a^{\#}} && \text{(by def. of } \alpha \text{ and } \gamma)\\
&= \overline{\mathcal{A}^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})
\end{aligned}
$$

□

Lemma 6.2. $\overline{\mathcal{A}_\alpha}[\![s]\!]\overline{\gamma}^{\#}(\overline{a_0^{\#}})(\mathtt{x}) \mathrel{\dot{\sqsubseteq}} \overline{\widehat{\gamma}}^{\#}\big(\overline{\mathcal{A}_\alpha^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})\big)$.

Proof. By induction on the structure of $\alpha \in Abs$ and $s \in Stm$. Note that $\overline{a_0}(\mathtt{x}) = \overline{\widehat{\gamma}}^{\#}(\overline{a_0^{\#}}(\mathtt{x}))$. Apart from the `#if`-statement, for all other statements the proof is an immediate result of definitions of $\overline{\mathcal{A}}_\alpha$, $\overline{\mathcal{A}_\alpha^{\#}}$, and the soundness result in Eqn. (4). By definition of Galois connections, we have $\overline{\mathcal{A}_\alpha}[\![s]\!]\overline{\gamma}^{\#}(\overline{a_0^{\#}})(\mathtt{x}) \mathrel{\dot{\sqsubseteq}} \overline{\widehat{\gamma}}^{\#}\big(\overline{\mathcal{A}_\alpha^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})\big)$ iff $\overline{\widehat{\alpha}}^{\#}\big(\overline{\mathcal{A}_\alpha}[\![s]\!]\overline{\gamma}^{\#}(\overline{a_0^{\#}})(\mathtt{x})\big) \mathrel{\dot{\sqsubseteq}} \overline{\mathcal{A}_\alpha^{\#}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x})$.

We show the most illustrative case $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}$ for `#if` (θ) s.

$$
\begin{aligned}
&\overline{\mathcal{A}^{\#}_{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}}[\![\texttt{\#if } (\theta) \ s]\!]\overline{a_0^{\#}}(\mathtt{x})\\
&= \prod_{k\in\{k\in\mathbb{K}\mid k\models\varphi\}} \begin{cases} \overline{\mathcal{A}^{\#}_{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}}[\![s]\!]\overline{a_0^{\#}}(\mathtt{x}) & \text{if } k \models \theta \\ \overline{a_0^{\#}}(\mathtt{x}) & \text{if } k \models \neg\theta \end{cases} && \text{(by def. of } \overline{\mathcal{A}^{\#}_{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}}\text{)}\\
&\mathrel{\dot{\sqsupseteq}} \prod_{k\in\{k\in\mathbb{K}\mid k\models\varphi\}} \begin{cases} \overline{\widehat{\alpha}}^{\#}\big(\overline{\mathcal{A}_{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}}[\![s]\!]\overline{a_0}(\mathtt{x})\big) & \text{if } k \models \theta \\ \overline{\widehat{\alpha}}^{\#}\big(\overline{a_0}(\mathtt{x})\big) & \text{if } k \models \neg\theta \end{cases} && \text{(by IH)}\\
&= \overline{\widehat{\alpha}}^{\#}\big(\overline{\mathcal{A}_{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}}[\![\texttt{\#if } (\theta) \ s]\!]\overline{a_0}(\mathtt{x})\big) && \text{(by def. of } \overline{\mathcal{A}_{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}}\text{)}
\end{aligned}
$$

The other cases are similar. □

7. Lifting as Binary Decision Diagram

There are two ways to speed up analyses: by increasing abstraction and by improving representation. The former has received considerable attention in the previous section, we now investigate the latter. Recall that the lifted analysis domain $\mathbb{A}^{lifted}$ is defined as *cartesian product* of the analysis domain $\mathbb{A}$ for the single-program analysis we want to lift, i.e. we have $\mathbb{A}^{lifted} = \mathbb{A}^{\mathbb{K}}$. Thus, no interaction (sharing) is explicitly possible between analysis results (properties) corresponding to different configurations. The key for enabling interaction and sharing is the proper handling of disjunctions of properties.

We now propose a new, more general definition of the lifted analysis domain as a *binary decision diagram* (BDD) domain functor, which can express disjunctive properties depending on the values of available features with sharing at the leaves [CC15, CCM10]. In this way, it enables the interaction (sharing) between analyses corresponding to different configurations. Available features $\mathbb{F}$ are organized in decision nodes of a BDD, where each top-down path (without leaf) represents one or several configurations, and in each leaf node is stored the analysis property corresponding to the given configurations. The BDD domain is parametric in the choice of the abstract domain $\mathbb{A}$ for the leaf nodes.

A *binary decision tree* $t \in \mathbb{T}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ over the set $\mathbb{F}$ of features, the set $\mathbb{K}$ of valid configurations, and the leaf abstract domain $\mathbb{A}$ is either a leaf node $a \in \mathbb{A}$ and $\mathbb{F} = \mathbb{K} = \emptyset$, or $[\![A : tl, tr]\!]$, where A is the feature which

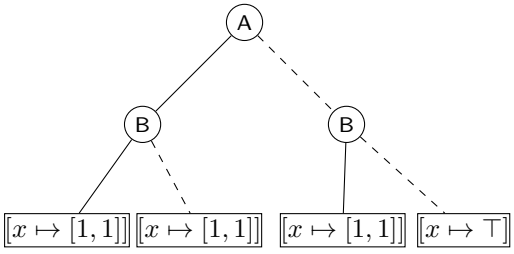


Fig. 4. A binary decision tree.

