Variability abstractions for lifted analyses☆

Aleksandar S. Dimovski[a,b,*], Claus Brabrand[b], Andrzej Wasowski[b]

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

Abstract

Family-based (lifted) static analysis for "highly configurable programs" (program families) is capable of analyzing all variants at once without generating any of them explicitly. It takes as input only the common code base, which encodes all variants of a program family, and produces precise analysis results corresponding to all variants. However, the computational cost of the lifted analysis still depends inherently on the number of variants, which is in the worst case exponential in the number of statically configurable options (features). For a large number of features, the lifted analysis may be too costly or even infeasible. In this work, we introduce variability abstractions defined as Galois connections, which simplify variability away from program families based on `#ifdef`-s. Then, we use abstract interpretation as a formal method for the calculational-based derivation of abstracted lifted analyses, which are sound by construction.

Our approach for abstracting lifted analysis is orthogonal to the particular program analysis chosen as a client. While a single program analysis operates on program states and depends on language-specific constructs, the lifted analysis assumes that a single program analysis already exists and lifts its results to all variants of the analyzed program family. Variability abstractions aim to reduce this variability-specific component of the lifted analysis, which handles variability and `#ifdef`-s. Furthermore, given the "orthogonality" of variability abstractions to the rest of the analysis (its language-specific component), we can implement abstractions as a preprocessor. In particular, given an abstraction we define a syntactic transformation, which translates any program family into an abstracted version of it, such that the analysis of the abstracted program family coincides with the corresponding abstracted analysis of the original program family. We have implemented the proposed approach, and we evaluate its practicality on three Java benchmarks. The evaluation shows that abstractions yield significant performance gains, especially for families with higher variability.

Keywords: Program Families, Static Analysis, Abstract Interpretation

☆Partially supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE.
*Corresponding author

1. Introduction

Highly configurable (variable) software appears in many application areas and for many reasons. One common scenario is the development of software product lines (SPLs) [1], where features are used 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 (code reuse is the main motivation for software product line architectures). Software product lines are commonly seen in development of commercial embedded software (e.g., cars, phones, avionics). In this case, variation points are used to either support different application scenarios for embedded components, to provide software portability across different hardware platforms and configurations, or to produce variations of products for different market segments or different customers. However, highly configurable software is not limited to embedded software. Many off-the-shelf reusable components and software products allow or even require extensive customization. Most system level software is highly configurable (for instance the Linux kernel [2]), frameworks and development platforms also allow adding and removing many features (for instance, plugins provide features in Eclipse [3]), as well as many web-solutions are highly customizable (for instance Drupal or Wordpress [4]). While there are many implementation strategies, many popular industrial product lines are implemented using annotative approaches such as conditional compilation; in particular, via the C-preprocessor `#ifdef` construct [5].

Recently, formal analysis and verification of program families have been a topic of considerable research (see [6] for a survey). The challenge is to develop analysis and verification techniques that work at the level of program families, rather than the level of individual programs. Given that the number of variants can potentially grow exponentially with the number of features, the need for efficient analysis and verification techniques is essential. To address this, a number of so-called *lifted* techniques have emerged, essentially lifting existing analysis and verification techniques to work on program families, rather than on individual programs. This includes lifted type checking [7, 8], lifted data-flow analysis [9, 10, 11], lifted model checking [12, 13, 14, 15], lifted verification [16, 17, 18]. They are also known as family-based (*variability-aware*) techniques. Lifted techniques are capable of analyzing the entire code base (all variants at once), without having to explicitly generate and analyze all individual variants, one at a time. Also, lifted techniques are able to pinpoint errors directly in the program family, as opposed to reporting errors in individual variants derived from the family.

There are two ways to speed up analyses: improving *representation* and increasing *abstraction*. The former has received considerable attention in the field of lifted analysis [9, 10]. In this paper, we investigate the latter [19, 20, 21]. We consider a range of abstractions at the *variability level* that may tame the combinatorial explosion of the number of configurations (variants) and reduce it to something more tractable by manipulating the configuration space of a

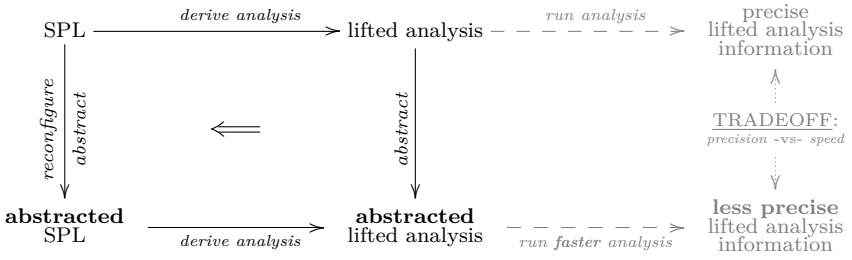

Figure 1: Diagram illustrating the role and intended usage of the `reconfigurator` transformation. Instead of abstracting an already existing (or derived) lifted analysis, our transformation allows abstraction to be applied directly to the SPL. The two paths from SPL to "abstracted lifted analysis" are guaranteed to produce the same analysis.

program. Such variability abstractions enable deliberate trading of precision for speed in lifted analyses, even turn infeasible analyses into feasible ones, while retaining an intimate relationship back to the original analysis (via the abstraction).

We organize our variability abstractions in a calculus that provides convenient and compositional declarative specification of abstractions. We propose two basic abstraction operators (*join* and *project*), two compositional abstraction operators (*sequential composition* and *parallel composition*), and two more derived abstractions (*join-project* and *feature ignore*). The variability abstractions may take arbitrary relationships among features into account. An abstraction based on the `join` may *join* together all variants of a family conservatively over-approximating the analysis information retained commonly for all of them. Another abstraction based on the `project` could *project* all variability onto certain set of configurations (e.g., configurations maintained by a certain developer or configurations with something in common). *Sequential composition* will run two abstractions, in sequence; whereas *product* will run both abstractions, in parallel. Several interesting abstractions can be derived from the above basic ones: `join-project` which applies join after projecting on a subset of configurations; and `feature ignore` which ignores a set of features that are deemed irrelevant.

Each abstraction expresses a compromise between precision and speed in the induced abstracted analysis. We show how to apply each of these abstractions to lifted static analyses, to derive their corresponding efficient and sound (correct) abstracted lifted analysis based on the calculational approach of abstract interpretation [22]. Thus, we obtain a set of abstracted lifted analysis parameterized by the choice of abstraction we use. Note that the approach is general and applicable to *any* static analysis phrased as an abstract interpretation.

We observe that for variability abstractions, *analysis abstraction* and *analysis derivation* commute. Figure 1 illustrates how analysis abstraction is classically undertaken and how we propose to optimize it. The top left corner shows a product line that we want to analyze. A lifted analyzer will take an SPL

(program family) as input and derive a "lifted analysis" (rightward arrow), which works on the level of SPLs. Examples of such derived (client) analyses are: constant propagation, reaching definitions, uninitialized variables, interval analysis [23]. We can then run that lifted analysis (next rightward dashed arrow) and obtain our "*precise* lifted analysis information" (i.e. precise analysis results for all variants in the SPL). Note that for some analyzers, the phases *derive analysis* and subsequent *run analysis* may be so intertwined that they are not independently distinguishable. Since running the analysis might be too slow or infeasible, we may decide to use abstraction to obtain a faster, although less precise analysis. Classically, an abstraction is applied to the derived lifted analysis before it is run (middle arrow down) which, after an often long and complex process, produces an "abstracted lifted analysis". When that analysis is subsequently run, it will produce less precise analysis information, but it will do so faster than the original analysis (i.e. there is a *precision vs. speed tradeoff*).

Interestingly, for lifted analyses and variability abstractions, the analysis abstraction (down) and derivation (right) commute and we may swap their order of application, as indicated by the short double leftward arrow in the center. The implications are quite significant. It means that variability abstractions can be applied *before*, and independently of, the subsequent analysis. This also means that the same variability abstractions might be applicable to all sorts of client static analyses that are specifiable via abstract interpretation.

We exploit this observation to define a *stand-alone* source-to-source transformation, called `reconfigurator`, for programs with `#ifdef`s. It takes an input SPL program and a variability abstraction that aims to reduce the configuration space, and it produces an abstracted SPL program for which the subsequent lifted analysis agrees with "abstracted lifted analysis" of the original unabstracted SPL. Like a preprocessor the `reconfigurator` is essentially unaware of the programming language syntax, thus it can be used for any client static analysis [23]. Many existing analysis methods that are unable to abstract variability benefit from this work instantly. Almost no extension or adaptation is required as the abstraction is applied to source code before analysis.

We evaluate our approach by comparing analyses of a range of increasingly abstracted SPLs against their origins without abstraction, quantifying to what extent precision can be traded for speed in lifted analyses.

In summary, the work makes the following contributions:

C1: A *calculus* for modular specification of variability abstractions;

C2: *Variability abstraction* as a method for trading precision for speed in lifted static analysis based on abstract interpretation;

C3: The observation that certain analysis derivations and analysis abstractions *commute*, meaning that variability abstractions can be applied directly on an SPL *before* (and independently of) subsequent lifted analysis;

C4: Generalization of the above methods to any static analysis from the abstract interpretation framework;

C5: A stand-alone *transformation* based on the above ideas;

C6: An *evaluation* of the tradeoff between precision and speed in abstracted lifted analyses.

We direct this work to program analysis and software engineering researchers. The method of *variability abstractions* (**C1**–**C4**) is directed at designers of lifted analyses for product lines. They may use our insights to design improved abstracted analyses that appropriately trade precision for speed. Note that the ideas apply beyond the context of static analyses (e.g., to model checking, type systems, verification, and testing). The `reconfigurator` (**C5**) and the evaluation lessons (**C6**) are relevant for software engineers working on preprocessor-based product lines and who would like to speed up existing analyzers.

This work represents an extended and revised version of [24]. Compared to the earlier work, we make the following extensions here: (1) We provide formal proofs for all main results; (2) We expand and elaborate the examples, and motivate the practicality of applying variability abstractions to lifted analysis; (3) We show how the source-to-source transformations can be also defined for derived abstractions (e.g. `feature ignore`); (4) We implement one more client analysis, that is interval analysis, and we provide more evaluation results; (5) We explain how the proposed method can be generalized and applied to *any* (monotone) static analysis based on abstract interpretation.

We proceed by giving a few example scenarios of applying variability abstractions to lifted analysis in Section 2. The basics of lifted analysis based on abstract interpretation is introduced in Section 3. Section 4 defines a calculus for specification of variability abstractions. Section 5 explains how to apply an abstraction to a lifted analysis. It uses constant propagation as an example. The `reconfigurator` is described in Section 6 along with its correctness for our example analysis. In Section 7, we show how our results can be generalized to any monotone static analysis phrased in the abstract interpretation framework. Section 8 presents the evaluation on three Java SPLs. Finally, we discuss the relation to other works and conclude.

2. Motivation: Application Scenarios for Variability Abstractions

One challenge in the development of configurable systems is analyzing all configurations. This challenge is amplified by increasing interoperability of systems, components and services in the Internet of Things [25]. It is usually infeasible to run an analyzer on all possible configurations due to exponentially many configurations. Researchers have addressed this problem by designing aggregate so-called lifted analyses, that simultaneously analyze all the variants at once [9, 10, 11]. The process of generalizing various single program analyses to operate on multiple variants is called lifting. Even so, lifted analyses still face scalability issues, despite using smart representations for multiple configurations. In order to further scale the lifted analysis we follow the classic route of addressing inherently hard problems: solving similar but simpler problems. In the analysis

space it amounts to loosing precision in favor of gaining performance, by means of abstraction.

In order to realize this vision, we need to identify new abstractions that work for lifted analyses. In particular, we are interested in abstractions simplifying not the representations of the program state for individual runs, but in abstractions reducing the configuration space of a program. The two basic strategies are to *confound* (join) configurations and to *divide-and-conquer* the configuration space. With confounding configurations distinct variants are merged, so that the analysis has less variability to consider, at the cost of introducing additional control flow. With divide-and-conquer we analyze a subset of configurations at a time. In this paper we realize these two basic strategies by defining *joining* (confounding) abstractions, denoted $\boldsymbol{\alpha}^{\text{join}}$, and *projecting* (divide-and-conquer) abstractions on configurations satisfying the constraint φ, denoted $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}$. Then we use composition operators $\circ$ and $\otimes$ to build other, more sophisticated strategies out of these two basic blocks. We will now illustrate several such strategies by means of example scenarios.

Scenario I: Immediate Feedback during Development

Consider a programmer working on a highly configurable program using incremental analyses during development. The analyses may report warnings and errors such as "variable `x` is used before it is initialized" or "there is a reference outside the bounds of array `a`", which are results of program analyses such as uninitialized variables and interval analysis. The analyses are run regularly and the programmer maintains the cleanness of the code so that no warnings are raised in the committed code. In this scenario where most of the code was analyzed before and the changes are relatively small, it is most likely that an incremental analyzer will conclude quickly that no such alarm messages are present in the code. We build such an analyzer by merging all configurations into a single program with over-approximated control flow. This way the required analyses results may be obtained faster. In our calculus such an abstraction can be specified as:

$$\boldsymbol{\alpha}^{\text{join}}$$

If no warning or error message is reported by this fast abstracted analysis, then all variants are correct and the programmer can continue with the development. Otherwise, a more precise analysis can be subsequently run that will determine which of the reported warnings and errors are genuine. This way both quick feedback and precise results can be delivered to the developer.

Scenario II: Full Precision on Safety-Critical Variants, Speed on Regular Code

Now consider a different scenario of an industrial product line, where only a subset of the products is safety critical. The certification of functional safety for the safety-critical products poses a legal requirement that the software is thoroughly analyzed. This analysis is very slow and it takes many hours in daily continuous integration test. At the same time there is no need to analyze non-certified products so thoroughly. In this segment precision is not so

important, while the changes occur much more often. Analysis speed is thus of high importance. Without abstraction all products could only be analyzed once every two days. The configuration manager considers use of abstractions to cut the analysis time on the non-safety critical variants, so that the analysis could be run daily.

Let the safety-critical variants be identified as those whose configuration satisfies a constraint φ. We would like to analyze all these configurations at full precision, as required by our development process. We are also interested in analyzing all the other configurations, but this could be done fast, at low precision. We compose two abstractions: one (projection on φ) that selects only the critical configurations and another one (projection on $\neg\varphi$) which selects the remaining configurations *and* confounds (joins) their control flow to increase speed. We do not want to apply both abstractions to the same executions twice. The two groups of products share a lot of code, so it is beneficial to perform the entire analysis for both groups simultaneously in a single run benefiting from partial results on either side due to sharing. Such a simultaneous recomposition of two analysis in a single compound analysis is realized in our calculus using a parallel composition "$\otimes$" of abstractions specified as:

$$\boldsymbol{\alpha}_{\varphi}^{\text{proj}} \otimes (\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_{\neg\varphi}^{\text{proj}})$$

A lifted analysis using the above abstraction will analyze the entire product family in one run, gathering information equivalent to running the two parts separately, including a projection on one part of the configuration space, and a sequential composition $\circ$ of a join and projection on the other one. Effectively the speed will benefit both from controlled information loss (join), from dividing the problem space and from efficient representations used in simultaneous analysis execution.

Scenario III: Deliberate Over-Approximation on 3rd-Party Code

An embedded systems vendor is integrating a configurable third-party component into a product line. The third-party component needs to participate in a whole program analysis. However we already know that the external component is of high quality, so in order to speed up the analysis using the abstraction of the component, rather than its precise semantics will be beneficial for the performance. We will first project the analysis on the condition φ which defines the configurations that utilize the component, and then confound together configurations that only differ on the features of the external component, effectively ignoring its internal configuration. As a result we will use an over-approximation of the external component in our analysis. Instead of analysing its complex variability, we will see it as one subsystem that encompasses control flow of all its variants. Let $\mathbb{E}$ be the set of internal features of the integrated component. In our calculus such an abstraction could be specified as follows:

$$\boldsymbol{\alpha}_{\mathbb{E}}^{\text{fignore}} \circ \boldsymbol{\alpha}_{\varphi}^{\text{proj}}$$

The abstraction function (read from right to left): first selects the configurations that are relevant for analyzing the integration of the system and the component, and then ignores (by $\boldsymbol{\alpha}_{\mathbb{E}}^{\text{fignore}}$) the internal features ($\mathbb{E}$) of the component.

In subsequent sections we develop the abstraction calculus that allows specifying abstraction scenarios like those above.

3. Background: Family-Based Program Analyses

We start by summarizing the exisitng background for our work. We define *features*, *configurations*, *feature expressions*, and *feature models*. Hereafter, we describe a simple imperative language $\overline{\text{IMP}}$ for implementing program families. Finally, we give an overview of the basic ideas and concepts of abstract interpretation, and we briefly sketch a lifted constant propagation analysis for $\overline{\text{IMP}}$, formally derived in [11]. We focus on constant propagation for presentation purposes; but our method is generically applicable to any lifted (monotone) static analysis phrased as an abstract interpretation.

3.1. A Language for Program Families

Features, Feature Expressions, and Feature Model. Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite set of Boolean variables representing the *features* available in a program family. Each of the features may be *enabled* or *disabled* in a particular program variant, thus controlling presence and absence of software functionality. A *feature expression*, *FeatExp* formula, is a propositional logic formula over $\mathbb{F}$, defined inductively by:

$$\varphi \quad ::= \quad \text{true} \quad | \quad A \in \mathbb{F} \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2$$

A truth assignment or a *valuation* is a mapping that assigns a truth value to each feature. Given a valuation v, every feature expression evaluates to a truth value. We say that φ is *valid*, denoted as $\models \varphi$, if φ evaluates to *true* for all valuations v. We say that φ is *satisfiable*, denoted as $\text{sat}(\varphi)$, if there exists a valuation v such that φ evaluates to true under v. Otherwise, we say that φ is *unsatisfiable*, denoted as $\text{unsat}(\varphi)$. We say that the formula θ is a semantic consequence of φ, denoted as $\varphi \models \theta$, if for all satisfiable valuations v of φ it follows that θ evaluates to *true* under v. Otherwise, we have $\varphi \not\models \theta$.