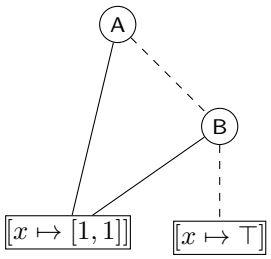


Fig. 5. A binary decision diagram.

occurs earliest in a fixed enumeration of all possible features in $\mathbb{F}$, tl is the left subtree of t representing its true branch, and tr is the right subtree of t representing its false branch, such that $tl, tr \in \mathbb{T}(\mathbb{F}\backslash\{A\}, \mathbb{K}_{\backslash\{A\}}, \mathbb{A})$. Here, $\mathbb{K}_{\backslash\{A\}}$ denotes the removal of A or $\neg A$ from each configuration of $\mathbb{K}$. However, there are several optimizations (removal of leaves and non-leaves, removal of redundant tests) which can be applied to binary decision trees in order to reduce (compress) their representation [Bry86]. If all possible reductions are applied to a binary decision tree $t \in \mathbb{T}(\mathbb{F}, \mathbb{K}, \mathbb{A})$, then the result is a reduced *binary decision diagram* $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. Moreover, if the ordering on the Boolean variables from $\mathbb{F}$ occurring on any path is fixed (the same) for all BDDs, then the resulting BDDs have a *canonical form*.

Example 7.1. Let $\mathbb{F}$ be $\{A, B\}$ and let $\mathbb{A}$ be the $Interval$ lattice. Consider the binary decision tree t shown in Fig. 4, where the edges are labeled with the truth value of the decision on the parent node, true or false (we use solid edges for true, and dashed edges for false). In first order logic, the tree t expresses the formula:

$$(A \wedge B \wedge [x \mapsto [1,1]]) \vee \big((\neg A \wedge B \wedge [x \mapsto [1,1]]) \vee (A \wedge \neg B \wedge [x \mapsto [1,1]]) \vee (\neg A \wedge \neg B \wedge [x \mapsto \top])$$

The reduced BDD obtained from t is shown in Fig. 5.

On the other hand, if we use the standard tuple-based lifted analysis domain defined as cartesian product of $\mathbb{K}$ copies of $\mathbb{A}$, the above analysis element is represented as:

$$\big(\overbrace{[\mathtt{x} \mapsto [1,1]]}^{A \wedge B}, \overbrace{[\mathtt{x} \mapsto [1,1]]}^{A \wedge \neg B}, \overbrace{[\mathtt{x} \mapsto [1,1]]}^{\neg A \wedge B}, \overbrace{[\mathtt{x} \mapsto \top]}^{\neg A \wedge \neg B}\big)$$

We can see that the BDD-based representation uses only two leaf nodes, while the tuple-based representation uses four. □

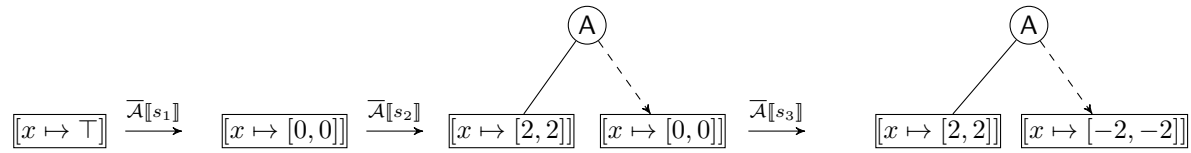
The lifted domain $\mathbb{A}^{lifted}$ is defined as a binary decision diagram domain functor: $(\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A}), \sqsubseteq_{\mathbb{D}}, \sqcup_{\mathbb{D}}, \sqcap_{\mathbb{D}}, \bot_{\mathbb{D}}, \top_{\mathbb{D}})$. The definitions of the lattice operators of $\mathbb{T}(\mathbb{F}, \mathbb{A})$ as well as transfer functions are similar to the ones given in [CC15, CCM10].

We consider the inclusion operator $\sqsubseteq_{\mathbb{T}}$. Given $d_1, d_2 \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$, we can check $d_1 \sqsubseteq_{\mathbb{D}} d_2$ by comparing each pair (a_1, a_2) of leaves in (d_1, d_2) where a_1 and a_2 are defined by the same configuration path $k : \mathbb{F} \to \{\text{true}, \text{false}\}$. If each pair (a_1, a_2) satisfies $a_1 \sqsubseteq_{\mathbb{A}} a_2$ then $d_1 \sqsubseteq_{\mathbb{D}} d_2$; otherwise $d_1 \not\sqsubseteq_{\mathbb{D}} d_2$. Given $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$, the analysis of the assignment "$\mathtt{x:=}e$" is performed at each leaf a in d by using the assignment transfer function of $\mathbb{A}$. That is, the leaf a becomes $\mathcal{A}[\![\mathtt{x:=}e]\!]a$.

Consider the case of the preprocessor statement "$\mathtt{\#if}\,(\theta)\,s$". Let $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. For each leaf a in d, let $k : \mathbb{F} \to \{\text{true}, \text{false}\}$ be the configuration path leading to it. If $k \models \theta$ then the new updated leaf will be $\mathcal{A}[\![s]\!]a$. Otherwise, if $k \not\models \theta$ then the leaf a is not updated.

The operators of the binary decision diagram domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ and transfer functions are combined together to analyze program families. In the first iteration of the analysis, we build BDDs with only one leaf node that can be reached along only valid paths. For the first program point the leaf node is $\top_{\mathbb{A}}$, whereas for the other program points the leaf node is $\bot_{\mathbb{A}}$. The analysis properties are then propagated forward towards the final control point taking assignments and tests into account with join and widening around `while`-s.

Example 7.2. Reconsider the $\overline{\text{IMP}}$ program P' from Example 4.1. The lifted interval analysis of P' using the lifted domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, Interval)$ is:



□

8. Evaluation

We now evaluate our pre-analysis guided approach for finding suitable variability abstractions for lifted analysis. The evaluation aims to show the following objectives:

O1: We can find suitable abstract lifted analysis for which the precision-speed tradeoff is acceptable, i.e. the speed up of the found abstract lifted analysis compared to plain lifted analysis is significant while the precision loss is small.

O2: We can find practical application scenarios of using the found abstract lifted analysis to efficiently analyse program families.

8.1. Experimental setup

We have developed an implementation, which uses the SOOT's intra-procedural dataflow analysis framework [VRCG+99] for analyzing Java programs and an existing SOOT extension for lifted dataflow analyses of Java program families [BRT+13]. Although our approach is general and can be applied to any annotation-based program family, our implementation (that is, the lifted dataflow analysis framework) parses and performs analyses of Java program families where variability is implemented using graphical CIDE (Colored IDE) [Kas10].

Colored IDE. The CIDE is an Eclipse plug-in, which annotates variability in Java code using background colors rather than `#ifdef` directives. Every feature in a program family is thus associated with a unique color. The CIDE plug-in supports only disciplined `#ifdef`-s, which means that only optional code fragments can be annotated. For example, we can annotate an entire method, an entire statement, an entire class, etc. But, we cannot annotate arbitrary code fragments like isolated brackets. The CIDE also supports nested `#ifdef`-s, like the one in Example 4.2, which results in overlapping colors. In this case, CIDE displays only the color corresponding to the innermost annotation, and adds a left frame for each outer color annotation [Kas10]. Although, overlapping colors may make variability annotations difficult to read, still our lifted analyzer can easily parse them using the API's provided by the CIDE framework.

Experiments. 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 dataflow analyses to be performed. The implementation, benchmarks, and all results obtained from our experiments are available from: `https://aleksdimovski.github.io/pre-analysis.html`.

Client analysis. Our pre-analysis guided approach for lifted analysis is orthogonal to the particular analysis chosen as a client. While a single-program client analysis operates on states and depends on language-specific constructs, our approach based on lifting single-program analysis and variability abstractions depends on variability-specific constructs, such as: features, configurations, and `#ifdef`-s. Therefore, our approach can be applied to any programs (e.g. Java, C, IMP) for which a given single-program analysis exists and a static variability based on `#ifdef`-s can be added.

For our experiment, we have chosen the interval analysis as a client analysis. We have implemented the single-program versions of interval pre-analysis and interval analysis in the SOOT framework. For interval analysis, the so-called delayed widening is implemented using the $flowThrough$ method of $ForwardFlowAnalysis$ class by counting the number of times a node was visited and applying a widening operator once a threshold has been reached. The widening operator w works relatively to a fixed subset B of integers which includes 0, $-\infty$, $+\infty$ and all constants that occur in the method to be analyzed [CC77, NNH99]. We use the following

Benchmark	avg. $\lvert\mathbb{K}\rvert$	$\lvert\mathbb{F}\rvert$	LOC	#method	#method-int	#vars	#confs
`GPL`	N=3.9	18	1,350	135	19	33	242
`Prevayler`	N=1.3	5	8,000	779	130	174	226
`BerkeleyDB`	N=1.6	42	84,000	3608	1286	2654	7386

Table 2. Characteristics of our three benchmarks (average #configurations in all methods, total #features, LOC, total #methods, #methods with integer variables, and total #integer variables along with #configurations where they appear).

definition for widening: $w([l,h]) = [max\{i \in B \mid i \leq l\}, min\{i \in B \mid i \geq h\}]$. Then, on top of a lifted dataflow analyzer [BRT+13], we have implemented variability-aware versions of interval pre-analysis and interval analysis described in Section 6 and Section 4, respectively. The lifted pre-analysis, which is designed to work with respect to queries that ask for non-negative intervals of variables, 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 non-negative values for all integer (*int* and *long*) variables in given methods.

Types of lifted analysis. We consider here optimized versions of lifted analyses and abstract lifted analyses, which use improved representation via sharing of analysis equivalent configurations. More specifically, sets of configurations with equivalent analysis information are compactly represented as formulae as described in Section 6.1 (see the paragraph on Computational efficiency). We abbreviate them as $\overline{\mathcal{A}}_{sh}$ in the following. We will also consider $\overline{\mathcal{A}}_{bdd}$ which is lifted analysis that uses binary decision diagrams for lifted domains as described in Section 7.