The set of *valid configurations* (variants) of a program family is encoded in a separate *feature model* [26], i.e. a tree-like structure that describes which combinations of features and relationships among them are valid. For our purposes a feature model can be equated to a propositional formula [27], say $\psi \in \textit{FeatExp}$, as the semantic aspects of feature models beyond the configuration semantics, are not relevant here. We write $\mathbb{K}_\psi$ to denote the set of all *valid* configurations described by the feature model ψ; i.e. the set of all satisfiable valuations of ψ. One satisfiable valuation v of ψ represents a valid configuration, and it can be also encoded as a conjunction of literals: $k_v = v(A_1) \cdot A_1 \wedge \cdots \wedge v(A_n) \cdot A_n$, where $true \cdot A = A$ and $false \cdot A = \neg A$, such that $k_v \models \psi$. The truth value of a feature in v indicates whether the given feature is *enabled* (included)

or *disabled* (excluded) in the configuration. Let $k_{v_1}, \ldots, k_{v_n}$ $(1 \leq n \leq 2^{|\mathbb{F}|})$ represent all satisfiable valuations of ψ expressed as formulas, then the set of valid configurations is $\mathbb{K}_\psi = \{k_{v_1}, \ldots, k_{v_n}\}$.

Example 3.1. *Let the set of features* $\mathbb{F}$ *be* $\{A, B\}$*. Some feature expressions defined over* $\mathbb{F}$ *are:* $A \vee B$*,* $A \wedge \neg B$*,* $\neg A$*,* B*, etc. The feature model* $\psi = A \vee B$ *yields the following set of* valid *configurations:* $\mathbb{K}_{A \vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$.

The Programming Language. $\overline{\text{IMP}}$ is an extension of the imperative language IMP [28] often used in semantic studies. $\overline{\text{IMP}}$ adds a compile-time conditional statement for encoding multiple variants of a program. The new statement "`#if` (θ) s" contains a feature expression $\theta \in \mathit{FeatExp}$ as a presence condition and a statement s that will be run, i.e. included in a variant, iff the condition θ is satisfied by the corresponding configuration $k \in \mathbb{K}_\psi$. The abstract syntax of the language is given by the following grammar:

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

where n ranges over integers, `x` ranges over variable names *Var*, and $\oplus$ over binary arithmetic operators. The set of all generated statements s is denoted by *Stm*, whereas the set of all expressions e is denoted by *Exp*. Notice that $\overline{\text{IMP}}$ is only used for presentational purposes as a well established minimal language. The introduced methodology is not limited to $\overline{\text{IMP}}$ or its features. In fact in Section 8, we evaluate our approach on program families written in Java.
Remark. Conditional constructs "`#if`" are defined at the level of statements purely for pedagogical reasons. This allows us to keep the presentation focussed and improves readability of definitions and proofs. However, it is known that conditional constructs defined on arbitrary language elements could be translated into constructs that respect the appropriate syntactic structure of the language by code duplication [29].

The semantics of $\overline{\text{IMP}}$ has two stages. First, a preprocessor takes as input an $\overline{\text{IMP}}$ program and a configuration $k \in \mathbb{K}_\psi$, and outputs a variant, i.e. an IMP program without `#if`-s, corresponding to k. Second, the obtained variant is executed (compiled) using the standard IMP semantics [28]. The first stage of computation (also called *projection*) is a simple preprocessor from $\overline{\text{IMP}}$ to IMP specified by the projection function P_k mapping an $\overline{\text{IMP}}$ program family into an IMP single program corresponding to the configuration $k \in \mathbb{K}_\psi$. The function P_k copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP, and recursively pre-processes all sub-statements of compound statements. For "`#if` (θ) s", the statement s is included in the generated variant iff $k \models \theta$, and removed otherwise. Thus, we have:

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

3.2. Abstract Interpretation

We now present the basic ideas and concepts of abstract interpretation.

Complete lattices. A *partial order* [23] 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 $\forall x \in X : x \leq_L u$. Similarly, $\ell \in L$ is a *lower bound* for X, written $\ell \leq_L X$, if $\forall x \in X : \ell \leq_L x$. 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, *greatest lower bound*, $\sqcap$, may be defined.) Usually, binary infix notation, $x \sqcup y$, is used whenever the operator is applied to only two elements, i.e. $x \sqcup y = \sqcup\{x, y\}$. 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 $\bot = \sqcap L$. A function, $f : L \to L$, is *monotone* when $\forall x, y \in L : x \leq_L y \implies f(x) \leq_L f(y)$. An element, $x \in L$, is called a *fixed point* of $f : L \to L$, if $x = f(x)$. *Tarski's fixed point theorem* says that the fixed points of a monotone function, $f : L \to L$, on a complete lattice, $\langle L, \leq_L \rangle$, themselves form a complete lattice. This guarantees the existence of a fixed point, and of a *unique least fixed point*, $\mathrm{lfp}(f) \in L$. A fixed point of a monotone function over an *infinite* height complete lattice is thus well-defined but it is not necessarily *computable*. However, when the height of a complete lattice is *finite* the fixed point may then be computed via *Kleene's fixed point theorem*: $\mathrm{lfp}(f) = \sqcup_i f^i(\bot)$.

Galois connections. We consider the standard Galois connection based abstract interpretation [30]. A *Galois connection* 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.

Galois connections have a number of important properties [31]:

- α and γ are *monotone*;
- $\alpha \circ \gamma$ is *reductive*, i.e. $(\alpha \circ \gamma)(m) \sqsubseteq m$, 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 $\langle L, \leq_L \rangle \underset{\alpha' \circ \alpha}{\overset{\gamma \circ \gamma'}{\leftrightarrows}} \langle N, \leq_N \rangle$.

3.3. Lifted Analysis

In the context of $\overline{\text{IMP}}$ lifting means taking a static analysis that works on IMP programs, and transforming it into an analysis that works on $\overline{\text{IMP}}$ programs, without preprocessing them (i.e. on all the variants simultaneously). We summarize the process briefly below. For more detail, we refer to [11].

Suppose that we have a (monotone) analysis from the abstract interpretation framework for single programs, and we want to lift the analysis to all variants. The analysis operates on a domain $\langle \mathbb{A}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, which is a complete lattice. More specifically, $\mathbb{A}$ is a set equipped with a partial order relation $\sqsubseteq$, a least upper bound (join) $\sqcup$, a greatest lower bound (meet) $\sqcap$, a least element (bottom) $\bot$, and a greatest element (top) $\top$. Using variational abstract interpretation [11], we can derive the corresponding lifted analysis for $\overline{\text{IMP}}$. The lifted analysis domain is $\langle \mathbb{A}^{\mathbb{K}_\psi}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$ where $\mathbb{K}_\psi$ is the set of valid configurations. In the following, we will use constant propagation analysis to demonstrate this method for deriving computationally cheap abstracted lifted analysis. Still, the method is by no means limited to constant propagation, but it is applicable to any analysis from the abstract interpretation framework [23].

Constant Propagation Analysis. This analysis establishes whether a variable has a constant value whenever the execution reaches a given program point. We first define the constant propagation lattice $\langle \mathit{Const}, \sqsubseteq_C \rangle$, where the set $\mathit{Const} = \mathbb{Z} \cup \{\bot_C, \top_C\}$ is partially ordered as follows: $\bot_C \sqsubseteq_C v \sqsubseteq_C \top_C$ for all $v \in \mathit{Const}$, and all elements from $\mathbb{Z}$ are incomparable, i.e. $v_1 \sqsubseteq_C v_2$ iff $v_1 = v_2$ for all $v_1, v_2 \in \mathbb{Z}$. In *Const*, $\top_C$ indicates a value which may be a *non-constant*, and $\bot_C$ indicates *unanalyzed* information. All other elements indicate constant values. The partial ordering $\sqsubseteq_C$ induces a least upper bound, $\sqcup_C$, and a greatest lower bound operator, $\sqcap_C$, on the lattice elements. For example, $0 \sqcup_C 1 = \top_C$, $\top_C \sqcap_C 1 = 1$, etc.

The constant propagation analysis is given in terms of abstract *constant propagation stores*, denoted by a, essentially mappings of variables to elements of *Const*. The idea is for each program variable $\mathtt{x}$, $a(\mathtt{x})$ will give information whether or not $\mathtt{x}$ is a constant and in the case it is what is the value of $\mathtt{x}$. We write $\mathbb{A} = \mathit{Var} \to \mathit{Const}$ to denote the domain of all constant propagation stores. Since *Const* is a complete lattice then so is $\langle \mathbb{A}, \sqsubseteq_\mathbb{A}, \sqcup_\mathbb{A}, \sqcap_\mathbb{A}, \bot_\mathbb{A}, \top_\mathbb{A} \rangle$ obtained by point-wise lifting [23, 28]. Thus, for any $a, a' \in \mathbb{A}$ we have: $a \sqsubseteq_\mathbb{A} a'$ iff $\forall \mathtt{x} \in \mathit{Var}$, $a(\mathtt{x}) \sqsubseteq_C a'(\mathtt{x})$; $(a \sqcup_\mathbb{A} a')(\mathtt{x}) = a(\mathtt{x}) \sqcup_C a'(\mathtt{x})$; $(a \sqcap_\mathbb{A} a')(\mathtt{x}) = a(\mathtt{x}) \sqcap_C a'(\mathtt{x})$; $\bot_\mathbb{A}(\mathtt{x}) = \bot_C$; and $\top_\mathbb{A}(\mathtt{x}) = \top_C$ for any $\mathtt{x} \in \mathit{Var}$. We omit subscripts C and $\mathbb{A}$ in lattice operators whenever they are clear from the context.

Lifted Constant Propagation Analysis. For the lifted constant propagation analysis, we work with the lifted property domain $\langle \mathbb{A}^{\mathbb{K}_\psi}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, where $\mathbb{A}^{\mathbb{K}_\psi}$ is shorthand for the $|\mathbb{K}_\psi|$-fold product $\prod_{k \in \mathbb{K}_\psi} \mathbb{A}$, i.e. there is one separate copy of $\mathbb{A}$ for each valid configuration of $\mathbb{K}_\psi$. The ordering $\dot{\sqsubseteq}$ is lifted configuration-wise; i.e. for $\overline{a}, \overline{a}' \in \mathbb{A}^{\mathbb{K}_\Psi}$ we have $\overline{a} \dot{\sqsubseteq} \overline{a}' \equiv_{def} \pi_k(\overline{a}) \sqsubseteq_\mathbb{A} \pi_k(\overline{a}')$ for all $k \in \mathbb{K}_\psi$. Here π_k selects the k^{th} component of a tuple. Similarly, we lift configuration-wise all other elements of the complete lattice $\mathbb{A}$, obtaining $\dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top}$. In

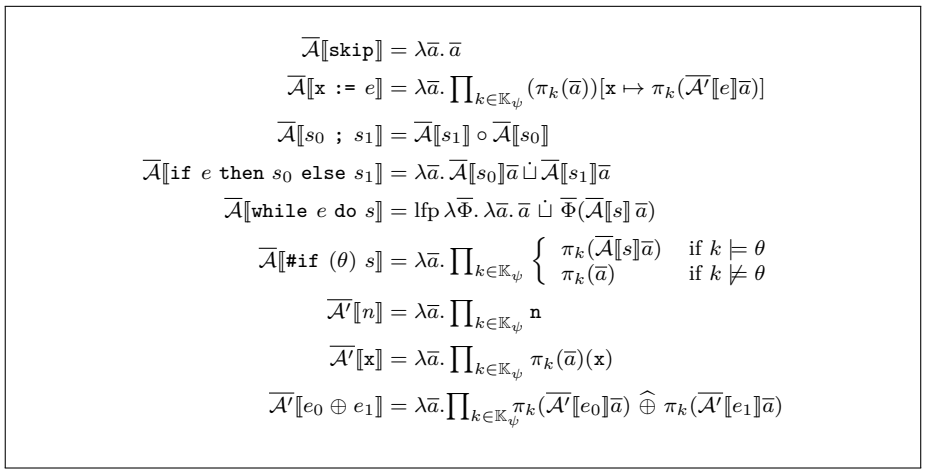

Figure 2: Definitions of $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}_\psi}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathit{Const})^{\mathbb{K}_\psi}$. The analysis of `while` is the least fixed point (lfp) of the functional $\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a} \dot{\sqcup} \overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\,\overline{a})$.

particular, $\overline{a} \dot{\sqcup} \overline{a}' = \prod_{k\in\mathbb{K}_\psi} \pi_k(\overline{a}) \sqcup_{\mathbb{A}} \pi_k(\overline{a}')$; $\overline{a} \dot{\sqcap} \overline{a}' = \prod_{k\in\mathbb{K}_\psi} \pi_k(\overline{a}) \sqcap_{\mathbb{A}} \pi_k(\overline{a}')$; $\dot{\bot} = \prod_{k\in\mathbb{K}_\psi} \bot_{\mathbb{A}} = (\bot_{\mathbb{A}}, \ldots, \bot_{\mathbb{A}})$; and $\dot{\top} = \prod_{k\in\mathbb{K}_\psi} \top_{\mathbb{A}} = (\top_{\mathbb{A}}, \ldots, \top_{\mathbb{A}})$.

The lifted analysis $\overline{\mathcal{A}}[\![s]\!]$ is a function from $\mathbb{A}^{\mathbb{K}_\psi}$ to $\mathbb{A}^{\mathbb{K}_\psi}$. However in practice, using a tuple of $|\mathbb{K}_\psi|$ independent simple functions of type $\mathbb{A} \to \mathbb{A}$ is sufficient, because lifting corresponds to running $|\mathbb{K}_\psi|$ independent analyses in parallel. Thus, the lifted analysis is given by the function $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}_\psi}$, which represents a tuple of $|\mathbb{K}_\psi|$ functions of type $\mathbb{A} \to \mathbb{A}$. The k-th component of $\overline{\mathcal{A}}[\![s]\!]$ defines the analysis corresponding to the valid configuration described by the formula k. In other words, an analysis $\overline{\mathcal{A}}[\![s]\!]$ transforms a lifted store, $\overline{a} \in \mathbb{A}^{\mathbb{K}_\psi}$, into another lifted store of the same type. For simplicity, 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 semantic equations for lifted analysis $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}_\psi}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathit{Const})^{\mathbb{K}_\psi}$ that analyse all valid configurations simultaneously are given in Fig 2. They are systematically derived in [11] by following the steps of calculational approach to abstract interpretation [22]: define collecting semantics, specify a series of Galois connections and compose them with the collecting semantics to obtain the resulting analysis, which is thus sound (correct) by construction. Monotonicity of $\overline{\mathcal{A}}[\![s]\!]$ and $\overline{\mathcal{A}'}[\![e]\!]$ was shown in [11] as well.

The (transfer) function $\overline{\mathcal{A}}[\![s]\!]$ captures the effect of analysing the statement s in a lifted store $\overline{a}$ by computing an output store $\overline{a}'$. For the `skip` statement, the analysis function is an identity on lifted stores. For the assignment statement, `x :=` e, the value of variable `x` is updated in every component of the input store $\overline{a}$ by the value of the expression e evaluated in the corresponding component of $\overline{a}$. The analysis of conditional statement `if` results in the least upper bound (join) of the effects from the two corresponding branches. It abstracts away the

analysis information at the guard (condition) point. 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. This fixed point exists and is computable by Kleene's fixed point theorem, since the functional is a monotone function over complete lattice with finite height (that is the *Const* lattice) [11, 32] [1]. For the `#if` (θ) s statement, we check for each valid configuration k [2] 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. The function $\overline{\mathcal{A}'}[\![e]\!]$ describes the result of evaluating the expression e in a lifted store. Note that, for each binary operator $\oplus$, we define the corresponding constant propagation operator $\widehat{\oplus}$, which operates on values from *Const*, as follows:

$$v_0 \mathbin{\widehat{\oplus}} v_1 = \begin{cases} \bot & \text{if } v_0 = \bot \vee v_1 = \bot \\ \mathtt{n} & \text{if } v_0 = \mathtt{n}_0 \wedge v_1 = \mathtt{n}_1, \text{ where } \mathtt{n} = \mathtt{n}_0 \oplus \mathtt{n}_1 \\ \top & \text{otherwise} \end{cases} \tag{2}$$

We lift the above operation configuration-wise, and in this way obtain a new operation $\hat{\widehat{\oplus}}$ on tuples of *Const* values. We have $\overline{v_1}\,\hat{\widehat{\oplus}}\,\overline{v_2} = \prod_{k\in\mathbb{K}_\psi} (\pi_k(\overline{v_1}) \mathbin{\widehat{\oplus}} \pi_k(\overline{v_2}))$, for $\overline{v_1}, \overline{v_2} \in \mathit{Const}^{\mathbb{K}_\psi}$. For example, $(2,5)\hat{\widehat{+}}(5,2) = (7,7)$.