Solution of an analysis. Each analysis uses a control flow graph (CFG) of an analyzed method, in which nodes correspond to program points and edges represent possible flow of control, and a lattice, which represents the analysis domain. The analysis then runs a fixed-point algorithm to compute the unique least solution which to every node in the CFG assigns an element from the analysis domain. We only assess the analysis elements assigned to the final nodes of CFGs (i.e. the exit of methods).

Benchmarks. We analyze three case studies written in CIDE [Kas10]. Graph PL (GPL) is a small application with high variability usage. It contains about 1,35 kLOC, 18 features, and 135 methods with 3.9 valid configurations per method on average. Prevayler is a slightly larger product line with low variability usage, which contains 8 kLOC, 5 features, and 779 methods with 1.3 configurations per method on average. BerkeleyDB is a large database library with moderate variability usage. It contains about 84 kLOC, 42 features, and 3608 methods with 1.6 valid configurations per method on average. Table 2 summarizes relevant characteristics for each benchmark: the average number of valid configurations in all methods in the benchmark, the total number of features in the entire benchmark, the total number of lines of code (LOC), the total number of methods, the number of methods that contain integer variables which will be analyzed, and the total number of integer variables to be analyzed along with the total number of configurations where they occur.

8.2. Precision-speed tradeoff

Figure 6 illustrates the tradeoff between precision and speed of our pre-analysis guided approach for lifted analysis. Figure 6a shows the performance of full lifted analysis which is used as a baseline, whereas Fig. 6b presents the performance of our approach based on pre-analysis followed by the corresponding abstract lifted analysis. We measured the analysis precision by counting the number of integer variables for which our approach accurately calculates their analysis information in the final nodes of CFGs (see full precision column, Table 6b). This is established by checking that the same analysis results for those variables are obtained with the full lifted interval analysis (see analysis results column, Fig. 6a). Note that the obtained analysis result for each variable can be either a specific non-negative interval or the coarsest $\top$ value. In Fig. 6, we report the number of variables in the final nodes of analysed methods that have a specific non-negative interval, denoted "var $[\,]$", and the $\top$ value, denoted "var $\top$". We report the number of configurations in which those precisely tracked variables occur, denoted "con $[\,]$" and "con $\top$". We also measured the number of variables

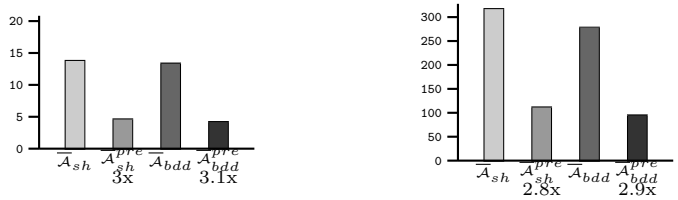
Benchmark	*Lifted analysis*					
	analysis results				Time	
	var []	con []	var ⊤	con ⊤	$\overline{\mathcal{A}}_{sh}$	$\overline{\mathcal{A}}_{bdd}$
GPL	33	216	18	26	73.1	69.5
Prevayler	56	58	166	168	83.2	81.1
BerkeleyDB	1144	3197	2139	4189	2908	2800

(a) Performance results for lifted (variability-aware) analysis which is used as a baseline.

Benchmark	*Pre-analysis guided approach*							
	full precision				prec. loss		Time	
	var []	con []	var ⊤	con ⊤	var⊤	con⊤	$\overline{\mathcal{A}}_{sh}$	$\overline{\mathcal{A}}_{bdd}$
GPL	33	216	18	26	0	0	33.4	25.8
Prevayler	56	58	166	168	0	0	62.3	54.0
BerkeleyDB	1141	3154	2137	4232	3	43	1933	1750

(b) Performance results for our pre-analysis guided approach which consists of running a pre-analysis followed by a subsequent abstract lifted analysis.

Fig. 6. Performance comparison for baseline lifted analysis vs. pre-analysis guided approach. All times are in ms (milliseconds).



GPL::main()
$\mathbb{K}$=8, #var = 5

BerkeleyDB::main()
$\mathbb{K}$=40, #var =3

Fig. 7. Performance results for selected methods: baseline lifted analysis ($\overline{\mathcal{A}}_{sh}$ and $\overline{\mathcal{A}}_{bdd}$) vs. pre-analysis guided abstract lifted analysis ($\overline{\mathcal{A}}^{pre}_{sh}$ and $\overline{\mathcal{A}}^{pre}_{bdd}$). All times are in ms.

and corresponding configurations where there is a precision loss (see precision loss column, Table 6b), i.e. our approach produces the ⊤ value but the full lifted 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 with $\overline{\mathcal{A}}_{sh}$ and 2.7 times with $\overline{\mathcal{A}}_{bdd}$, whereas for Prevayler we have 1.3 times speed-up with $\overline{\mathcal{A}}_{sh}$ and 1.5 times with $\overline{\mathcal{A}}_{bdd}$ (pre-analysis+abstract vs. lifted 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 3154+4189=7343 cases (configurations). Yet, we achieve 1.5 times speed-up using $\overline{\mathcal{A}}_{sh}$ and 1.6 times using $\overline{\mathcal{A}}_{bdd}$ with our pre-analysis guided approach for BerkeleyDB (Objective (**O1**)).

In Fig. 7, we show the performance results for selected methods, `GPL::main()` and `BerkeleyDB::main()`, in isolation. For each method, we show: the number of configurations $|\mathbb{K}|$, the number of analyzed integer variables $\#var$. We report the times needed for the baseline lifted analysis and the pre-analysis guided approach ($\overline{\mathcal{A}}_{sh}$ and $\overline{\mathcal{A}}_{bdd}$ versions). In these cases, we observe speed ups of more than 3 times with no precision loss (Objective (**O1**)).

8.3. Application scenario

Figure 8 shows a (slightly modified) fragment extracted from `GPL`'s `main()` method with N=8 configurations. First, local variables, `numEdges` and `j`, and an array, `startVertices`, are defined and initialized. Then,

```
void   main(..) {
1      .. int numEdges = 10, j=0;
2      int[] startVertices = new int[numEdges];
3      for (int i=0; i<numEdges; i++) {
4          #ifdef (Prog) startVertices[i]=i #endif
5          #ifdef (Transpose) j-- #endif
6      }
7      ...startVertices[j]=0;...
}
```

Fig. 8. Code extracted from `GPL::main()`.

the array `startVertices` is conditionally updated in a `for` loop, and in each iteration the local `j` is also conditionally decreased. We want to establish the range of possible values of `j` in line 7 in order to check if there is an array out-of-bounds access. The query we consider is: `j` is non-negative at line 7. The pre-analysis reports that `j` is mapped to ★ for all configurations that satisfy ¬`Transpose`. Moreover, the analysis result for `j` in the above configurations does not depend on any feature, so we obtain the abstraction $\boldsymbol{\alpha}_{\mathbb{F}}^{\text{fignore}} \circ \boldsymbol{\alpha}_{\neg\texttt{Transpose}}^{\text{proj}}$ (which is equivalent to $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_{\neg\texttt{Transpose}}^{\text{proj}}$). The subsequent abstract lifted analysis will report the interval $[0, 0]$ for `j` at line 7. That means that there is no array out-of-bounds error for configurations that satisfy ¬`Transpose` (Objective (**O2**)).

8.4. Discussion

We are now ready to confirm that our Objectives **O1** and **O2** are achieved. We can use our pre-analysis guided approach to construct abstract lifted analyses for which the precision-speed tradeoff is acceptable compared to the standard lifted analysis, i.e. the performance speed ups range from 1.3 to three times while the precision loss is very small (less than 1%) - Objective **O1**. The constructed optimal abstraction is also very useful in practice to efficiently verify various interesting program properties, such as array out-of-bounds access, buffer overflows, division by zero, etc - Objective **O2**.

Threats to validity. We perform intra-procedural interval analysis of relatively small methods. We have not evaluated our approach for larger program families using more complex inter-procedural analysis. However, the focus of our approach based on abstractions is to combat the configuration space explosion of program families, not their size. Therefore, we expect to obtain similar or even better results for larger programs and more complex client analysis.

9. Related Work

We divide our discussion of related work into four categories: finding a good analysis parameter, other related analyses based on abstract interpretation, lifted analysis, and other lifted techniques.

Finding a good analysis parameter Oh et. al. [OLH+14, OLH+16] have proposed the idea of adjusting the main analysis precision by using a pre-analysis. More specifically, they design pre-analysis for estimating the impact and finding the optimal values of several analysis parameters, such as: context sensitivity, flow sensitivity, and relational constraints between variables. In this work, we further investigate this technique and we apply it in the context of lifted analysis. In particular, we show how to construct a pre-analysis that estimates the impact of variability on analysis for program families. In future, it would be interesting to consider designing a pre-analysis that estimates the combination of several precision parameters (context, flow, relational constraints between variables, and variability) at the same time.

In general, there are many parameters to adjust in a dataflow analysis in order to improve either precision or scalability. The work [LTN11] uses machine learning to construct a minimal abstraction that is good enough to show all queries provable by the most precise abstraction. Similarly, the work [ZNY13] generates the optimum abstraction that is able to show the correctness of a given query in the context of disjunctive analysis. Finally, in [NYCS12], a dynamic analysis is used to select an appropriate parameter for a given query, which is guaranteed to be a necessary condition to prove the query.

Other related analyses based on abstract interpretation Rival and Mauborgne in [RM07] have proposed a generic framework for defining trace partitioning abstract domains, which have a wide range of instantiations. They allow partitioning of traces to be based on the history of control flow. Similarly, in this work we also propose a generic framework for defining abstract lifted domains, where partitioning on valid configurations is performed. Moreover, the suggested partitionings in both approaches can be handled by syntactic rewriting of the code.

The application of disjunctive abstract domains (e.g. our binary decision diagrams in Section 7) in static analysis have recently become very popular. A segmented decision tree abstract domain where disjunctions are determined by values of variables is proposed in [CCM10], whereas in [CC15] disjunctions are determined by the branch conditions. The `Function` analyzer [UM14] for proving program termination is also based on a decision tree abstract domain.