Example 3.2. *Consider the IMP program* S_1*:*

$$\begin{array}{l} \boldsymbol{x} := 0; \\ \texttt{\#if}\ (A)\ \boldsymbol{x} := \boldsymbol{x} + 1; \\ \texttt{\#if}\ (B)\ \boldsymbol{x} := 1 \end{array}$$

with the set $\mathbb{K}_{A\vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$*. By using the rules of Figure 2, we can calculate* $\overline{\mathcal{A}}[\![S_1]\!]$ *for a lifted store in which* $\boldsymbol{x}$ *is uninitialized, i.e. it has the value* $\top$*. We assume a convention here that the first component of the store corresponds to configuration* $A \wedge B$*, the second to* $A \wedge \neg B$*, and the third to* $\neg A \wedge B$*. We write* $\overline{a_0} \overset{\overline{\mathcal{A}}[\![s]\!]}{\longmapsto} \overline{a_1}$ *when* $\overline{\mathcal{A}}[\![s]\!]\overline{a_0} = \overline{a_1}$*. We have:*

$$\big(\overbrace{[\boldsymbol{x}\mapsto\top]}^{A\wedge B},\overbrace{[\boldsymbol{x}\mapsto\top]}^{A\wedge\neg B},\overbrace{[\boldsymbol{x}\mapsto\top]}^{\neg A\wedge B}\big) \overset{\overline{\mathcal{A}}[\![\boldsymbol{x}:=0]\!]}{\longmapsto} \big([\boldsymbol{x}\mapsto 0],[\boldsymbol{x}\mapsto 0],[\boldsymbol{x}\mapsto 0]\big) \overset{\overline{\mathcal{A}}[\![\#\mathsf{if}\ (A)\ \boldsymbol{x}:=\boldsymbol{x}+1]\!]}{\longmapsto}$$
$$\big([\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto 0]\big) \overset{\overline{\mathcal{A}}[\![\#\mathsf{if}\ (B)\ \boldsymbol{x}:=1]\!]}{\longmapsto} \big([\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto 1]\big)$$

Note that when we analyse `#if` *with the presence condition* A *(resp.,* B*), the input store is updated only for components* $A\wedge B$*,* $A\wedge\neg B$ *(resp.,* $A\wedge B$*,* $\neg A\wedge B$*)*

[1] If the lattice is with infinite height and infinite ascending chains (e.g. the Interval lattice [30]), then the computation of the fixed point can be achieved by using so-called widening operators [30, 23]

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

which satisfy that presence condition by the effect of analysing the statement associated with the given `#if`*. After analysing* S_1*, the variable* $\boldsymbol{x}$ *has the constant value 1 for all valid configurations.*

Let S_2 *be a program obtained from* S_1*, such that* `#if` (B) $\boldsymbol{x} := 1$ *is replaced with* `#if` (B) $\boldsymbol{x} := \boldsymbol{x} - 1$*. Then, we have:* $\overline{\mathcal{A}}[\![S_2]\!]\big([\boldsymbol{x} \mapsto \top],[\boldsymbol{x} \mapsto \top],[\boldsymbol{x} \mapsto \top]\big) = \big([\boldsymbol{x} \mapsto 0],[\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto -1]\big)$*, i.e.*

$$\begin{aligned}\big([\boldsymbol{x} \mapsto \top],[\boldsymbol{x} \mapsto \top],[\boldsymbol{x} \mapsto \top]\big) &\overset{\overline{\mathcal{A}}[\![\boldsymbol{x}:=0;\#\mathsf{if}\ (A)\ \boldsymbol{x}:=\boldsymbol{x}+1]\!]}{\longmapsto} \big([\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto 0]\big) \\ &\overset{\overline{\mathcal{A}}[\![\#\mathsf{if}\ (B)\ \boldsymbol{x}:=\boldsymbol{x}-1]\!]}{\longmapsto} \big([\boldsymbol{x} \mapsto 0],[\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto -1]\big)\end{aligned}$$

We will use programs S_1 *and* S_2 *as running examples throughout the paper.*□

4. Variability Abstractions

When the set of configurations $\mathbb{K}_\psi$ is large, calculations on the property domain $\mathbb{A}^{\mathbb{K}_\psi}$ become expensive, even when using symbolic representations or sharing to avoid direct storage of $|\mathbb{K}_\psi|$-sized tuples as done in [9]. We want to replace $\mathbb{A}^{\mathbb{K}_\psi}$ with a smaller domain obtained by abstraction and perform an approximate, but feasible, lifted analysis on it.

4.1. Basic Abstractions

We describe a compositional way of constructing abstractions over the domain $\mathbb{A}^{\mathbb{K}}$, where $\mathbb{K}$ represents an arbitrary set of valid configurations, using two basic operators, join and projection, along with a sequential and parallel composition of abstractions. The set of abstractions G^{va} is generated by the following grammar:

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

where $\varphi \in FeatExp$. Below we define the abstractions and motivate them with examples. For the sake of readability, we use the constant propagation lattice $\mathbb{A}$, however the results hold for any complete lattice.

Join. Consider an analysis that is run interactively and finds simple errors and warnings. The analysis must be fast and it should consider all legal configurations $\mathbb{K}$. It is not problematic if some spurious errors are introduced, as previously mentioned, a more thorough analysis is run regularly. Here, the precision with respect to configurations can be reduced by confounding the control-flow of all the products, obtaining an analysis that runs as if it was analyzing a single product, but involving code that participate in all products.

The *join* abstraction gathers the information about all configurations $k \in \mathbb{K}$ into one value of $\mathbb{A}$. We formulate the abstraction $\boldsymbol{\alpha}^{\text{join}} : \mathbb{A}^{\mathbb{K}} \to \mathbb{A}^{\{\bigvee_{k\in\mathbb{K}} k\}}$ and the concretization function $\boldsymbol{\gamma}^{\text{join}} : \mathbb{A}^{\{\bigvee_{k\in\mathbb{K}} k\}} \to \mathbb{A}^{\mathbb{K}}$ as:

$$\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = \big(\textstyle\bigsqcup_{k\in\mathbb{K}} \pi_k(\overline{a})\big) \quad \text{and} \quad \boldsymbol{\gamma}^{\text{join}}(a) = \prod_{k\in\mathbb{K}} a \tag{3}$$

for $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $a \in \mathbb{A}$. We overload abstraction names (α) to apply not only to domain elements but also to the sets of configurations, and, later, to the sets of features and program code. The new abstract set of valid configurations is $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\bigvee_{k\in\mathbb{K}} k\}$. Thus, we obtain only one abstract valid configuration denoted by the compound formula $\bigvee_{k\in\mathbb{K}} k$. Observe that this means that the obtained abstract domain is effectively $\mathbb{A}^1$, which is isomorphic to $\mathbb{A}$. The proposed abstraction–concretization pair is a Galois connection, which means that it can be used to construct analyses using calculational abstract interpretation.

Theorem 4.1. $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{join}}(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ *is a Galois connection* [3].

Proof. *Let* $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ *and* $a \in \mathbb{A}^{\boldsymbol{\alpha}^{\text{join}}(\mathbb{K})}$.

$$\begin{aligned}
\boldsymbol{\alpha}^{\text{join}}(\overline{a}) \mathbin{\dot{\sqsubseteq}} (a) &\iff \textstyle\bigsqcup_{k\in\mathbb{K}} \pi_k(\overline{a}) \sqsubseteq a && \text{(by def. of } \boldsymbol{\alpha}^{\text{join}}) \\
&\iff \forall k \in \mathbb{K}.\, \pi_k(\overline{a}) \sqsubseteq a && \text{(by def. of } \sqcup) \\
&\iff \overline{a} \mathbin{\dot{\sqsubseteq}} \boldsymbol{\gamma}^{\text{join}}(a) && \text{(by def. of } \boldsymbol{\gamma}^{\text{join}})
\end{aligned}$$

□

Example 4.1. *Let us return to the scenario of using* join *for improving the analysis performance. Assume that the feature model is given by* $\psi = A \vee B$ *with valid configurations* $\mathbb{K}_{A\vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. *By applying the join abstraction on lifted stores generated during analysis of the program* S_1 *in Example 3.2, we obtain:* $\boldsymbol{\alpha}^{\text{join}}(\big([\boldsymbol{x} \mapsto \top],[\boldsymbol{x} \mapsto \top],[\boldsymbol{x} \mapsto \top]\big)) = \big([\boldsymbol{x} \mapsto \top]\big)$, $\boldsymbol{\alpha}^{\text{join}}(\big([\boldsymbol{x} \mapsto 0],[\boldsymbol{x} \mapsto 0],[\boldsymbol{x} \mapsto 0]\big)) = \big([\boldsymbol{x} \mapsto 0]\big)$, $\boldsymbol{\alpha}^{\text{join}}(\big([\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto 1]\big)) = \big([\boldsymbol{x} \mapsto 1]\big)$, *and* $\boldsymbol{\alpha}^{\text{join}}(\big([\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto 1],[\boldsymbol{x} \mapsto 0]\big)) = \big([\boldsymbol{x} \mapsto \top]\big)$. *In all cases the state representation has been significantly decreased to only one component (i.e. 1-sized tuple). In the first three cases, the abstraction promptly notices that* $\boldsymbol{x}$ *is a constant regardless of the configuration. In the last case, the abstraction looses precision by saying that* $\boldsymbol{x}$ *is not a constant in general, even if it was a constant in each of the configurations considered in isolation.*

If we infer the corresponding abstracted analysis for $\boldsymbol{\alpha}^{\text{join}}$ *(see Section 5 for details), we will obtain the final store* $\big([\boldsymbol{x} \mapsto \top]\big)$ *after analyzing the program* S_1 *in the input store* $\big([\boldsymbol{x} \mapsto \top]\big)$, *although we can see that* $\boldsymbol{x}$ *has the same constant value 1 for all configurations in the final store of the concrete (unabstracted) lifted analysis shown in Example 3.2. This indicates that more precise abstractions need to be introduced. We will remedy this shortly.* □

Projection. In industrial practice the number of products actually deployed is often only a small subset of $\mathbb{K}$. In such case, analyzing all valid configurations seems unnecessary, and performance of analyses can be improved by abstracting

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

many products away. This is achieved by a configuration projection, which removes configurations that do not satisfy a given constraint. Projection can be helpful in other similar scenarios; for instance, to parallelize the analysis—by partitioning the product space using project and analyzing each partition separately.

Let φ be a feature expression. We define a *projection* abstraction mapping $\mathbb{A}^{\mathbb{K}}$ into the domain $\mathbb{A}^{\{k\in\mathbb{K}|k\models\varphi\}}$, which preserves only the values corresponding to configurations from $\mathbb{K}$ that satisfy φ. The information about configurations violating φ is disregarded. The abstraction and concretization functions between $\mathbb{A}^{\mathbb{K}}$ and $\mathbb{A}^{\{k\in\mathbb{K}|k\models\varphi\}}$ are defined as follows:

$$\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\overline{a}) = \textstyle\prod_{k\in\mathbb{K},k\models\varphi} \pi_k(\overline{a}) \quad \text{and} \quad \boldsymbol{\gamma}^{\text{proj}}_{\varphi}(\overline{a}') = \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{a}') & \text{if } k \models \varphi \\ \top & \text{if } k \not\models \varphi \end{cases} \tag{4}$$

for $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $\overline{a}' \in \mathbb{A}^{\{k\in\mathbb{K}|k\models\varphi\}}$. The new set of configurations is $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K}) = \{k \in \mathbb{K} \mid k \models \varphi\}$. Naturally, we have a Galois connection here.

Theorem 4.2. $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}{\overset{\boldsymbol{\gamma}^{\text{proj}}_{\varphi}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ *is a Galois connection.*

Proof. *Let* $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ *and* $\overline{a}' \in \mathbb{A}^{\{k\in\mathbb{K}|k\models\varphi\}}$.

$$\begin{aligned}
\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\overline{a}) \,\dot{\sqsubseteq}\, \overline{a}' &\iff \forall k \in \mathbb{K}, k \models \varphi.\, \pi_k(\overline{a}) \sqsubseteq \pi_k(\overline{a}') && \text{(by def. of } \boldsymbol{\alpha}^{\text{proj}}_{\varphi}\text{)} \\
&\iff \forall k \in \mathbb{K}, k \models \varphi.\, \pi_k(\overline{a}) \sqsubseteq \pi_k(\overline{a'}) \,\wedge\, \forall k \in \mathbb{K}, k \not\models \varphi.\, \pi_k(\overline{a}) \sqsubseteq \top && \text{(by def. of } \top\text{)} \\
&\iff \overline{a} \,\dot{\sqsubseteq}\, \boldsymbol{\gamma}^{\text{proj}}_{\varphi}(\overline{a}') && \text{(by def. of } \gamma^{proj}_{\varphi}\text{)}
\end{aligned}$$

□

Notice that $\boldsymbol{\alpha}^{\text{proj}}_{\text{true}}$ is the identity function, since $k \models \text{true}$ for all $k \in \mathbb{K}$. On the other hand $\boldsymbol{\alpha}^{\text{proj}}_{\text{false}}$ is the coarsest collapsing abstraction that maps any tuple into an empty one, since $k \not\models \text{false}$, for all k.

Example 4.2. *Let us revisit our scenario, where a set of deployed configurations is much smaller than the set of configurations defined by the feature model* ψ. *Let us consider the lifted stores* $\overline{a}_{S_1} = \big([\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto 0]\big)$ *and* $\overline{a}_{S_2} = \big([\boldsymbol{x}\mapsto 0],[\boldsymbol{x}\mapsto 1],[\boldsymbol{x}\mapsto -1]\big)$ *obtained during analysis of programs* S_1 *and* S_2 *in Example 3.2. The set of deployed products is defined by formula* $\varphi = A$. *By definition of projection Eqn. (4), we have:* $\boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_1}) = \big(\pi_{A\wedge B}(\overline{a}_{S_1}), \pi_{A\wedge\neg B}(\overline{a}_{S_1})\big) = \big([\boldsymbol{x}\mapsto 1], [\boldsymbol{x}\mapsto 1]\big)$, $\boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_2}) = \big([\boldsymbol{x}\mapsto 0], [\boldsymbol{x}\mapsto 1]\big)$; *and* $\boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_1}) = (\pi_{\neg A\wedge B}(\overline{a}_{S_1})) = ([\boldsymbol{x}\mapsto 0])$, $\boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_2}) = ([\boldsymbol{x}\mapsto -1])$. *The state representation is effectively decreased to two components for* $\boldsymbol{\alpha}^{\text{proj}}_{A}$, *and to one component for* $\boldsymbol{\alpha}^{\text{proj}}_{\neg A}$. □

An attentive reader might discount the idea of the projection abstraction as being overly heavy. In the end, it appears to be equivalent to running the original analysis, just with a strengthened feature model ($\psi \wedge \varphi$). However,

as we shall see in the subsequent developments, projection is indeed useful. Thanks to the composition operators it can enter intricate scenarios that cannot be expressed using a simple strengthening of a global feature model. We use two composition operators, sequential and parallel composition, to build more complex abstractions out of the two fundamental abstractions, join and projection. This also allows us to keep the number of operators in the framework low, since many other sugared operators can be derived from the basic ones from G^{va}.

Sequential Composition. Let $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ and $\langle \mathbb{A}^{\alpha_1(\mathbb{K})}, \dot{\sqsubseteq} \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_2(\alpha_1(\mathbb{K}))}, \dot{\sqsubseteq} \rangle$ be two Galois connections. Then, we define their *composition* as $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_2 \circ \alpha_1)(\mathbb{K})}, \dot{\sqsubseteq} \rangle$, where

$$(\alpha_2 \circ \alpha_1)(\overline{a}) = \alpha_2(\alpha_1(\overline{a})) \quad \text{and} \quad (\gamma_1 \circ \gamma_2)(\overline{a}') = \gamma_1(\gamma_2(\overline{a}')) \tag{5}$$

for $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ and $\overline{a}' \in \mathbb{A}^{(\alpha_2 \circ \alpha_1)(\mathbb{K})}$. The abstract set of configurations is $(\alpha_2 \circ \alpha_1)(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$.

Example 4.3. *Now consider the process of deriving an analysis, which only considers products actually deployed described by a formula φ (see previous example), but which should trade precision for speed, by confounding their execution. Such an analysis is derived using the composed abstraction: $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\varphi}$.*

Let $\varphi = A$. Configurations $A \wedge B$ and $A \wedge \neg B$ satisfy φ, whereas $\neg\varphi$ is satisfied only by $\neg A \wedge B$. We have: $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_1}) = (\pi_{A \wedge B}(\overline{a}_{S_1}) \sqcup \pi_{A \wedge \neg B}(\overline{a}_{S_1})) = ([\boldsymbol{x} \mapsto 1])$, and $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_1}) = (\pi_{\neg A \wedge B}(\overline{a}_{S_1})) = ([\boldsymbol{x} \mapsto 0])$.

If we now derive the abstracted analysis for $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}$ and analyze the program S_1, we will obtain the final store $([\boldsymbol{x} \mapsto 1])$. This means that we precisely obtained the information that $\boldsymbol{x}$ is the constant 1 at the end of the program S_1 for all configurations that satisfy A. Still, we have disregarded the information for the configuration $\neg A \wedge B$. So, a more precise analysis is needed. □

Parallel Composition. Consider a family where two groups of variants share the same code base: one group is safety-critical, the other comprises non-critical products. The former should be analyzed with highest precision possible to obtain the most precise analysis results, the latter can be analyzed faster. We can set up such analyses by using a projection abstraction to analyze the safety-critical group precisely, and the join abstraction to analyze the non-critical group. However running the analyses twice, ignores the fact that the code is shared between the groups. We can combine two separate analyses by creating a compound abstraction: a *product* of the two. The product abstraction will correspond exactly to executing the projection on the safety-critical products, and join on the non-critical ones. But since the product creates a single Galois connection of the two, it can be used to derive an analysis which will deliver this in a single run, which is more efficient due to reuse of the states explored.

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 $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{\alpha_2(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ over the same domain $\mathbb{A}^{\mathbb{K}}$ can be composed into one that combines the abstraction results "side-by-side". The result is a new compound abstraction, $\alpha_1 \otimes \alpha_2$, of

the domain $\mathbb{A}^{\mathbb{K}}$ obtained by applying the two simpler abstractions in parallel. The parallel composition of abstractions is defined using a direct tensor product. For the resulting Galois connection, we have $(\alpha_1 \otimes \alpha_2)(\mathbb{K}) = \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$. Given $\overline{a}_1 \in \mathbb{A}^{\alpha_1(\mathbb{K})}$ and $\overline{a}_2 \in \mathbb{A}^{\alpha_2(\mathbb{K})}$, we first define a helper operator $\overline{a}_1 \times \overline{a}_2 \in \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$ that combines two lifted stores with overlapping configurations (components) as follows:

$$\overline{a}_1 \times \overline{a}_2 = \prod_{k \in \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})} \begin{cases} \pi_k(\overline{a}_1) & \text{if } k \in \alpha_1(\mathbb{K}) \setminus \alpha_2(\mathbb{K}) \\ \pi_k(\overline{a}_1) \sqcup \pi_k(\overline{a}_2) & \text{if } k \in \alpha_1(\mathbb{K}) \cap \alpha_2(\mathbb{K}) \\ \pi_k(\overline{a}_2) & \text{if } k \in \alpha_2(\mathbb{K}) \setminus \alpha_1(\mathbb{K}) \end{cases}$$

The direct tensor product is given as $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1 \otimes \alpha_2}{\overset{\gamma_1 \otimes \gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}, \dot{\sqsubseteq} \rangle$, where

$$(\alpha_1 \otimes \alpha_2)(\overline{a}) = \alpha_1(\overline{a}) \times \alpha_2(\overline{a}) \quad \text{and} \quad (\gamma_1 \otimes \gamma_2)(\overline{a}') = \gamma_1(\pi_{\alpha_1(\mathbb{K})}(\overline{a}')) \sqcap \gamma_2(\pi_{\alpha_2(\mathbb{K})}(\overline{a}')) \tag{6}$$

where $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, $\overline{a}' \in \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}$, $\pi_{\alpha_1(\mathbb{K})}(\overline{a}') = \prod_{k \in \alpha_1(\mathbb{K})} \pi_k(\overline{a}')$ and $\pi_{\alpha_2(\mathbb{K})}(\overline{a}') = \prod_{k \in \alpha_2(\mathbb{K})} \pi_k(\overline{a}')$.