Dalla Preda et al. in [PGD15] use abstract interpretation to extract metamorphic signatures from metamorphic programs, which can change during execution. A metamorphic program applies semantics-preserving transformations to modify its own code so that one instance of the program (called variant) is syntactically different but semantically equivalent to another one. Such self-modifying programs are commonly encountered in malware. The work in [PGD15] provides the theoretical foundation for verifying whether one program is a variant of a metamorphic program. This is done by extracting an approximated representation of all its possible variants (abstract metamorphic signature). In this work, we also consider many variants extracted from a common code base. However, in contrast to [PGD15] where all variants are semantically equivalent, the generated variants here are distinct but still very similar. Therefore, we propose various variability abstractions which take into account the similarity between different variants and thus derive optimized, approximated static analyses for them.

Lifted analysis Brabrand et. al. [BRT+13] show how to lift a dataflow analysis from the monotone framework, resulting in a lifted dataflow analysis for program families. The obtained lifted dataflow analyses are much faster than ones based on the "brute force" strategy, which explicitly generates and analyzes all variants one by one, individually. SPL$^{\text{LIFT}}$ [BTR+13] represents an implementation of the lifted dataflow analysis formulated within the IFDS framework [RHS95]. It has been shown that the SPL$^{\text{LIFT}}$'s running time of analyzing all variants in a family is close to the analysis of a single program. However, this approach works only for analyses phrased within the IFDS framework [RHS95], a subset of dataflow analyses with certain properties, such as distributivity of transfer functions. Many dataflow analyses, including interval and sign detection analyses, are not distributive and cannot be encoded in IFDS. A formal methodology for systematic derivation of lifted static analyses from existing single-program analyses was proposed in [MDBW15]. The method uses the calculational approach to abstract interpretation of Cousot [Cou99] in order to derive a lifted analysis which is correct by construction. In [DBW15, DBW18], an expressive calculus of variability abstractions is devised for deriving abstract lifted analyses. Such variability abstractions enable deliberate trading of precision for speed in lifted analysis. However, in this approach we assume that a user has a good knowledge of the given program family and query, so he can manually devise suitable abstractions before analysis. In the present work, we pursue this line of work by devising an automatic technique for finding suitable variability abstractions given a lifted analysis, a program family, and a query to prove.

Other lifted techniques Recently, various approaches have been proposed for lifting existing analysis and verification techniques to work on the level of families, rather than on the level of single programs/systems (see [TAK+14] for a survey). Many of those approaches analyze entire families at once through sharing, by splitting where necessary and joining at fine granularity. Our lifted analysis on binary decision diagrams is an example of such analysis with sharing. Some other successful examples with sharing are dynamic analysis of program families based on variability-aware execution [MWK+16], and software model checking of program families based on variability encoding (to transform compile-time to run-time variability) and BDDs (to represent variability information in states) [vR16]. Another approaches [ILMD+17, IAD+15] for verifying C programs with `#ifdef`-s are also based on variability encoding and several off-the-shelf single-program verification tools. TypeChef [KGR+11] and SuperC [GG12] are variability-aware parsers, which can parse languages with preprocessor annotations thus producing ASTs with variability nodes. The difference between these two approaches is that feature expressions are represented as formulae in TypeChef, and as BDD's in SuperC. Several approaches have been proposed for type checking program families directly. In particular, lifted type checking for Featherweight Java was presented in [KA08], whereas variational lambda calculus was studied in [CEW12].

Lifted model checking has been an active research field in recent years, where many interesting approaches have been developed for verifying variational systems. One of the earliest attempts to model variational systems and perform lifted model checking is by using Modal Transition Systems (MTSs) [LNW07], where optional 'may' transitions are used to model variability. Then, Beek et al. [tBFGM16] have implemented a model checking tool, called VMC, for verifying variability models expressed as MTSs and properties expressed as v-ACTL formulae. Subsequently, various variability models have been developed. Ultimately, the popular Feature Transition Systems (FTSs) have been introduced by Classen et. al. [CCS$^+$13], which is widely accepted today as the model essentially sufficient for most purposes of lifted model checking. Classen et. al. [CCH$^+$12] have proposed specially designed lifted model checking algorithms for efficient verification of LTL properties of such systems, which are implemented in the SNIP model checker. The input language to this tool is fPromela, which is a feature-aware extension of the well-known SPIN's language Promela. Subsequently, this approach was extended to handle CTL properties [CHSL11], as well as different modelling formalisms such as probabilistic [CDKB18] and real-time systems [CSHL12]. In [DABW15, DABW17], we introduce variability abstractions in the context of lifted model checking, which are used for abstracting FTSs. This allows to efficiently verify some interesting LTL properties of variational systems by only a few calls to an off-the-shelf (single-system) model checker, such as SPIN. Variability abstractions which are sound with respect to the whole CTL* are defined in [Dim18a], whereas the application of variability abstractions for verifying real-time variational systems is described in [DW17a]. Similarly as in the present work where we propose an automated procedure for family-based static analysis, an automatic verification (abstraction refinement) procedure for lifted model checking has been proposed in [DW17b], which works until a genuine counterexample is found or the property satisfaction is shown for all variants in the family. In particular, the Craig interpolation is used in each iteration to extract the relevant information from a spurious counterexample in case of an imprecise answer. This information is then used to define the refinement for the next iteration. As opposed to the procedure proposed here which always runs in two iterations and may lose some precision, the procedure from [DW17b] may run in several iterations (in the worst-case, it will verify all variants in the brute-force fashion one by one) and always produces precise results for all variants.