Theorem 4.3. *$\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1 \otimes \alpha_2}{\overset{\gamma_1 \otimes \gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ is a Galois connection.*

Proof. *We show: $(\alpha_1 \otimes \alpha_2)(\overline{a}) \,\dot{\sqsubseteq}\, \overline{a'} \iff \overline{a} \,\dot{\sqsubseteq}\, (\gamma_1 \otimes \gamma_2)(\overline{a'})$ for all $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, $\overline{a'} \in \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}$ [33, App. A].* □

Example 4.4. *Assume that for products without the feature A we need precise analysis results, and for products containing this feature we do not need so precise results. We are interested in analyzing products without A thoroughly, while the analysis of the products with A can be speeded up. To this end we build the following abstraction: $(\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{\neg A}$. For $\overline{a}_{S_1} = \big([\boldsymbol{x} \mapsto 1], [\boldsymbol{x} \mapsto 1], [\boldsymbol{x} \mapsto 0]\big)$, we have $(\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_1}) = (\pi_{A \wedge B}(\overline{a}_{S_1}) \sqcup \pi_{A \wedge \neg B}(\overline{a}_{S_1}), \pi_{\neg A \wedge B}(\overline{a}_{S_1})) = ([\boldsymbol{x} \mapsto 1], [\boldsymbol{x} \mapsto 0])$.*

The abstracted analysis $(\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{\neg A}$ when executed for the program S_1 will report $\big([\boldsymbol{x} \mapsto 1], [\boldsymbol{x} \mapsto 1]\big)$ as the final lifted store (see Example 5.1 for details). Hence, it provides the precise analysis information for $\boldsymbol{x}$ at the end of S_1. This example shows that by using an appropriate combination of abstraction operators, we can control the precision as much as necessary. The above four operators are also sufficient to partition the configuration space arbitrarily, and to recombine the analysis results of partitions in one run. □

4.2. Derived Abstractions

We shall now discuss more abstractions that can be derived from the above basic ones.

Join-Project. Recall the construction of Example 4.3, where we combined projection with a join in order to confound a subset of legal configurations. This pattern has occurred so often in our experiments that we introduced a syntactic sugar for it. For a formula φ over features, the abstraction $\boldsymbol{\alpha}_{\varphi}^{\text{join-proj}}$ gathers the information about all valid configurations $k \in \mathbb{K}$ that satisfy φ, i.e. $k \models \varphi$, into one value of $\mathbb{A}$, whereas the information about all other valid configurations $k \in \mathbb{K}$ that do not satisfy φ is disregarded. We define

$$\boldsymbol{\alpha}_{\varphi}^{\text{join-proj}} = \boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_{\varphi}^{\text{proj}} \quad \text{and} \quad \boldsymbol{\gamma}_{\varphi}^{\text{join-proj}} = \boldsymbol{\gamma}_{\varphi}^{\text{proj}} \circ \boldsymbol{\gamma}^{\text{join}} \tag{7}$$

where $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}_{\varphi}^{\text{proj}}}{\overset{\boldsymbol{\gamma}_{\varphi}^{\text{proj}}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}_{\varphi}^{\text{proj}}(\mathbb{K})}, \dot{\sqsubseteq} \rangle$, $\langle \mathbb{A}^{\boldsymbol{\alpha}_{\varphi}^{\text{proj}}(\mathbb{K})}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathbb{A}^{(\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_{\varphi}^{\text{proj}})(\mathbb{K}))}, \dot{\sqsubseteq} \rangle$ are Galois connections. Now the compositions in Example 4.3 can be written simply as $\boldsymbol{\alpha}_{A}^{\text{join-proj}}$ and $\boldsymbol{\alpha}_{\neg A}^{\text{join-proj}}$.

Ignoring features. We now consider a feature ignore abstraction, which simplifies lifted domains by confounding executions differing only on uninteresting features. We first present a simple case of ignoring a single feature $A \in \mathbb{F}$ that is not directly relevant for the current analysis. The *ignore feature* A abstraction, denoted $\boldsymbol{\alpha}_{A}^{\text{fignore}}$, merges any configurations that only differ with regard to A, and are identical with regard to remaining features, $\mathbb{F} \backslash \{A\}$. Given a formula (feature expression) φ, 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, we write $\varphi \backslash_A$ for 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 a corresponding abstract configuration in $\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K})$ determined by the formula $\bigvee_{k \in \mathbb{K}, k \backslash_A \equiv k'} k$. Thus, the new set of configurations is given by $\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K}) = \{\bigvee_{k \in \mathbb{K}, k \backslash_A \equiv k'} k \mid k' \in \{k \backslash_A \mid k \in \mathbb{K}\}\}$. The abstraction $\boldsymbol{\alpha}_{A}^{\text{fignore}} : \mathbb{A}^{\mathbb{K}} \to \mathbb{A}^{\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K})}$ and concretization functions $\boldsymbol{\gamma}_{A}^{\text{fignore}} : \mathbb{A}^{\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K})} \to \mathbb{A}^{\mathbb{K}}$ are:

$$\boldsymbol{\alpha}_{A}^{\text{fignore}}(\overline{a}) = \textstyle\prod_{k' \in \boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K})} \bigsqcup_{k \in \mathbb{K}, k \models k'} \pi_k(\overline{a}), \boldsymbol{\gamma}_{A}^{\text{fignore}}(\overline{a}') = \textstyle\prod_{k \in \mathbb{K}} \pi_{k'}(\overline{a}') \text{ if } k \models k' \tag{8}$$

It turns out that ignoring features can be derived from the above abstractions.

Theorem 4.4. *Let* $\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K}) = \{k'_1, \ldots, k'_n\}$. *Then:*

$$\boldsymbol{\alpha}_{A}^{\text{fignore}} = \boldsymbol{\alpha}_{k'_1}^{\text{join-proj}} \otimes \cdots \otimes \boldsymbol{\alpha}_{k'_n}^{\text{join-proj}} \quad \textit{and} \quad \boldsymbol{\gamma}_{A}^{\text{fignore}} = \boldsymbol{\gamma}_{k'_1}^{\text{join-proj}} \otimes \cdots \otimes \boldsymbol{\gamma}_{k'_n}^{\text{join-proj}} \ .$$

Proof. *By induction on the length of the set* $\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K})$ *[33, App. A].* □

Example 4.5. *We consider the lifted store* $\overline{a}_{S_2} = \big([\boldsymbol{x} \mapsto 0], [\boldsymbol{x} \mapsto 1], [\boldsymbol{x} \mapsto -1]\big)$ *with* $\mathbb{K}_{A \vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. *Then, we have* $\boldsymbol{\alpha}_{A}^{\text{fignore}}(\mathbb{K}_{A \vee B}) = \{(A \wedge B) \vee$

$(\neg A \wedge B), A \wedge \neg B\}$ *and* $\boldsymbol{\alpha}_A^{\text{fignore}}(\overline{a}_{S_2}) = (\pi_{A\wedge B}(\overline{a}_{S_2}) \sqcup \pi_{\neg A\wedge B}(\overline{a}_{S_2}), \pi_{A\wedge\neg B}(\overline{a}_{S_2})) = ([\boldsymbol{x}\mapsto\top], [\boldsymbol{x}\mapsto 1])$. *On the other hand, we have* $\boldsymbol{\alpha}_B^{\text{fignore}}(\mathbb{K}_{A\vee B}) = \{(A \wedge B) \vee (A \wedge \neg B), \neg A \wedge B\}$ *and* $\boldsymbol{\alpha}_B^{\text{fignore}}(\overline{a}_{S_2}) = (\pi_{A\wedge B}(\overline{a}_{S_2}) \sqcup \pi_{A\wedge\neg B}(\overline{a}_{S_2}), \pi_{\neg A\wedge B}(\overline{a}_{S_2})) = ([\boldsymbol{x}\mapsto\top], [\boldsymbol{x}\mapsto -1])$. $\square$

Now, if we need to ignore a larger number of features (say features outside a certain component of interest), we can do it using a feature ignore operator which simply ignores a set of features $\{A_1, \ldots, A_k\} \subseteq \mathbb{F}$. The definitions of $\boldsymbol{\alpha}_{\{A_1,\ldots,A_k\}}^{\text{fignore}}$ and $\boldsymbol{\gamma}_{\{A_1,\ldots,A_k\}}^{\text{fignore}}$ can be obtained by generalizing the definitions of $\boldsymbol{\alpha}_A^{\text{fignore}}$ and $\boldsymbol{\gamma}_A^{\text{fignore}}$, where a set of features $\{A_1, \ldots, A_k\}$ is taken into account instead of only one feature A.

It follows from the theorems of Section 4.1 that all the derived pairs of abstraction-concretization functions are Galois connections.

5. Abstracting Lifted Analyses

We will now demonstrate how to derive algorithmically abstracted lifted analyses (i.e. equations for transfer functions) using the operators of Section 4. We use the case of constant propagation as an example. Recall that lifted constant propagation analysis has been specified by: 1) the domain $\mathbb{A}^{\mathbb{K}_\psi}$; 2) the statement transfer function $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}_\psi}$; and 3) the expression evaluation (transfer) function $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathit{Const})^{\mathbb{K}_\psi}$. Let $\langle \mathbb{A}^{\mathbb{K}_\psi}, \dot{\sqsubseteq}\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}^{\alpha(\mathbb{K}_\psi)}, \dot{\sqsubseteq}\rangle$ be a Galois connection constructed using the operators presented in Section 4. For the sake of brevity, we will also write $(\alpha, \gamma) \in G^{va}$ or only $\alpha \in G^{va}$ to denote a Galois connection (abstraction) obtained in such a way.

Any function f defined on the concrete domain of a Galois connection can be abstracted to work on the abstract domain by applying concretization to its argument and an abstraction to its value, i.e. by the function $F = \alpha \circ f \circ \gamma$, where $\circ$ denotes the usual composition of functions. In fact, any monotone over-approximation of the composition $\alpha \circ f \circ \gamma$ is sufficient for a sound analysis. Even fixed points can be transferred from a concrete to an abstract domain of a Galois connection. If both domains are complete lattices and f is a monotone function on the concrete domain, then using fixed point transfer theorem (FPT for short) [32]: $\alpha(\text{lfp}f) \sqsubseteq \text{lfp}F \sqsubseteq \text{lfp}F^\#$. Here $F = \alpha \circ f \circ \gamma$ and $F^\#$ is some monotone, conservative *over*-approximation of F, i.e. $F \sqsubseteq F^\#$. The calculational approach to abstract interpretation [22] used in this work, advocates simple algebraic manipulation to obtain a *direct expression* for the function F (if it exists) or for an over-approximation $F^\#$.

In our case, for any lifted store $\overline{a} \in \mathbb{A}^{\mathbb{K}_\psi}$, we calculate an abstracted lifted store by $\alpha(\overline{a}) = \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)}$. Now, we use a Galois connection to derive an over-approximation of $\alpha \circ \overline{\mathcal{A}}[\![s]\!] \circ \gamma$ obtaining a new abstracted statement transfer function $\overline{\mathcal{D}}_\alpha[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K}_\psi)}$. Similarly, one can derive an abstracted analysis for expressions $\overline{\mathcal{D}'}_\alpha[\![e]\!]$, approximating $\alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma$. These approximations are derived using structural induction on statements (respectively on expressions),

in a process that resembles a simple algebraic calculation, deceivingly akin to equation reasoning.

Consider the derivation steps for the static conditional statement "`#if` (θ) s" in detail. Our inductive hypothesis (IH) is that for statements s' that are structurally smaller than "`#if` (θ) s" the (yet-to-be-calculated) $\overline{\mathcal{D}}_\alpha[\![s']\!]$ soundly approximates $\alpha \circ \overline{\mathcal{A}}[\![s']\!] \circ \gamma$, formally: $\alpha \circ \overline{\mathcal{A}}[\![s']\!] \circ \gamma \mathrel{\dot\sqsubseteq} \overline{\mathcal{D}}_\alpha[\![s']\!]$. The derivation begins with composing the concretization and abstraction functions with the concrete transfer function and then proceeds by expanding definitions:

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!] \circ \gamma)(\overline{d}) = \alpha(\overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!](\gamma(\overline{d}))) = && \text{(by def. of } \circ) \\
&\quad = \alpha\Big(\prod_{k \in \mathbb{K}_\psi} \begin{cases} \pi_k(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d})) & \text{if } k \models \theta \\ \pi_k(\gamma(\overline{d})) & \text{if } k \not\models \theta \end{cases} \Big) && \text{(def. of } \overline{\mathcal{A}} \text{ in Fig. 2)} \\
&\quad \mathrel{\dot\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\alpha(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d}))) & \text{if } k' \models \theta \\ \pi_{k'}(\alpha(\gamma(\overline{d}))) \sqcup \pi_{k'}(\alpha(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d}))) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \\ \pi_{k'}(\alpha(\gamma(\overline{d}))) & \text{if } k' \models \neg\theta \end{cases} \\
&\qquad && \text{(by Lemma 5.1 below)} \\
&\quad \mathrel{\dot\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } k' \models \theta \\ \pi_{k'}(\overline{d}) \sqcup \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \\ \pi_{k'}(\overline{d}) & \text{if } k' \models \neg\theta \end{cases} \\
&\qquad && \text{(by IH and } \alpha \circ \gamma \text{ reductive)} \\
&\quad = \overline{\mathcal{D}}_\alpha[\![\texttt{\#if } (\theta)\ s]\!]\,\overline{d}
\end{aligned}
$$

The 'reductive' property of all Galois connections [31, 22] that we use is $(\alpha \circ \gamma)(\overline{d}) \sqsubseteq \overline{d}$ for all $\overline{d}$. In the last step we apply the inductive hypothesis, to obtain a closed representation independent of $\overline{\mathcal{A}}$. This representation, just before the final equality, is the newly obtained (calculated) definition of the abstracted analysis $\overline{\mathcal{D}}_\alpha$. Interestingly, the derivation is independent of the structure of the abstraction α, so this form works for any abstraction specified using our operators.

A configuration k' in the "abstract world" $\alpha(\mathbb{K}_\psi)$ can be any (compound) formula from $FeatExp$, not only a valuation formula as in the "concrete world" $\mathbb{K}_\psi$. Hence, a relation between an abstract configuration $k' \in \alpha(\mathbb{K}_\psi)$ and a presence condition $\theta \in FeatExp$ can be: k' entails θ; k' entails $\neg\theta$; or $(k' \wedge \theta)$ and $(k' \wedge \neg\theta)$ are both satisfiable. This is expressed by the following helper lemma, which is used in the derivation of "`#if` (θ) s".

Lemma 5.1. *For all abstractions $\alpha \in G^{va}, \theta \in FeatExp, \overline{a_1}, \overline{a_2} \in \mathbb{A}^{\mathbb{K}}$:*

$$
\alpha\Big(\prod_{k \in \mathbb{K}} \begin{cases} \pi_k(\overline{a_1}) \textit{ if } k \models \theta \\ \pi_k(\overline{a_2}) \textit{ if } k \not\models \theta \end{cases} \Big) \mathrel{\dot\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K})} \begin{cases} \pi_{k'}(\alpha(\overline{a_1})) & \textit{if } k' \models \theta \\ \pi_{k'}(\alpha(\overline{a_1})) \sqcup \pi_{k'}(\alpha(\overline{a_2})) & \textit{if } sat(k' \wedge \theta) \wedge sat(k' \wedge \neg\theta) \\ \pi_{k'}(\alpha(\overline{a_2})) & \textit{if } k' \models \neg\theta \end{cases} \tag{9}
$$

Proof. *By induction on the structure of α [33, App. C]. We show $\boldsymbol{\alpha}^{\text{join}}$ case.*

$$\boldsymbol{\alpha}^{\text{join}}\Big(\prod_{k\in\mathbb{K}}\begin{cases}\pi_k(\overline{a_1}) & \text{if } k\models\theta\\ \pi_k(\overline{a_2}) & \text{if } k\not\models\theta\end{cases}\Big) \;=\; \Big(\bigsqcup_{k\in\mathbb{K}}\begin{cases}\pi_k(\overline{a_1}) & \text{if } k\models\theta\\ \pi_k(\overline{a_2}) & \text{if } k\not\models\theta\end{cases}\Big)$$

(by def. of $\boldsymbol{\alpha}^{\text{join}}$)

$$= \Big(\begin{cases}\bigsqcup_{k\in\mathbb{K}}\pi_k(\overline{a_1}) & \text{if } \bigvee_{k\in\mathbb{K}}k\models\theta\\ \bigsqcup_{\{k\in\mathbb{K}\mid k\models\theta\}}\pi_k(\overline{a_1})\sqcup\bigsqcup_{\{k\in\mathbb{K}\mid k\models\neg\theta\}}\pi_k(\overline{a_2}) & \text{if } sat\,(\bigvee_{k\in\mathbb{K}}k\wedge\theta)\ \wedge\ sat(\bigvee_{k\in\mathbb{K}}k\wedge\neg\theta)\\ \bigsqcup_{k\in\mathbb{K}}\pi_k(\overline{a_2}) & \text{if } \bigvee_{k\in\mathbb{K}}k\models\neg\theta\end{cases}\Big)$$

(by def. of π_k and $\sqcup$)

$$\dot{\sqsubseteq}\Big(\begin{cases}\bigsqcup_{k\in\mathbb{K}}\pi_k(\overline{a_1}) & \text{if } \bigvee_{k\in\mathbb{K}}k\models\theta\\ \bigsqcup_{k\in\mathbb{K}}\pi_k(\overline{a_1})\sqcup\bigsqcup_{k\in\mathbb{K}}\pi_k(\overline{a_2}) & \text{if } sat\,(\bigvee_{k\in\mathbb{K}}k\wedge\theta)\ \wedge\ sat(\bigvee_{k\in\mathbb{K}}k\wedge\neg\theta)\\ \bigsqcup_{k\in\mathbb{K}}\pi_k(\overline{a_2}) & \text{if } \bigvee_{k\in\mathbb{K}}k\models\neg\theta\end{cases}\Big)$$

(by def. of π_k and $\sqcup$)

Finally, by definition of $\boldsymbol{\alpha}^{\text{join}}$ we obtain:

$$= \Big(\begin{cases}\boldsymbol{\alpha}^{\text{join}}(\overline{a_1}) & \text{if } \bigvee_{k\in\mathbb{K}}k\models\theta\\ \boldsymbol{\alpha}^{\text{join}}(\overline{a_1})\sqcup\boldsymbol{\alpha}^{\text{join}}(\overline{a_2}) & \text{if } sat\,(\bigvee_{k\in\mathbb{K}}k\wedge\theta)\ \wedge\ sat(\bigvee_{k\in\mathbb{K}}k\wedge\neg\theta)\\ \boldsymbol{\alpha}^{\text{join}}(\overline{a_2}) & \text{if } \bigvee_{k\in\mathbb{K}}k\models\neg\theta\end{cases}\Big)$$

□

Notice that we lose precision in the above derivation of "`#if` (θ) s" since we use $\dot{\sqsubseteq}$-relation instead of equality in Eq. (9), Lemma 5.1. We provide an example confirming that the above relation is not equality. Let $\mathbb{K} = \{A\wedge B, A\wedge\neg B\}$, $\overline{a_1} = ([x\mapsto 2],[x\mapsto 4])$, $\overline{a_2} = ([x\mapsto 6],[x\mapsto 2])$ and $\theta = B$. For the left-hand side (LHS) of Eq. (9), $\overline{a} = \big(\pi_{A\wedge B}(\overline{a_1}),\pi_{A\wedge\neg B}(\overline{a_2})\big)$ since $A\wedge B\models B$ and $A\wedge\neg B\not\models B$, so we have $\overline{a} = ([x\mapsto 2],[x\mapsto 2])$. Then $\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = ([x\mapsto 2])$ is the result of the LHS of Eq. (9). On the other hand, for the right-hand side (RHS) of Eq. (9), we have $k' = (A\wedge B)\vee(A\wedge\neg B)\in\boldsymbol{\alpha}^{\text{join}}(\mathbb{K})$, and so $k'\wedge B$ and $k'\wedge\neg B$ are both satisfiable. In this way, the second case of the RHS of Eq. (9) is taken for k', so we have that $\boldsymbol{\alpha}^{\text{join}}(\overline{a_1})\sqcup\boldsymbol{\alpha}^{\text{join}}(\overline{a_2}) = ([x\mapsto\top])\sqcup([x\mapsto\top]) = ([x\mapsto\top])$ is the result of the RHS of Eq. (9).

We now present derivational steps for the most illustrative case for expressions:

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}'}[\![e_0 \oplus e_1]\!] \circ \gamma)(\overline{d}) \\
&= \alpha\big(\prod_{k \in \mathbb{K}_\psi} \pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d})) \mathbin{\widehat{\oplus}} \pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))\big) && \text{(by def. of } \circ\text{, and } \overline{\mathcal{A}'} \text{ in Fig. 2)} \\
&= \alpha\big(\prod_{k \in \mathbb{K}_\psi} \pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d}) \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))\big) && \text{(by def. of } \pi_k \text{ and } \dot{\widehat{\oplus}}\text{)} \\
&= \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\alpha\big(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d}) \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d})\big)) && \text{(by def. of } \alpha\text{)} \\
&\dot{\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\alpha(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d})) \mathbin{\dot{\widehat{\oplus}}} \alpha(\overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))) && \text{(by Lemma 5.2 below)} \\
&\dot{\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d} \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d}) && \text{(by IH, twice)} \\
&\dot{\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d}) \mathbin{\widehat{\oplus}} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d}) = \overline{\mathcal{D}'}_\alpha[\![e_0 \oplus e_1]\!]\overline{d} && \text{(by } \pi_{k'} \text{ and } \dot{\widehat{\oplus}}\text{)}
\end{aligned}
$$

In the derivation, we use the following helper lemma.

Lemma 5.2. *For all abstractions* $\alpha \in G^{va}, \overline{v_1}, \overline{v_2} \in Const^{\mathbb{K}}$*:*

$$\alpha(\overline{v_1} \mathbin{\dot{\widehat{\oplus}}} \overline{v_2}) \mathbin{\dot{\sqsubseteq}} \alpha(\overline{v_1}) \mathbin{\dot{\widehat{\oplus}}} \alpha(\overline{v_2}) \tag{10}$$

Proof. *By induction on the structure of* α*. We only consider the most involved case for* $\boldsymbol{\alpha}^{\text{join}}$*. The other cases can be found in [33, App. C].*

$$
\begin{aligned}
\boldsymbol{\alpha}^{\text{join}}(\overline{v_1} \mathbin{\dot{\widehat{\oplus}}} \overline{v_2}) &= \textstyle\bigsqcup_{k \in \mathbb{K}} \pi_k(\overline{v_1} \mathbin{\dot{\widehat{\oplus}}} \overline{v_2}) && \text{(by def. of } \boldsymbol{\alpha}^{\text{join}}\text{)} \\
&= \textstyle\bigsqcup_{k \in \mathbb{K}} \big(\pi_k(\overline{v_1}) \mathbin{\widehat{\oplus}} \pi_k(\overline{v_2})\big) && \text{(by def. of } \pi_k \text{ and } \dot{\widehat{\oplus}}\text{)} \\
&\dot{\sqsubseteq} \big(\textstyle\bigsqcup_{k \in \mathbb{K}} \pi_k(\overline{v_1})\big) \mathbin{\widehat{\oplus}} \big(\textstyle\bigsqcup_{k \in \mathbb{K}} \pi_k(\overline{v_2})\big) && \text{(by def. of } \textstyle\bigsqcup \text{ and } \widehat{\oplus}\text{)} \\
&= \boldsymbol{\alpha}^{\text{join}}(\overline{v_1}) \mathbin{\widehat{\oplus}} \boldsymbol{\alpha}^{\text{join}}(\overline{v_2}) && \text{(by def. of } \boldsymbol{\alpha}^{\text{join}}\text{)}
\end{aligned}
$$

□

Again, we lose precision in the above derivation of $e_0 \oplus e_1$ due to the $\dot{\sqsubseteq}$-relation in Eq. (10), Lemma 5.2. We provide an example confirming the need for approximating from above. Let $\overline{v_1} = (5, 2)$, $\overline{v_2} = (2, 5)$, and $\oplus = +$. Then $\boldsymbol{\alpha}^{\text{join}}((5,2) \mathbin{\dot{\widehat{+}}} (2,5)) = \boldsymbol{\alpha}^{\text{join}}((7,7)) = 7$ is the result of the LHS of Eq. (10). On the other hand, $\boldsymbol{\alpha}^{\text{join}}((5,2)) = \top$, $\boldsymbol{\alpha}^{\text{join}}((2,5)) = \top$, and $\top \mathbin{\widehat{+}} \top = \top$ is the result of the RHS of Eq. (10).

The derivations for other cases are similar and can be found in [33, App. B]. The process results in the definitions of $\overline{\mathcal{D}}_\alpha[\![s]\!]$ and $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ presented in Figure 3. Observe that the definitions of $\overline{\mathcal{D}}_\alpha$ and $\overline{\mathcal{D}'}_\alpha$ are identical to the definitions of $\overline{\mathcal{A}}$ and $\overline{\mathcal{A}'}$ except for the case of the preprocessor statement "`#if` (θ) s". This is expected since variability abstractions only affect the configuration-specific

$$
\begin{array}{l}
\overline{\mathcal{D}}_\alpha[\![\texttt{skip}]\!] = \lambda\overline{d}.\,\overline{d} \\
\overline{\mathcal{D}}_\alpha[\![\texttt{x := } e]\!] = \lambda\overline{d}.\,\prod_{k'\in\alpha(\mathbb{K}_\psi)}(\pi_{k'}(\overline{d}))[\texttt{x} \mapsto \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e]\!]\overline{d})] \\
\overline{\mathcal{D}}_\alpha[\![s_0 \texttt{ ; } s_1]\!] = \overline{\mathcal{D}}_\alpha[\![s_1]\!] \circ \overline{\mathcal{D}}_\alpha[\![s_0]\!] \\
\overline{\mathcal{D}}_\alpha[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] = \lambda\overline{d}.\,\overline{\mathcal{D}}_\alpha[\![s_0]\!]\overline{d} \dot{\sqcup} \overline{\mathcal{D}}_\alpha[\![s_1]\!]\overline{d} \\
\overline{\mathcal{D}}_\alpha[\![\texttt{while } e \texttt{ do } s]\!] = \mathrm{lfp}\lambda\overline{\Phi}.\,\lambda\overline{d}.\,\overline{d} \dot{\sqcup} \overline{\Phi}(\overline{\mathcal{D}}_\alpha[\![s]\!]\,\overline{d}) \\
\overline{\mathcal{D}}_\alpha[\![\texttt{\#if } (\theta)\ s]\!] = \lambda\overline{d}.\,\prod_{k'\in\alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } k' \models \theta \\ \pi_{k'}(\overline{d}) \sqcup \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \\ \pi_{k'}(\overline{d}) & \text{if } k' \models \neg\theta \end{cases} \\
\overline{\mathcal{D}'}_\alpha[\![n]\!] = \lambda\overline{d}.\,\prod_{k'\in\alpha(\mathbb{K}_\psi)} \mathtt{n} \\
\overline{\mathcal{D}'}_\alpha[\![\mathtt{x}]\!] = \lambda\overline{d}.\,\prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{d})(\mathtt{x}) \\
\overline{\mathcal{D}'}_\alpha[\![e_0 \oplus e_1]\!] = \lambda\overline{d}.\,\prod_{k'\in\alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d}) \mathbin{\widehat{\oplus}} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d})
\end{array}
$$

Figure 3: Definitions of $\overline{\mathcal{D}}_\alpha[\![\overline{s}]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K}_\psi)}$ and $\overline{\mathcal{D}'}_\alpha[\![\overline{e}]\!] : (\mathbb{A} \to \mathit{Const})^{\alpha(\mathbb{K}_\psi)}$.

aspect of the lifted analysis. This observation is the basis for defining abstractions as source-to-source transformations in Section 6.

Soundness of the abstracted analysis follows by construction; more precisely the complete calculation constitutes an inductive proof of the theorem:

Theorem 5.1 (Soundness of Abstracted Analysis). *We have that:*

(i) $\forall e \in \mathit{Exp}, (\alpha,\gamma) \in G^{va}, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} :\ \alpha \circ \overline{\mathcal{A}'}[\![e]\!] \circ \gamma(\overline{d}) \mathbin{\dot{\sqsubseteq}} \overline{\mathcal{D}'}_\alpha[\![e]\!]\,\overline{d}$

(ii) $\forall s \in \mathit{Stm}, (\alpha,\gamma) \in G^{va}, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} :\ \alpha \circ \overline{\mathcal{A}}\,[\![s]\!] \circ \gamma(\overline{d}) \mathbin{\dot{\sqsubseteq}} \overline{\mathcal{D}}\,_\alpha[\![s]\!]\,\overline{d}$

Theorem 5.2 (Monotonicity of Abstracted Analysis). *For all $s \in \mathit{Stm}$ and $e \in \mathit{Exp}$, $\overline{\mathcal{D}}_\alpha[\![s]\!]$ and $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ are monotone functions.*

Proof. *By structural induction on s and e [33, App. D].* □

Example 5.1. *Consider the program S_1 from Example 3.2, with $\mathbb{K}_{A\vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. We calculate $\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![S_1]\!]$ for $\boldsymbol{\alpha}_A^{\text{join-proj}} = \boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_A^{\text{proj}}$. Note that $\boldsymbol{\alpha}_A^{\text{join-proj}}(\mathbb{K}_\psi)$ has only one valid configuration $k' = (A \wedge B) \vee (A \wedge \neg B)$. Following the rules of Figure 3, we obtain the following confounded abstract execution of all configurations containing the feature A:*

$$
\Big(\overbrace{[\boldsymbol{x} \mapsto \top]}^{k'}\Big) \overset{\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\boldsymbol{x}:=0]\!]}{\longmapsto} \big([\boldsymbol{x} \mapsto 0]\big) \overset{\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\#\text{if } (A)\ \boldsymbol{x}:=\boldsymbol{x}+1]\!]}{\longmapsto} \big([\boldsymbol{x} \mapsto 1]\big)
$$
$$
\overset{\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\#\text{if } (B)\ \boldsymbol{x}:=1]\!]}{\longmapsto} \big([\boldsymbol{x} \mapsto 1]\big)
$$

Note that $k' \models A$, so we have $\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\texttt{\#if}(A)\ \boldsymbol{x} := \boldsymbol{x}+1]\!] = \overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\boldsymbol{x} := \boldsymbol{x}+1]\!]$. In the last step, we use $\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\texttt{\#if}(B)\ \boldsymbol{x} := 1]\!]([\boldsymbol{x} \mapsto 1]) = ([\boldsymbol{x} \mapsto 1]) \dot{\sqcup} \overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![\boldsymbol{x} := 1]\!]([\boldsymbol{x} \mapsto 1])$ since $k' \wedge B$ and $k' \wedge \neg B$ are both satisfiable. The final result shows that the value of $\boldsymbol{x}$ is the constant 1 for every configuration that

satisfies A. Similarly, we can calculate that $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}[\![S_1]\!]([\boldsymbol{x} \mapsto \top]) = ([\boldsymbol{x} \mapsto \top])$, *and* $\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}} \otimes \boldsymbol{\alpha}_{\neg A}^{\text{proj}}}[\![S_1]\!]([\boldsymbol{x} \mapsto \top], [\boldsymbol{x} \mapsto \top]) = ([\boldsymbol{x} \mapsto 1], [\boldsymbol{x} \mapsto 1])$.

On the other hand, for the program S_2 *we obtain* $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}[\![S_2]\!]([\boldsymbol{x} \mapsto \top]) = ([\boldsymbol{x} \mapsto \top])$ *and* $\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}[\![S_2]\!]([\boldsymbol{x} \mapsto \top]) = ([\boldsymbol{x} \mapsto \top])$, *so the value of* x *is lost (approximated) by* $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}$ *and* $\overline{\mathcal{D}}_{\boldsymbol{\alpha}_A^{\text{join-proj}}}$. □

6. Syntactic Transformations

The analyses $\overline{\mathcal{A}}$ and $\overline{\mathcal{D}}_\alpha$ can be implemented either directly by using definitions of Figs. 2 and 3, or by extracting the corresponding data-flow equations. An entirely different way to implement $\overline{\mathcal{D}}_\alpha$ is to execute the abstraction on the source program family, before running the analysis, and then running the previously existing analysis $\overline{\mathcal{A}}$ on this transformed program family. We take this route as it allows to completely reuse the effort invested in designing and implementing $\overline{\mathcal{A}}$.

Any $\overline{\text{IMP}}$ program s with sets of features $\mathbb{F}$ and valid configurations $\mathbb{K}$ is translated into a corresponding abstract program $\alpha(s)$ with a set of features $\alpha(\mathbb{F})$ and a set of valid configurations $\alpha(\mathbb{K})$. We will call this transformation `reconfigurator`. The transformation is defined recursively over the structure of α. The function α copies all basic statements of $\overline{\text{IMP}}$, and recursively calls itself for all sub-statements of compound statements other than `#if`. For example, $\alpha(\texttt{skip}) = \texttt{skip}$ and $\alpha(s_0 \texttt{ ; } s_1) = \alpha(s_0) \texttt{ ; } \alpha(s_1)$. We discuss the rewrites for `#if` statements below.

In the rewrite, we associate a fresh feature name $Z \notin \mathbb{F}$, with every join abstraction $\boldsymbol{\alpha}^{\text{join}}$ (consequently written $\boldsymbol{\alpha}_{Z,}^{\text{join}}$). The new feature Z is an abstract name (renaming) of the compound formula $\bigvee_{k \in \mathbb{K}} k$. It denotes the single valid configuration obtained from $\boldsymbol{\alpha}^{\text{join}}$. The new feature name is used to simplify conditions in the transformed code. The $\boldsymbol{\alpha}_{Z,}^{\text{join}}$ rewrite is defined as follows:

$$\boldsymbol{\alpha}_{Z,}^{\text{join}}(\mathbb{F}) = \{Z\}, \quad \boldsymbol{\alpha}_{Z,}^{\text{join}}(\mathbb{K}) = \{Z\}$$

$$\boldsymbol{\alpha}_{Z,}^{\text{join}}(\texttt{\#if } (\theta)\ s) = \begin{cases} \texttt{\#if } (Z)\ \boldsymbol{\alpha}_{Z,}^{\text{join}}(s) & \text{if } \bigvee_{k \in \mathbb{K}} k \models \theta \\ \texttt{\#if } (Z)\ \texttt{lub}(\boldsymbol{\alpha}_{Z,}^{\text{join}}(s), \texttt{skip}) & \text{if sat}(\bigvee_{k \in \mathbb{K}} k \wedge \theta) \wedge \text{sat}(\bigvee_{k \in \mathbb{K}} k \wedge \neg\theta) \\ \texttt{\#if } (\neg Z)\ \boldsymbol{\alpha}_{Z,}^{\text{join}}(s) & \text{if } \bigvee_{k \in \mathbb{K}} k \models \neg\theta \end{cases}$$

where the newly introduced statement $\texttt{lub}(s_0, s_1)$ represents the least upper bound (join) of the results obtained by analysing s_0 and s_1, i.e. a non-deterministic choice between executing s_0 or s_1. This is the only language-dependent aspect of `reconfigurator`. It can have different implementations depending on the programming language and the applied analysis. In our case, we exploit the definition of $\overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]$ (cf. Figure 2), and use $\texttt{lub}(s_0, s_1) = \texttt{if } (*) \texttt{ then } s_0 \texttt{ else } s_1$ where $*$ denotes any integer. The result of applying transformation $\boldsymbol{\alpha}_{Z,}^{\text{join}}$ to a program family s is a program family $\boldsymbol{\alpha}_{Z,}^{\text{join}}(s)$ which contains only one valid product where the feature Z is enabled. Hence, $\boldsymbol{\alpha}_{Z,}^{\text{join}}(s)$ is a single program and it can be analyzed with existing single-program analyses. In this way, $\boldsymbol{\alpha}_{Z,}^{\text{join}}$ enables performing family-based analyses using

single-program analyzers, albeit with loss of precision. To see that $\boldsymbol{\alpha}^{\text{join}}_{Z,'}(s)$ is a single program, observe that in $\boldsymbol{\alpha}^{\text{join}}_{Z,'}(\texttt{\#if}\ (\theta)\ s)$ we have that $\texttt{\#if}\ (\neg Z)\ \boldsymbol{\alpha}^{\text{join}}_{Z,'}(s)$ is equivalent to $\texttt{skip}$, and $\texttt{\#if}\ (Z)\ \boldsymbol{\alpha}^{\text{join}}_{Z,'}(s)$ (resp., $\texttt{\#if}\ (Z)\ \texttt{lub}(\boldsymbol{\alpha}^{\text{join}}_{Z,'}(s), \texttt{skip})$) is equivalent to $\boldsymbol{\alpha}^{\text{join}}_{Z,'}(s)$ (resp., $\texttt{lub}(\boldsymbol{\alpha}^{\text{join}}_{Z,'}(s), \texttt{skip})$), since the feature Z is enabled in the only valid product. However it is useful to keep the above statements in $\boldsymbol{\alpha}^{\text{join}}_{Z,'}(\texttt{\#if}\ (\theta)\ s)$, which makes it easy to merge abstract programs when we use compound abstractions.