In [Dim16, Dim18b], specifically designed family-based software model checking algorithms are used for verifying symbolic game semantics models [Dim14], which are extracted from open second-order programs with `#ifdef`-s that contain undefined components. Variability abstractions considered here can be also applied in this settings, in order to enable more scalable and efficient verification of program fragments.

In this work, we consider annotation-based program families where variability is integrated with the common code base. Apart from C-Preprocessors [KAK08] and graphical CIDE [Kas10], the choice calculus [EW11] represents another method to implement such annotation-based families. The choice calculus is a simple, formal language for representing variability in a way that maximizes sharing and minimizes redundancy, which is similar to the goals of binary decision diagram domain introduced in Section 7. The annotation-based families contrast sharply with composition-based program families [AK09], where features are implemented as separate and composable units. In this approach, features are developed and tested independently, and then combined in a prescribed manner to produce the desired set of variants. A well-recognized problem here are *feature interactions*, which are cases where composing several features alters the behaviour of one or several of them. While there are several approaches for detecting feature interactions [STAL11] as well as commutativity of features [CSDR18], there is a potential to apply variability abstractions in this context as one possible formal approach to address the above problem.

10. Conclusion

We present an automatic two-phase procedure for effective lifted analysis of program families. In the first phase, a specifically tailored pre-analysis is run to calculate suitable abstraction parameters. The pre-analysis aggressively abstracts the semantics aspects of the analysis that are not relevant for the queries we want to prove. In the second phase, the calculated abstract lifted analysis is used to show the analysis's queries. We demonstrate the effectiveness of our procedure with experiments by showing that it achieves a good balance between analysis precision and cost. On three Java SPL benchmarks, we saw analysis speed ups of between 1.3 and three times faster when first calculating an abstraction and then using it to analyze the programs. For the analyses and properties investigated (interval analysis and variables non-negative upon method exit), the increases in speed came at very small loss of precision. This demonstrates the overall feasibility of our approach.

We can extend our lifted analyzer with more complex (single-program) analysis domains, such as the most common numerical domains from the `APRON` library (octagons, polyhedra, etc) [JM09]. Those domains take into account relational constraints among program variables, thus making possible to infer more complex program properties. We can also devise a pre-analysis that estimates the impact of both variability and relational constraints among variables in order to make the resulting lifted analysis more scalable and efficient.

11. Notation and Symbols

To recap, we show here the notation and symbols used in this paper.

Symbols	Description
$\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$	the property domain (e.g. *Interval*)
$\langle \mathbb{A} = Var \to \mathbb{P}, \sqsubseteq \rangle$	the (single-program) analysis domain
$\mathcal{A}[\![s]\!] : \mathbb{A} \to \mathbb{A}$ $\mathcal{A}'[\![e]\!] : \mathbb{A} \to \mathbb{P}$	(single-program) transfer functions
$\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle$	the lifted analysis domain
$\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}}$	lifted transfer functions
$\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}^{\alpha(\mathbb{K})}, \dot{\sqsubseteq} \rangle$	variability abstractions
$\overline{\mathcal{A}}_\alpha[\![\bar{s}]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K})}$ $\overline{\mathcal{A}'}_\alpha[\![\bar{e}]\!] : (\mathbb{A} \to \mathbb{P})^{\alpha(\mathbb{K})}$	abstract lifted transfer functions
$\langle \mathbb{P}^\# = \{\bigstar, \top_{\mathbb{P}^\#}\}, \sqsubseteq_{\mathbb{P}^\#} \rangle$	the pre-analysis property domain
$\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle \underset{\widehat{\alpha}^\#}{\overset{\widehat{\gamma}^\#}{\leftrightarrows}} \langle \mathbb{P}^\#, \sqsubseteq_{\mathbb{P}^\#} \rangle$	abstractions for constructing $\mathbb{P}^\#$
$\langle \mathbb{A}^\# = Var \to \mathbb{P}^\#, \sqsubseteq \rangle$	the (single-program) pre-analysis domain
$\langle \mathbb{A}, \sqsubseteq \rangle \underset{\alpha^\#}{\overset{\gamma^\#}{\leftrightarrows}} \langle \mathbb{A}^\#, \sqsubseteq \rangle$	abstractions for constructing $\mathbb{A}^\#$
$\langle \mathbb{A}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$	the lifted pre-analysis domain
$\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\overline{\alpha}^\#}{\overset{\overline{\gamma}^\#}{\leftrightarrows}} \langle \mathbb{A}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$	abstractions for constructing $\mathbb{A}^{\mathbb{K}}$
$\langle \mathbb{P}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\overline{\overline{\alpha}}^\#}{\overset{\overline{\overline{\gamma}}^\#}{\leftrightarrows}} \langle \mathbb{P}^{\#\,\mathbb{K}}, \dot{\sqsubseteq} \rangle$	abstractions for constructing $\mathbb{P}^{\#\,\mathbb{K}}$
$\overline{\mathcal{A}^\#}[\![s]\!] : (\mathbb{A}^\# \to \mathbb{A}^\#)^{\alpha(\mathbb{K})}$ $\overline{\mathcal{A}'^\#}[\![e]\!] : (\mathbb{A}^\# \to \mathbb{P}^\#)^{\alpha(\mathbb{K})}$	pre-analysis lifted transfer functions
$\overline{\mathcal{A}^\#}[\![s]\!]^F : (\mathbb{A}^\# \to \mathbb{A}^\#)^{\alpha(\mathbb{K})}$ $\overline{\mathcal{A}'^\#}[\![e]\!]^F : (\mathbb{A}^\# \to \mathbb{P}'^\#)^{\alpha(\mathbb{K})}$	pre-analysis lifted transfer functions where $\mathbb{P}'^\# = \mathbb{P}^\# \times \mathcal{P}(\mathbb{F})$, $\mathbb{A}^\# = Var \to \mathbb{P}'^\#$