The rewrite for projection only changes the set of legal configurations:

$$\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{F}) = \mathbb{F}, \quad \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K}) = \{k \in \mathbb{K} \mid k \models \varphi\}, \quad \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\texttt{\#if}\,(\theta)\ s) = \texttt{\#if}\,(\theta)\ \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(s)$$

Note that the general scheme for the basic rewrites of `#if` statements can be summarized as

$$\alpha(\texttt{\#if}\ (\theta)\ s) = \texttt{\#if}\ (\overline{\alpha}(\theta))\ \overline{\alpha}(s, \theta)$$

where $\overline{\alpha}$ is a function transforming the presence condition θ and the statement s. It is easy to extract $\overline{\alpha}(\theta)$ and $\overline{\alpha}(s, \theta)$ from the above rewrites for $\boldsymbol{\alpha}^{\text{join}}_{Z,'}$ and $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}$. For example, we have $\overline{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}(\theta) = \theta$ and $\overline{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}(s, \theta) = \boldsymbol{\alpha}^{\text{proj}}_{\varphi}(s)$. We will use them in defining transformations for composite abstraction operators.

Now, for the case of parallel composition $\alpha_1 \otimes \alpha_2$, recall from Section 4 that the set $(\alpha_1 \otimes \alpha_2)(\mathbb{K})$ is the union of $\alpha_1(\mathbb{K})$ and $\alpha_2(\mathbb{K})$. However now in the rewrite semantics, we are sometimes modifying the set of features. If $\alpha_1(\mathbb{F}) \neq \alpha_2(\mathbb{F})$ then some of valid configurations in $\alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$ will not assign truth values to all features in $\alpha_1(\mathbb{F}) \cup \alpha_2(\mathbb{F})$. To take a meaningful union of configurations, we need to first unify their alphabets. To achieve this, each valid configuration can be extended by information that the missing features are excluded from it (negated). Now the rewrite rules for parallel composition are as follows:

$$(\alpha_1 \otimes \alpha_2)(\mathbb{F}) = \alpha_1(\mathbb{F}) \cup \alpha_2(\mathbb{F})$$

$$(\alpha_1 \otimes \alpha_2)(\mathbb{K}) = \{k_1 \textstyle\bigwedge_{A \in \alpha_2(\mathbb{F}) \setminus \alpha_1(\mathbb{F})} \neg A \mid k_1 \in \alpha_1(\mathbb{K})\} \cup \{k_2 \textstyle\bigwedge_{A \in \alpha_1(\mathbb{F}) \setminus \alpha_2(\mathbb{F})} \neg A \mid k_2 \in \alpha_2(\mathbb{K})\}$$

$$(\alpha_1 \otimes \alpha_2)(\texttt{\#if}\ (\theta)\ s) = \begin{cases} \texttt{\#if}\ \left(\overline{\alpha_1}(\theta) \vee \overline{\alpha_2}(\theta)\right)\ \overline{\alpha_1}(s, \theta) & \text{if } \overline{\alpha_1}(s, \theta) = \overline{\alpha_2}(s, \theta) \\ \alpha_1(\texttt{\#if}\ (\theta)\ s); \alpha_2(\texttt{\#if}\ (\theta)\ s) & \text{otherwise} \end{cases}$$

Observe that the second case of the parallel composition $\alpha_1 \otimes \alpha_2$ transformation can only appear if the second case of $\boldsymbol{\alpha}^{\text{join}}_{Z,'}$ transformation has been used somewhere in recursive rewriting of s (perhaps deep). This is because only in this case $\overline{\alpha}(s, \theta)$ is not equal to $\alpha(s)$, i.e. we have $\overline{\boldsymbol{\alpha}^{\text{join}}_{Z,'}}(s, \theta) = \texttt{lub}(\boldsymbol{\alpha}^{\text{join}}_{Z,'}(s), \texttt{skip})$, and so $\overline{\alpha_1}(s, \theta) \neq \overline{\alpha_2}(s, \theta)$. All the other rewrites leave s intact, i.e. $\overline{\alpha_1}(s, \theta) = \overline{\alpha_2}(s, \theta)$ and so the first case of $\alpha_1 \otimes \alpha_2$ transformation is taken. However, in case when $\overline{\alpha_1}(s, \theta) \neq \overline{\alpha_2}(s, \theta)$, the branches have disjoint feature alphabets, as every join is using a fresh feature name as parameter. This ensures that only one of the sequenced copies of s, $\overline{\alpha_1}(s, \theta)$ and $\overline{\alpha_2}(s, \theta)$, will actually be executed (and the other will amount to skip) in any given configuration of the product.

For sequential composition $\alpha_2 \circ \alpha_1$, we use the following rewrites:

$$(\alpha_2 \circ \alpha_1)(\mathbb{F}) = \alpha_2(\alpha_1(\mathbb{F})), \quad (\alpha_2 \circ \alpha_1)(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$$

$$(\alpha_2 \circ \alpha_1)(\texttt{\#if}\ (\theta)\ s) = \texttt{\#if}\ \left(\overline{\alpha_2}(\overline{\alpha_1}(\theta))\right)\ \overline{\alpha_2}(\overline{\alpha_1}(s, \theta), \overline{\alpha_1}(\theta))$$

Example 6.1. *Consider the program* S_1'*:*

$$\texttt{\#if}(A)\ \boldsymbol{x} := \boldsymbol{x}+1; \texttt{\#if}\,(B)\ \boldsymbol{x} := 1$$

with $\mathbb{F} = \{A, B\}$*,* $\psi = A \vee B$*, and* $\mathbb{K}_{A\vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$*. Then*

$$\boldsymbol{\alpha}^{\text{join}}_{\lhd Z} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(S_1') = \texttt{\#if}\ (Z)\ \boldsymbol{x} := \boldsymbol{x}+1; \texttt{\#if}\ (Z)\ \texttt{lub}(\boldsymbol{x} := 1, \texttt{skip}) \qquad (*)$$

The set of valid configurations after projection is changed to $\{A\wedge B, A\wedge\neg B\}$*, and after join again to just* $\{Z\}$*, where* Z *represents the formula* $(A \wedge B) \vee (A \wedge \neg B)$*. The obtained program has only one configuration, the one that satisfies* Z*. The projection does not change the statements of the program. However, the join rewrite simplifies the first* `#if` *(it is statically determined; cf. the first case of* $\boldsymbol{\alpha}^{\text{join}}_{\lhd Z}$*, transformation), and joins the second statement with* `skip` *as it is unknown whether it will be executed or not, in the lack of information about the assignment to* B *in the abstracted program (cf. the second case of* $\boldsymbol{\alpha}^{\text{join}}_{\lhd Z}$*, transformation). Note that since* Z *is the only one valid configuration, the obtained program is equivalent to:* $\boldsymbol{x} := \boldsymbol{x}+1$; $\texttt{lub}(\boldsymbol{x} := 1, \texttt{skip})$*. Similarly, we can calculate:*

$$\boldsymbol{\alpha}^{\text{join}}_{\lhd Z} \circ \boldsymbol{\alpha}^{\text{proj}}_{B}(S_1') = \texttt{\#if}\ (Z)\ \texttt{lub}(\boldsymbol{x} := \boldsymbol{x}+1, \texttt{skip}); \texttt{\#if}\ (Z)\ \boldsymbol{x} := 1$$

which is equivalent to $\texttt{lub}(\boldsymbol{x} := \boldsymbol{x}+1, \texttt{skip}); \boldsymbol{x} := 1$*.*

Now consider $((\boldsymbol{\alpha}^{\text{join}}_{\lhd Z} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{B})(S_1')$*. The new set of features is* $\{Z, A, B\}$*. The subset {A,B} is retained from the right projection component, and {Z} comes from the left join-project component. After extending the configurations of both components with negations of absent feature names we get the following set of valid configurations:* $\mathbb{K}' = \{Z \wedge \neg A \wedge \neg B, \neg Z \wedge A \wedge B, \neg Z \wedge \neg A \wedge B\}$*. The result of the left join-project operand is the program (*), and the right rewrite (projection) never changes the statements, so its result is identical to* S_1'*. Thus we are composing programs (*) and* S_1' *using the parallel composition rewrites. Then,* $((\boldsymbol{\alpha}^{\text{join}}_{\lhd Z} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}) \otimes \boldsymbol{\alpha}^{\text{proj}}_{B})(S_1')$ *is as follows:*

$$\texttt{\#if}\ (Z \vee A)\ \boldsymbol{x} := \boldsymbol{x}+1; \texttt{\#if}\ (Z)\ \texttt{lub}(\boldsymbol{x} := 1, \texttt{skip}); \texttt{\#if}\ (B)\ \boldsymbol{x} := 1$$

The first `#if` *has been unified using the first case of the transformation for* $\otimes$*, and the second* `#if` *is transformed into two copies of the statement with different guards, using the second case of the rewrite definition for* $\otimes$*. For any legal configuration in* $\mathbb{K}'$ *at most one of them does not reduce to* `skip`*.* □

The transformation for $\boldsymbol{\alpha}^{\text{fignore}}_{A}$ can be inferred from transformations for the basic abstractions (see Theorem 4.4). However, that transformation is very complex since it requires applying $\boldsymbol{\alpha}^{\text{join}}$ rewrites for all valid configurations and then parallel compositions on the obtained results, which can be exponentially many rewrites in the worst case. We can give more direct and compact definition of the rewrite rules for $\boldsymbol{\alpha}^{\text{fignore}}_{A}$. Similarly as in $\boldsymbol{\alpha}^{\text{join}}$, we rename the sets of feature names and valid configurations. Any compound formula $\bigvee_{k\in\mathbb{K}, k\setminus_A \equiv k'} k$ denoting a valid configuration in $\overline{\mathcal{D}}_\alpha$ is now renamed to k'. Thus, k' represents an abstract name of the compound formula $\bigvee_{k\in\mathbb{K}, k\setminus_A \equiv k'} k$. Now, we have:

$$\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{F}) = \mathbb{F}\setminus\{A\}, \quad \boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}) = \{k' \equiv k\setminus_A \mid k \in \mathbb{K}\}$$

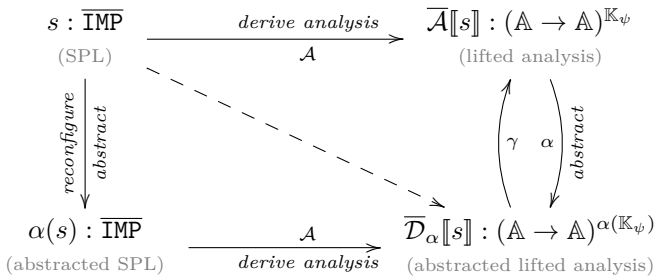

Figure 4: Illustration of *derive* vs *abstract*: $\overline{\mathcal{D}}_\alpha[\![s]\!] = \overline{\mathcal{A}}[\![\alpha(s)]\!]$.

To simplify the notation for the rewrite rules, we note that for each presence condition $\theta \in \mathit{FeatExp}$ the set of valid configurations $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})$ can be partitioned into three sets: K^θ_{true} containing all k' that entail θ; K^θ_{false} containing all k' that entail $\neg\theta$; and $K^\theta_?$ containing all the other k' from $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K})$. More formally, we have:

$$K^\theta_{\text{true}} = \{k' \in \boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) \mid \bigvee_{k\in\mathbb{K}, k\backslash_A \equiv k'} k \models \theta\}$$

$$K^\theta_{?} = \{k' \in \boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) \mid \text{sat}(\bigvee_{k\in\mathbb{K}, k\backslash_A \equiv k'} k \wedge \theta) \wedge \text{sat}(\bigvee_{k\in\mathbb{K}, k\backslash_A \equiv k'} k \wedge \neg\theta)\}$$

$$K^\theta_{\text{false}} = \{k' \in \boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) \mid \bigvee_{k\in\mathbb{K}, k\backslash_A \equiv k'} k \models \neg\theta\}$$

Now, the rewrite for "`#if` (θ) s" is:

$$\boldsymbol{\alpha}_A^{\text{fignore}}(\texttt{\#if } (\theta)\ s) = \begin{cases} \texttt{\#if } (\theta)\ \boldsymbol{\alpha}_A^{\text{fignore}}(s) & \text{if } \theta \equiv \theta\backslash_A \\ \texttt{\#if } (\theta\backslash_A)\ \{\texttt{\#if } (\bigvee_{k'\in K^\theta_{\text{true}}} k')\ \boldsymbol{\alpha}_A^{\text{fignore}}(s); & \\ \quad \texttt{\#if } (\bigvee_{k'\in K^\theta_{?}} k')\ \texttt{lub}(\boldsymbol{\alpha}_A^{\text{fignore}}(s), \texttt{skip})\ \} & \text{otherwise} \end{cases}$$

Note that $A\backslash_A \equiv \neg A\backslash_A \equiv \text{true}$. Again, it is easy to extract $\overline{\alpha}(\theta)$ and $\overline{\alpha}(s,\theta)$ from the above rewrites for $\boldsymbol{\alpha}_A^{\text{fignore}}$. Thus, we can use $\boldsymbol{\alpha}_A^{\text{fignore}}$ to build compound abstractions by using composition operators.

Example 6.2. *Consider the program S'_1 from Example 6.1:*

$$\texttt{\#if}(A)\ \boldsymbol{x} := \boldsymbol{x} + 1; \texttt{\#if}\,(B)\ \boldsymbol{x} := 1$$

with $\mathbb{F} = \{A, B\}$ and $\mathbb{K}_{A\vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Now, we have $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}) = \{B\}$ and $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}_\psi) = \{B, \neg B\}$. The program $\boldsymbol{\alpha}_A^{\text{fignore}}(S'_1)$ is:

$$\texttt{\#if}\ (\mathit{true})\ \{\texttt{\#if}\ (B)\ \texttt{lub}(\boldsymbol{x} := \boldsymbol{x} + 1, \texttt{skip}); \texttt{\#if}\ (\neg B)\ \boldsymbol{x} := \boldsymbol{x} + 1\};$$
$$\texttt{\#if}\ (B)\ \boldsymbol{x} := 1$$

The first `#if` from S'_1 is rewritten using the second case of the transformation rewrite for $\boldsymbol{\alpha}_A^{\text{fignore}}$, since $\theta = A$ and $\theta\backslash_A \equiv \mathit{true}$. We have $B \in K^A_?$ since

$((A \wedge B) \vee (\neg A \wedge B)) \wedge A$ *and* $((A \wedge B) \vee (\neg A \wedge B)) \wedge \neg A$ *are both satisfiable. Also* $\neg B \in K^A_{true}$ *since* $A \wedge \neg B \models A$. *The second* `#if` *from* S_1' *is transformed using the first case of the transformation rewrite for* $\boldsymbol{\alpha}_A^{\text{fignore}}$, *since* $B \backslash_A \equiv B$. $\square$

Now the analysis $\overline{\mathcal{A}}[\![\alpha(s)]\!]$ and $\overline{\mathcal{D}}_\alpha[\![s]\!]$ coincide up to renaming of valid configurations. So the source-to-source translator, `reconfigurator`, together with an existing implementation of $\overline{\mathcal{A}}$ gives us the abstracted analysis $\overline{\mathcal{D}}_\alpha$. The equality is illustrated by Figure 4. That means that we can effectively compute $\overline{\mathcal{D}}_\alpha$ on a program s by running the lifted analysis $\overline{\mathcal{A}}$ on the transformed abstract program $\alpha(s)$. The correctness of rewrite rules for calculating $\alpha(s)$ is proved by the following theorem.

Theorem 6.1.
$\forall s \in Stm, \alpha : \mathbb{A}^{\mathbb{K}_\psi} \to \mathbb{A}^{\alpha(\mathbb{K}_\psi)} \in G^{va}, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} :\ \overline{\mathcal{D}}_\alpha[\![s]\!]\,\overline{d} = \overline{\mathcal{A}}[\![\alpha(s)]\!]\,\overline{d}$.

Proof. *By induction on the structure of* $\alpha \in G^{va}$ *and* $s \in Stm$. *Apart from the* `#if`*-statement, for all other statements the proof is an immediate result of definitions of* $\overline{\mathcal{D}}_\alpha$, $\overline{\mathcal{A}}$, *and* $\alpha(s)$.

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

$$\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}[\![\texttt{\#if}\ (\theta)\ s]\!]\overline{d} \qquad \text{(set of feat. is } \mathbb{F}\text{, set of configs. is } \mathbb{K}_\psi)$$

$$= \begin{cases} \overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}[\![s]\!]\overline{d} & \text{if } \bigvee_{k\in\mathbb{K}_\psi} k \models \theta \\ \overline{d} \,\dot{\sqcup}\, \overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}[\![s]\!]\overline{d} & \text{if } sat(\bigvee_{k\in\mathbb{K}_\psi} k \wedge \theta) \wedge sat(\bigvee_{k\in\mathbb{K}_\psi} k \wedge \neg\theta) \\ \overline{d} & \text{if } \bigvee_{k\in\mathbb{K}_\psi} k \models \neg\theta \end{cases}$$

(by def. of $\overline{\mathcal{D}}_\alpha$)

$$= \begin{cases} \overline{\mathcal{A}}[\![\boldsymbol{\alpha}'^{\text{join}}_{Z'}(s)]\!]\overline{d} & \text{if } \bigvee_{k\in\mathbb{K}_\psi} k \models \theta \\ \overline{d} \,\dot{\sqcup}\, \overline{\mathcal{A}}[\![\boldsymbol{\alpha}'^{\text{join}}_{Z'}(s)]\!]\overline{d} & \text{if } sat(\bigvee_{k\in\mathbb{K}_\psi} k \wedge \theta) \wedge sat(\bigvee_{k\in\mathbb{K}_\psi} k \wedge \neg\theta) \\ \overline{d} & \text{if } \bigvee_{k\in\mathbb{K}_\psi} k \models \neg\theta \end{cases} \qquad \text{(by IH)}$$

$$= \begin{cases} \overline{\mathcal{A}}[\![\texttt{\#if}\ (Z)\ \boldsymbol{\alpha}'^{\text{join}}_{Z'}(s)]\!]\overline{d} & \text{if } \bigvee_{k\in\mathbb{K}_\psi} k \models \theta \\ \overline{\mathcal{A}}[\![\texttt{\#if}\ (Z)\ \texttt{lub}(\boldsymbol{\alpha}'^{\text{join}}_{Z'}(s), \texttt{skip})]\!]\overline{d} & \text{if } sat(\bigvee_{k\in\mathbb{K}_\psi} k \wedge \theta) \wedge sat(\bigvee_{k\in\mathbb{K}_\psi} k \wedge \neg\theta) \\ \overline{\mathcal{A}}[\![\texttt{\#if}\ (\neg Z)\ \boldsymbol{\alpha}'^{\text{join}}_{Z'}(s)]\!]\overline{d} & \text{if } \bigvee_{k\in\mathbb{K}_\psi} k \models \neg\theta \end{cases}$$

(by def. of $\overline{\mathcal{A}}$; renaming: set of feat. is $\{Z\}$, set of configs. is $\{Z\}$)

$$= \overline{\mathcal{A}}[\![\boldsymbol{\alpha}'^{\text{join}}_{Z'}(\texttt{\#if}\ (\theta)\ s)]\!]\overline{d} \qquad \text{(by def. of } \overline{\mathcal{A}} \text{ and } \boldsymbol{\alpha}'^{\text{join}}_{Z'}(\texttt{\#if}\ (\theta)\ s))$$

The other cases are similar [33, App. F]. $\square$

Example 6.3. *Consider the program* S_1 *from Example 3.2 with* $\mathbb{K}_{A\vee B} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. *We have calculated in Example 5.1 that* $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_A^{\text{proj}}}[\![S_1]\!]([\boldsymbol{x} \mapsto \top]) = ([\boldsymbol{x} \mapsto 1])$. *We now calculate* $\overline{\mathcal{A}}[\![\boldsymbol{\alpha}'^{\text{join}}_{Z} \circ \boldsymbol{\alpha}_A^{\text{proj}}(S_1)]\!]([\boldsymbol{x} \mapsto \top])$*:*

$$\big([\boldsymbol{x}\mapsto\top]\big) \overset{\overline{\mathcal{A}}[\![\boldsymbol{x}:=0]\!]}{\longmapsto} \big([\boldsymbol{x}\mapsto 0]\big) \overset{\overline{\mathcal{A}}[\![\#\text{if}\ (Z)\ \boldsymbol{x}:=\boldsymbol{x}+1]\!]}{\longmapsto} \big([\boldsymbol{x}\mapsto 1]\big) \overset{\overline{\mathcal{A}}[\![\#\text{if}\ (Z)\ \texttt{lub}(\boldsymbol{x}:=1, \texttt{skip})]\!]}{\longmapsto} \big([\boldsymbol{x}\mapsto 1]\big)$$

Hence, we obtain that $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}_A^{\text{proj}}}[\![S_1]\!]([\boldsymbol{x} \mapsto \top])$ *and* $\overline{\mathcal{A}}[\![\boldsymbol{\alpha}_Z^{\text{join}} \circ \boldsymbol{\alpha}_A^{\text{proj}}(S_1)]\!]([\boldsymbol{x} \mapsto \top])$ *are equal.* □

7. A Note On Generalizations

So far, we have introduced variability abstractions and we have presented how they can be used for deriving sound abstracted lifted analysis for constant propagation (Theorems 5.1 and 6.1). Here, we show how the proposed methodology can be generalized and applied to any other (monotone) analysis phrased in the abstract interpretation framework [32, 23].

A single-program analysis is specified by the following data. A complete lattice $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}} \rangle$ for describing the properties of the analysis. A domain $\mathbb{A} = \mathit{Var} \to \mathbb{P}$ of abstract stores which associates properties from $\mathbb{P}$ with the variables of the program. The analysis domain $\langle \mathbb{A}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ inherits the lattice structure from the property domain $\mathbb{P}$ in a point-wise manner. There are also transfer functions for expressions $\mathcal{A}'[\![e]\!] : \mathbb{A} \to \mathbb{P}$ and for statements $\mathcal{A}[\![s]\!] : \mathbb{A} \to \mathbb{A}$.

By using variational abstract interpretation [11], we can lift any single-program analysis to the corresponding family-based analysis, which is specified as follows. Given a set of valid configurations $\mathbb{K}_\psi$, the lifted domain is $\langle \mathbb{A}^{\mathbb{K}_\psi}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, which inherits the lattice structure of $\langle \mathbb{A}, \sqsubseteq \rangle$ in a component-wise manner. There are also transfer functions for expressions $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\mathbb{K}_\psi}$ and for statements $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}_\psi}$. Functions $\overline{\mathcal{A}'}[\![e]\!]$ and $\overline{\mathcal{A}}[\![s]\!]$ for most of the cases are defined as in Fig. 2. There are two exceptions. First, we need a way of turning values into properties, which is specified by a function: $\mathit{abs}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{P}$. Thus, the rule for constants n becomes: $\overline{\mathcal{A}'}[\![n]\!] = \lambda \overline{a}. \prod_{k \in \mathbb{K}_\psi} \mathit{abs}_{\mathbb{Z}}(\mathtt{n})$. Second, 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 $\mathbb{P}$. Note that for the constant propagation analysis the function $\mathit{abs}_{\mathbb{Z}}$ is the identity, i.e, $\mathit{abs}_{\mathbb{Z}}(n) = n$ for $n \in \mathbb{Z}$, whereas operators $\widehat{\oplus}$ are defined in Eq. (2).

Example 7.1. *The detection sign analysis is based on the property domain:* $\langle \mathit{Sign}, \sqsubseteq_S \rangle$*, where* $\mathit{Sign} = \{\bot, \top\} \cup \{$`-, +, 0, -\0, 0\+, -\+`$\}$*, and partial ordering* $\sqsubseteq_S$ *is defined as [23]:* `0` $\sqsubseteq_S$ `-\0`*,* `0` $\sqsubseteq_S$ `0\+`*, etc. The three basic properties of values are:* `+` *which indicates a positive value;* `0` *which denotes the value zero; and* `-` *which indicates a negative value. The other properties are obtained by combining the basic ones during the analysis. For example,* `0\+` *means that a value is always zero or positive. The property of the constant* n *is determined by:*

$$\mathit{abs}_{\mathbb{Z}}(n) = \begin{cases} \texttt{+} & \text{if } n > 0 \\ \texttt{0} & \text{if } n = 0 \\ \texttt{-} & \text{if } n < 0 \end{cases}$$

For each operator $\oplus$*, we have the corresponding operator* $\widehat{\oplus}$ *defined on properties from Sign [23]. For example, we have* (`+`)$\widehat{+}$(`-`) $= \top$*,* (`+`)$\widehat{+}$(`0`) $=$ `+`*, etc.* □

Now, we want to derive generalized definitions for $\overline{\mathcal{D}'}_\alpha[\![e]\!] : (\mathbb{A} \to \mathbb{P})^{\alpha(\mathbb{K}_\psi)}$ and $\overline{\mathcal{D}}_\alpha[\![s]\!] : (\mathbb{A} \to \mathbb{A})^{\alpha(\mathbb{K}_\psi)}$, that will hold for any analysis not only for constant propagation as in Fig. 3. For this aim, we need to repeat derivations for all language constructs described in Section 5 for constant propagation. However, all derivations, except those for constants n and binary operations $e_0 \oplus e_1$, are the same as before since their lifted analysis functions $\overline{\mathcal{A}'}[\![e]\!]$ and $\overline{\mathcal{A}}[\![s]\!]$ are as in Fig. 2. So, for those constructs we obtain the same definitions as in Fig. 3.

We now consider derivation steps of $\overline{\mathcal{D}'}_\alpha$ for constants n and binary operations $e_0 \oplus e_1$ for any analysis. First, by induction on the structure of α, we can show that: $\alpha(\prod_{k\in\mathbb{K}} abs_\mathbb{Z}(\mathtt{n})) = \prod_{k'\in\alpha(\mathbb{K})} abs_\mathbb{Z}(\mathtt{n})$ for any $\alpha \in Abs$. Then we obtain $\overline{\mathcal{D}'}_\alpha[\![\mathtt{n}]\!]\overline{d} = \prod_{k'\in\alpha(\mathbb{K}_\psi)} abs_\mathbb{Z}(\mathtt{n})$ by the following derivation:

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}'}[\![n]\!] \circ \gamma)(\overline{d}) = \alpha(\overline{\mathcal{A}'}[\![n]\!](\gamma(\overline{d}))) && \text{(by def. of } \circ\text{)}\\
&= \alpha(\prod_{k\in\mathbb{K}_\psi} abs_\mathbb{Z}(\mathtt{n})) = \prod_{k'\in\alpha(\mathbb{K}_\psi)} abs_\mathbb{Z}(\mathtt{n}) = \overline{\mathcal{D}'}_\alpha[\![n]\!]\overline{d}\\
& && \text{(by def. of } \overline{\mathcal{A}'}[\![n]\!] \text{ and above observation)}
\end{aligned}
$$

Derivation for $\overline{\mathcal{D}'}_\alpha[\![e_0 \oplus e_1]\!]$ proceeds as for constant propagation in Section 5, but now we need to show that the result corresponding to Lemma 5.2 holds for any vectors $\overline{v_1}, \overline{v_2} \in \mathbb{P}^{\mathbb{K}\psi}$ and any $\alpha \in Abs$, i.e. $\alpha(\overline{v_1} \mathbin{\hat{\oplus}} \overline{v_2}) \mathbin{\dot{\sqsubseteq}} \alpha(\overline{v_1}) \mathbin{\hat{\oplus}} \alpha(\overline{v_2})$. This can be proved by induction on the structure of α similarly as in Lemma 5.2, by using the properties of $\sqcup$ and $\widehat{\oplus}$ defined on values from $\mathbb{P}$.

Again, the soundness of the general abstracted analysis (i.e. the result corresponding to Theorem 5.1) follows by construction. Following the same proof methodology, we can prove that the result corresponding to Theorem 6.1 holds for any lifted analysis phrased in the abstract interpretation framework.

8. Evaluation

We now evaluate our technique for speeding up lifted analyses using variability abstractions on several case studies. The evaluation aims to show that we can use abstracted lifted analyses to successfully analyze realistic program families. To do so, we ask the following research questions:

RQ1: How efficient are the abstracted lifted analyses compared to the other approaches for analyzing program families, such as lifted analyses and lifted analyses with sharing?

RQ2: Can the abstracted lifted analyses turn some previously infeasible lifted analyses of program families into feasible ones?

RQ3: How precise are the abstracted lifted analyses? In particular, can we practically use the obtained results from abstracted lifted analyses?

8.1. Experimental Setup

For our experiments, we use an existing implementation of lifted data-flow analyses for Java SPLs [9]. The implementation is based on SOOT's intra-procedural data-flow analysis framework [34] for analyzing Java programs. It uses the Eclipse plug-in CIDE (Colored IDE) [35] to annotate statements using background colors rather than `#ifdef` directives. Every feature is thus associated with a unique color. The experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU with 8 GB memory. All times are reported as average over ten runs with the highest and lowest number removed. The implementation, benchmarks, and all results obtained from our experiments are available from: `https://aleksdimovski.github.io/var-abs.html`.

Client analyses. For our experiment, we have chosen three client analyses: *reaching definitions*, *uninitialized variables*, and *interval analysis*, for which we derived the corresponding definitions of abstracted lifted analysis. The reaching definitions analysis computes for every program point those assignments that may have defined the current values of variables. The analysis domain is the powerset of all assignments occurring in the method. The uninitialized variables analysis computes for every program point the set of variables that is possibly uninitialized, thus the analysis domain is the powerset of variables occurring in the method. The interval analysis computes for every variable a lower and an upper bound for its possible values at each program point. The basic properties are of the form $[l, h]$, where $l \in \mathbb{Z}_{32} \cup \{-\infty\}$, $h \in \mathbb{Z}_{32} \cup \{+\infty\}$, $l \leq h$, and $\mathbb{Z}_{32}$ is the finite set of 32-bit integers. The coarsest property is $\top = [-\infty, +\infty]$, whereas $\bot$ is the empty interval. Since the interval analysis is defined over finite 32-bit integers, the property domain has finite height. However, we still use a widening operator to accelerate the convergence of the least fixed points (lfp), since a widening can lead to an over-approximation of the lfp in a few steps while classical Kleene iterations may converge to the lfp after a long time. We have implemented the so-called delayed widening by counting the times a node was visited, and then once a given threshold of visiting a state has been reached we apply a widening operator. The widening operator w works relatively to a fixed finite subset B of integers which includes 0, $-\infty$, $+\infty$ and all constants that occur in the method to be analyzed [30, 23]. We use the following definition: $w([l, h]) = [max\{i \in B \mid i \leq l\}, min\{i \in B \mid i \geq h\}]$.

Types of lifted analyses. We will consider an unoptimized lifted intra-procedural analysis, known as $\mathcal{A}2$ (from [9]), that uses $|\mathbb{K}_\psi|$-sized tuples of analysis information, one analysis value per configuration. Also, we consider $\mathcal{A}3$ (from [9]) which is the same lifted analysis as $\mathcal{A}2$, but with improved representation via sharing analysis-equivalent configurations using a high-performance bit vector library [36]. So $\mathcal{A}3$ is an optimized version of $\mathcal{A}2$ where shared representation is used for representing sets of configurations (i.e. components of tuples) with equivalent analysis information. For example, the result of applying a lifted (constant propagation) analysis with sharing, $\mathcal{A}3$, to a simple program family

with configurations $\mathbb{K}_{true} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$ is:

$$\big([\![\text{true}]\!] \mapsto [\mathtt{x} \mapsto 0]\big) \overset{\overline{\mathcal{A}}[\![\#\mathsf{if}\,(A)\,\mathtt{x}{+}{+}]\!]}{\longmapsto} \big([\![A]\!] \mapsto [\mathtt{x} \mapsto 1], [\![\neg A]\!] \mapsto [\mathtt{x} \mapsto 0]\big) \overset{\overline{\mathcal{A}}[\![\#\mathsf{if}\,(B)\,\mathtt{x}{+}{+}]\!]}{\longmapsto}$$
$$\big([\![A \wedge B]\!] \mapsto [\mathtt{x} \mapsto 2], [\![(A \wedge \neg B) \vee (\neg A \wedge B)]\!] \mapsto [\mathtt{x} \mapsto 1], [\![\neg A \wedge \neg B]\!] \mapsto [\mathtt{x} \mapsto 0]\big)$$

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, that is $[\mathtt{x} \mapsto 0]$, associated with them. Then as the flow of control passes `#if` statements the configuration space is slowly split up into more and more equivalence classes, i.e. sets of analysis-equivalent configurations (e.g. $[\![A]\!] = \{A \wedge B, A \wedge \neg B\}$, $[\![\neg A]\!] = \{\neg A \wedge B, \neg A \wedge \neg B\}$). Note that $\mathcal{A}2$ corresponds to $\overline{\mathcal{A}}$ in Figure 2 and we will refer to it as $\overline{\mathcal{A}}$, while we will use $\overline{\mathcal{S}}$ for the analysis with sharing.

The performance of abstracted analyses depends on the size of tuples they work on. Therefore as variability abstractions, we have chosen $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}}$ which joins together (confounds) information from all configurations down to just one abstracted analysis value, and $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{proj}}_{N/2} \otimes \boldsymbol{\alpha}^{\text{join}}_{N/2}}$ (where $N = |\mathbb{K}_\psi|$) which is a parallel composition of a projection of 1/2 (randomly selected) configurations and a *join* of the remaining 1/2 configurations. We abbreviate them as $\overline{\mathcal{D}}_1$ and $\overline{\mathcal{D}}_{N/2}$ in the following. In our experiments, we will also use an optimized version of $\overline{\mathcal{D}}_{N/2}$ with sharing, which will be referred to as $\overline{\mathcal{S}}_{N/2}$. We have chosen those variability abstractions because they represent the coarsest abstraction $\overline{\mathcal{D}}_1$ that works on 1-sized tuples, and the medium abstraction $\overline{\mathcal{D}}_{N/2}$ that works on $N/2$-sized tuples. Any other abstraction will have a speed up anywhere between $\overline{\mathcal{A}}$ (no abstraction, which works on N-sized tuples), $\overline{\mathcal{D}}_{N/2}$ (medium abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction). It thus quantifies the potential of abstractions.

Solution of an analysis. Each analysis uses a control flow graph (CFG), in which nodes correspond to program points and edges represent possible flow of control, and a lattice with finite height, 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.

Benchmarks. We use three SPL benchmarks [35]: Graph PL (GPL) is a small desktop application with intensive feature usage; Prevayler is a slightly larger product line with low feature usage; and BerkelyDB is a larger database library with moderate feature usage. Table 1 summarises relevant characteristics for each benchmark: the average number of valid configurations in all methods in the SPL, the total number of features in the entire SPL, the total number of lines of code (LOC). Also, for each SPL, the figure details information about the method with the highest variability (most configurations): its number of valid configurations, features, and lines of code.

8.2. Performance

Figure 5 shows the time it takes to run the methods with maximum variability: `Prevayler::publisher()`, `BerkeleyDB::main()`, `GPL::display()`, as

Table 1: Characteristics of our three SPL benchmarks (average #configurations N in all methods in SPL, total #features, and LOC) along with, for each SPL, its method with maximum variability (#configurations N, local #features, and LOC).