References

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Lines Conference, SPLC '05*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

[BRT+13] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BTR+13] 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.

[CC77] 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.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2–3):103–179, 1992.

[CC15] Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *LNCS*, pages 36–53. Springer, 2015.

[CCH+12] 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.

[CCM10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A scalable segmented decision tree abstract domain. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.

[CCS+13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013.

[CDKB18] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Asp. Comput.*, 30(1):45–75, 2018.

[CEW12] 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.

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*, pages 321–330, 2011.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[Cou99] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, pages 1–88. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[CSDR18] Marsha Chechik, Ioanna Stavropoulou, Cynthia Disenfeld, and Julia Rubin. FPH: efficient non-commutativity analysis of feature-based systems. In *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Proceedings.*, volume 10802 of *LNCS*, pages 319–336. Springer, 2018.

[CSHL12] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural modelling and verification of real-time software product lines. In *16th International Software Product Line Conference, SPLC '12, Volume 1*, pages 66–75. ACM, 2012.

[DABW15] 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.

[DABW17] Aleksandar Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Efficient family-based model checking via variability abstractions. *STTT*, 19(5):585–603, 2017.

[DBW15] 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.

[DBW16] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Finding suitable variability abstractions for family-based analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 217–234, 2016.

[DBW18] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions for lifted analysis. *Sci. Comput. Program.*, 159:1–27, 2018.

[Dim14] Aleksandar Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014.

[Dim16] 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.

[Dim18a] Aleksandar S. Dimovski. Abstract family-based model checking using modal featured transition systems: Preservation of ctl\(^{\star }\). In *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Proceedings.*, volume 10802 of *LNCS*, pages 301–318. Springer, 2018.

[Dim18b] Aleksandar S. Dimovski. Verifying annotated program families using symbolic game semantics. *Theor. Comput. Sci.*, 706:35–53, 2018.

[DW17a] Aleksandar S. Dimovski and Andrzej Wasowski. From transition systems to variability models and from lifted model checking back to UPPAAL. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, volume 10460 of *LNCS*, pages 249–268. Springer, 2017.

[DW17b] Aleksandar S. Dimovski and Andrzej Wasowski. Variability-specific abstraction refinement for family-based model checking. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings*, volume 10202 of *LNCS*, pages 406–423, 2017.

[EW11] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, December 2011.

[GG12] Paul Gazzillo and Robert Grimm. Superc: parsing all of C by taming the preprocessor. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 323–334. ACM, 2012.

[IAD+15] 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.

[ILMD+17] Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Effective analysis of c programs by rewriting variability. *Programming Journal*, 1(1):1, 2017.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[KA08] Christian Kästner and Sven Apel. Type-checking software product lines - A formal approach. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2008)*, pages 258–267, 2008.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 311–320, Leipzig, Germany, 2008. ACM.

[Kas10] Christian Kastner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.

[KGR+11] 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.

[LNW07] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O automata for interface and product line theories. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Proceedings*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[LTN11] 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.

[MDBW15] 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.

[MWK+16] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 483–494. ACM, 2016.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Secaucus, USA, 1999.

[NYCS12] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pages 373–386, 2012.

[OLH+14] 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.

[OLH+16] 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.

[PGD15] Mila Dalla Preda, Roberto Giacobazzi, and Saumya K. Debray. Unveiling metamorphism by abstract interpretation of code properties. *Theor. Comput. Sci.*, 577:74–97, 2015.

[RHS95] 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.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5):26, 2007.

[STAL11] Wolfgang Scholz, Thomas Thüm, Sven Apel, and Christian Lengauer. Automatic detection of feature interactions using the java modeling language: an experience report. In *Software Product Lines - 15th International Conference, SPLC 2011, Workshop Proceedings (Volume 2)*, page 7. ACM, 2011.

[TAK+14] 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):6:1–6:45, 2014.

[tBFGM16] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *J. Log. Algebr. Meth. Program.*, 85(2):287–315, 2016.

[UM14] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings*, volume 8723 of *LNCS*, pages 302–318. Springer, 2014.

[vR16] Alexander von Rhein. *Analysis strategies for configurable systems*. PhD thesis, University of Passau, Germany, 2016.

[VRCG+99] 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.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

[ZNY13] 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.