Benchmark	avg. $\lvert\mathbb{K}_\psi\rvert$	$\lvert\mathbb{F}\rvert$	LOC	max var. method	$\lvert\mathbb{K}_\psi\rvert$	$\lvert\mathbb{F}\rvert$	LOC
`GPL`	3.9	18	1,350	`Vertex.display()`	106	9	31
`BerkelyDB`	1.6	42	84,000	`DBRunAction.main()`	40	7	165
`Prevayler`	1.3	5	8,000	`P'F'.publisher()`	8	3	10

a relative comparison between $\overline{\mathcal{A}}$ (baseline) and $\overline{\mathcal{S}}$ (baseline with sharing) *vs* $\overline{\mathcal{D}}_{N/2}$ (medium abstraction), $\overline{\mathcal{S}}_{N/2}$ (medium abstraction with sharing) and $\overline{\mathcal{D}}_1$ (maximum abstraction). Note that for interval analysis we evaluate `Prevayler::read()` with $N = 2$ configurations, since the method `publisher()` does not have integer variables. For each benchmark method, we give the speed up factor relative to the baseline (normalized with factor 1) and the number of configurations, N.

Our experiment confirms previous results that sharing is indeed effective and especially so for larger values of N [9]. On our methods, it translates to speed ups (i.e. $\overline{\mathcal{A}}$ *vs* $\overline{\mathcal{S}}$) anywhere between 3% faster for N=8 and slightly more than three times faster for N=106. We also observe that abstraction is not surprisingly significantly faster than unabstracted analyses (i.e. $\overline{\mathcal{D}}_{N/2}$ and $\overline{\mathcal{D}}_1$ *vs* $\overline{\mathcal{A}}$ and $\overline{\mathcal{S}}$); i.e. abstraction yields significant performance gains, especially for benchmarks with higher variability. For `GPL` with N=106, we see a dramatic 72, 47 and 28 times speed up depending on the analysis (i.e. $\overline{\mathcal{D}}_1$ *vs* $\overline{\mathcal{A}}$). Also, we note that increased abstraction is up to 26 times faster than improved representation (i.e. $\overline{\mathcal{D}}_1$ *vs* $\overline{\mathcal{S}}$). In general, it is obviously possible to combine the benefits from representation and abstraction to yield even more efficient analyses. For example, we observe that medium abstraction with sharing $\overline{\mathcal{S}}_{N/2}$ is always faster than the corresponding version without sharing $\overline{\mathcal{D}}_{N/2}$. On the other hand, since $\overline{\mathcal{D}}_1$ collapses information onto only *one* abstracted value, sharing will, of course, not help for this abstraction (addresses **RQ1**).

8.3. Taming Combinatorial Blow-up of Configurations

For very large values of $N = |\mathbb{K}_\psi|$, the lifted analysis (even with sharing) may become impractically slow or even infeasible since the properties and transfer functions are N-sized tuples. In that case, we can use variability abstractions to reduce the configuration space, and thus obtain an approximate, but feasible (faster) lifted analysis.

As an experiment, we have tested the limits of family-based analysis $\overline{\mathcal{A}}$. We took a large method, `processFile()` from `BerkeleyDB`, and we have gradually added unconstrained variability into it. This was done by adding optional features and by sequentially composing `#if` statements guarded by all existing features. Already for N=2^{13}=8,192 configurations, the analysis $\overline{\mathcal{A}}$ took 138 seconds. For N=2^{14}=16,384, it ran more than ten minutes until it eventually produced an out-of-memory error. In contrast, variability abstraction $\overline{\mathcal{D}}_1$ analyses the same high variability method in less than 8 ms (albeit less precisely). Hence, abstraction

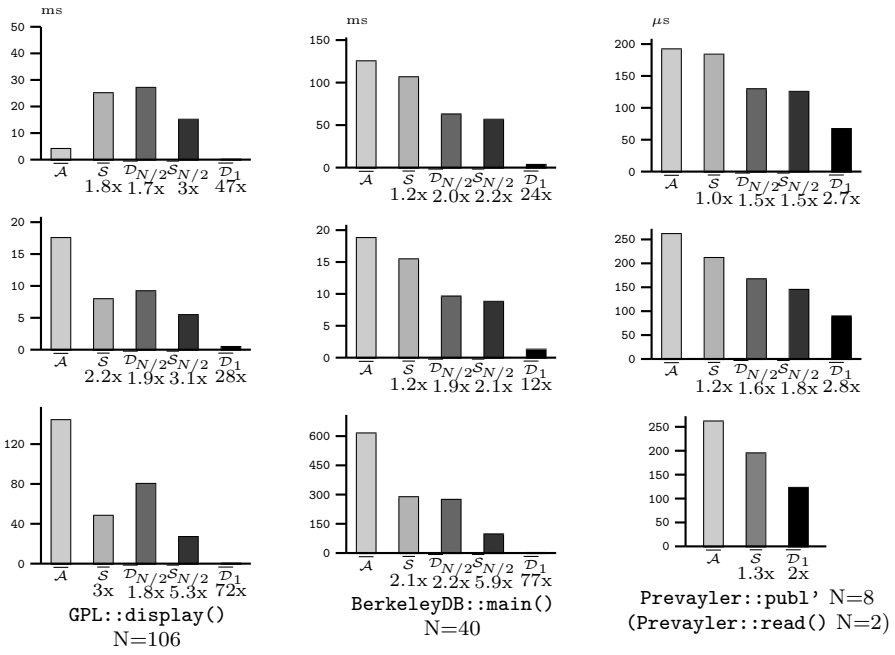

Figure 5: Analysis time for *reaching definitions* (above), *uninitialized variables* (middle) and *interval analysis* (below): $\overline{\mathcal{A}}$ (baseline) and $\overline{\mathcal{S}}$ (baseline with sharing) vs. $\overline{\mathcal{D}}_{N/2}$ (medium abstraction), $\overline{\mathcal{S}}_{N/2}$ (medium abstraction with sharing) and $\overline{\mathcal{D}}_1$ (maximum abstraction).

can not only speed up analyses, but also turn previously infeasible analyses feasible (addresses **RQ2**).

In fact, since the maximum (join) abstraction $\overline{\mathcal{D}}_1$ works on 1-size tuples, we observe that its analysis time is quite close to the single program analysis that runs on only one valid product from a family. This is illustrated in Figure 6, where for methods with the highest number of configurations we show the running times of the maximum abstraction $\overline{\mathcal{D}}_1$ and the average duration of all single program analyses that take one valid product from an SPL at a time (addresses **RQ1** and **RQ2**).

8.4. Precision-Speed Tradeoff

Figure 7 illustrates the tradeoff between *precision* and *speed* as a function of increasingly coarser abstractions. Information loss is quantified by the percentage of nodes in the CFG for which the solution of a given abstracted analysis is the *same* as the baseline analysis $\overline{\mathcal{A}}$. More specifically, we measure the percentage of nodes for which an abstracted analysis (e.g. $\overline{\mathcal{D}}_1$) accurately calculates their analysis results, such that the same analysis results are obtained with the full lifted analysis $\overline{\mathcal{A}}$. Note that once a solution for a node is found by an abstracted analysis, we use the corresponding concretization function to calculate a solution for each configuration at that node. As a baseline normalized with speed and precision loss factor 1, we take the analysis $\overline{\mathcal{A}}$ with no abstraction, and hence no

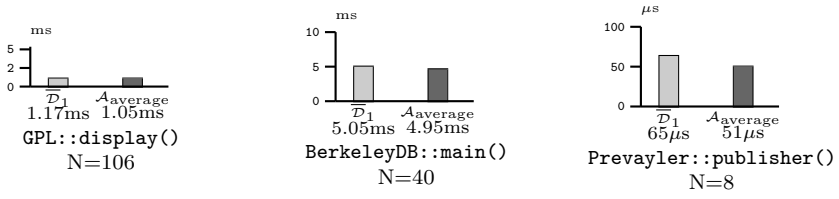

Figure 6: Performance comparison for reaching definitions analysis: maximum abstraction $\overline{\mathcal{D}}_1$ vs. the average time for single program analysis $\mathcal{A}_{\text{average}}$ for methods with the highest number of configurations.

precision loss, which works with N-sized tuples, one per configuration. At the other extreme, we plot the speed up and precision loss factor for the maximum abstraction $\overline{\mathcal{D}}_1$ that works on 1-sized tuples. In between, we manufacture interpolations of the two by abstractions that respectively work with $\frac{3}{4}$N-, $\frac{1}{2}$N-, and $\frac{1}{4}$N-sized tuples. For instance, we express the first of these, $\overline{\mathcal{D}}_{\frac{3}{4}N}$, as the *parallel composition* of a *projection* of 3/4 (randomly selected) configurations and a *join* of the remaining 1/4 configurations.

Note that we use entire SPLs (aggregate of analyzing all methods), rather than individual methods, because it gives a "smoother profile" (snapshot) of the tradeoff. Obviously, for lower values of N (e.g., `Prevayler`), we get less extreme speed ups and precision loss, but the overall picture looks the same as for the maximum variability methods (although more continuous; i.e. less threshold-like).

We see that, as expected, both speed up and precision loss increase with N and abstraction. Interestingly, we see vertical curves where further abstraction will increase the speed almost without compromising precision. For example, the vertical line from $\overline{\mathcal{D}}_{N/2}$ to $\overline{\mathcal{D}}_{N/4}$ means that we have only speed ups with no precision loss. Note that this phenomenon also occurs when considering individual methods. We take this as an *indication* that there exists an optimal abstraction which may be more useful and finer abstractions may not pay off (addresses **RQ3**).

For example, for interval analysis we obtain the following results. For `GPL`, we analyze 19 methods that contain 33 integer variables occurring in 22,500 CFG nodes (we consider all possible configurations). The baseline $\overline{\mathcal{A}}$ runs in 231.6 ms reporting the exact intervals for 15,203 nodes, and there is a precision loss for 7,297 nodes for which the top value $\top$ is obtained. On the other hand, $\overline{\mathcal{D}}_{N/2}$ runs in 131.1 ms (1.7x speed up) giving accurate results for 9,868 nodes (35% loss); whereas the coarsest abstraction $\overline{\mathcal{D}}_1$ runs in 26.6 ms (8.7x speed up) reporting the exact intervals for 4,306 nodes (71.6% loss). For `BerkeleyDB`, we analyze 1,268 methods that contain 174 integer variables occurring in 1,020,352 CFG nodes in total. We obtain the following results: $\overline{\mathcal{A}}$ runs in 9,718 ms reporting the accurate intervals for 687,799 nodes; $\overline{\mathcal{D}}_{N/2}$ runs in 2,856 ms (3.4x speed up) with correct results for 668,914 nodes (2.7% loss); and $\overline{\mathcal{D}}_1$ runs in 1,789 ms (5.4x speed up) reporting the exact intervals for 599,968 nodes (12.7% loss) (addresses **RQ3**).

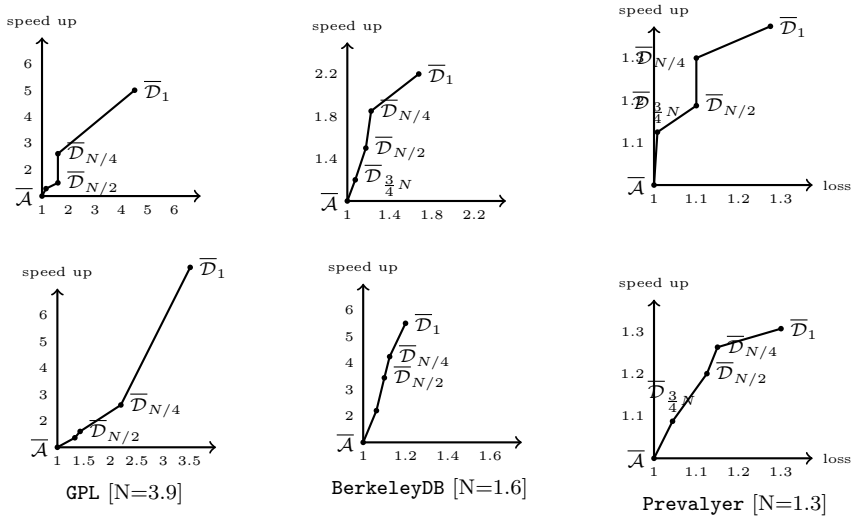

Figure 7: Tradeoff for reaching definitions (above) and interval analysis (below): *precision loss* (x-axis) -vs- *speed up* (y-axis) for various analyses with increasing abstraction.

8.5. Application Scenarios

We now present several interesting application scenarios of using variability abstractions to efficiently analyse either an entire family or just a single method by using reaching definitions and interval analysis.

Entire SPL. GPL is a family of classical graph applications with variability on its representation and algorithms. For instance, the features `Directed` and `Undirected` control whether or not graphs are *directed*; `Weighted` and `Unweighted` control whether or not the graphs are *weighted*; and, the features `BFS` and `DFS` control the search algorithm used (*breadth-first search* or *depth-first search*). It is common industrial practice, to ship products with a subset of configurations, and thereby functionality. Here, we may use projection to *disable* features `BFS` and `Undirected`, along with any features that only work on undirected graphs: (`Connected`, `MSTKruskal`, and `MSTPrim` for implementing *connected components* and *minimum spanning trees* algorithms) which can be obtained from GPL's feature model, detailing such feature dependencies. With this projection (abstraction), the configuration space of GPL is reduced from 528 to 370 valid configurations. This, in turn, cuts analysis time of reaching definitions in half (from 90ms to 49ms). For 123 out of 135 methods, the abstracted analysis computes the exact same analysis information. For larger product lines and projections, lots of time may be saved in this way (addresses **RQ1** and **RQ3**).

Reaching definitions. Figure 8a shows a fragment taken from `BerkeleyDB`'s `main()` method with N=40 valid configurations. A local variable, `doAction`,

```
void main(..) {
1   ..  int doAction = 0; ..
2   #ifdef Cleaner
3   if (..)  doAction = CLEAN;
4   #endif
5   #ifdef INCompresser
6   if (..)  doAction = COMPRESS;
7   #endif
8   if (..)  doAction = CHECKPOINT;
9   #ifdef Statistics
10  if (..)  doAction = DBSTATS;
11  #endif
12  ..switch (doAction) {..}..
}
```

(a) Code extracted from `BerkeleyDB::main()` [N=40].

```
void main(..) {
1   ..  int numEdges = 10, j=0;
2   int[] startVertices = new int[numEdges];
3   #ifdef Prog
4   for (int i=0; i<numEdges; i++) {
5       startVertices[i]=i;
6       #ifdef Transpose
7           j--;
8       #endif
9   }
10  #endif
11  ..startVertices[j]=0;..
}
```

(b) Code extracted from `GPL::main()` [N=4].

Figure 8: Application scenarios for reaching definitions and interval analysis.

is defined and initialized to zero, after which it is conditionally assigned three times in statements guarded by `#ifdef`s. (Actually, there are two more similar `#ifdef`s involving features `Evictor` and `DeleteOp`, but we have omitted those for brevity in the code fragment.) We can use a join abstraction of the reaching definitions analysis to compute what are the possible values (definitions) that *reach* the condition of the `switch` statement in line 12. An abstracted analysis would be able to determine that these are the assignments in lines 1, 3, 6, 8, and 10, by analyzing only *one* crudely over-approximated configuration instead of all (N=40) configurations. In general, by inspecting the structure of the code and the features used, we can tailor abstactions that can analyze individual methods much faster than analyzing all configurations (addresses **RQ3**).

Interval analysis. Figure 8b shows a (slightly modified) fragment extracted from `GPL`'s `main()` method with N=4 configurations. First, local variables, `numEdges` and `j`, and an array, `startVertices`, are defined and initialized. Then, 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 11 in order to check if there is an array out-of-bounds access. If the features `Prog` and `Transpose` are both on, then the widening operator will make `j` mapped to $[-\infty, 0]$ at line 11. That is found, for example, by visiting 2 times the node at line 7 and calculating $w([-2, 0]) = [-\infty, 0]$. Thus, we have an array out-of-bound error at line 11, since the range of indexes for `startVertices` is $[0, 9]$. However, if we use the abstraction $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\neg\texttt{Transpose}}$, we obtain that `j` is mapped to $[0, 0]$ at line 11, so there is no array out-of-bound error for configurations that satisfy $\neg$`Transpose` (addresses **RQ3**).

8.6. Discussion

We are now ready to answer the research questions **RQ1**, **RQ2**, and **RQ3** for our approach. The abstracted lifted analyses outperform the other competitive (unabstracted) approaches for analyzing program families. We observe significant

performance gains (up to maximal 72 times speed up), especially for coarser abstractions and programs with higher variability (**RQ1**). The abstracted lifted analyses can turn some previously infeasible lifted analyses of program families into feasible ones. We have synthetically generated a program with huge configuration space ($2^{14} = 16,384$), such that the lifted analysis is infeasible for it but the maximum abstraction analysis runs in only 8 ms (**RQ2**). We can find suitable abstracted lifted analyses for which the precision-speed tradeoff is acceptable, i.e. the speed up of lifted analyses is significant while the precision loss is small. The optimal abstraction is very useful in practice enabling us to efficiently and precisely analyze various program families (**RQ3**).

Threats to validity.. We perform intra-procedural data-flow analysis of relatively small methods. We have not evaluated our approach for larger real files using inter-procedural data-flow analysis. However, the focus of variability abstractions is to combat the configuration space blow-up of program families, not their size. So we expect to obtain similar or even better results for larger programs.

9. Related Work

We divide our discussion of related work into three categories: comparison with standard static analysis, lifted static analysis and other lifted techniques.

Comparison with standard static analysis. The standard off-the-shelf static analyzers based on abstract interpretation [37, 38] can be also used to analyze program families. In this case, we can use the brute-force strategy to perform $|\mathbb{K}_\psi|$ analyses in order to analyze all variants one by one. On the other hand, the lifted static analyzer $\overline{\mathcal{A}}$ performs one analysis on $|\mathbb{K}_\psi|$-sized tuples. Still, the lifted analyzer brings several advantages compared to the standard analyzers. First, off-the-shelf analyzers that use brute-force strategy have to preprocess a given program family, then build the CFG and execute the fixed point algorithm once for each variant. In contrast, the lifted analysis builds one CFG and executes the fixed point algorithm once per program family. Second, many transfer functions act identically for all (or some) valid configurations. Thus they can be executed efficiently by running them once (or several times), instead of $|\mathbb{K}_\psi|$ times. Third, lifted analysis can be improved by using sharing for representing sets of configurations with the same analysis information. Finally, as demonstrated in this work we can use variability abstractions as another way along with sharing to speed up lifted analysis, albeit with some information loss. However, very often abstracted lifted analyses provide us with sufficiently precise results. For example, consider the live variables analysis [23]. It determines which variables may be live at a program point, that is there is a path from the program point to a use of the variable that does not redefine it. Consider the program family:

$$\texttt{x := } 5; \texttt{y := } 1; \texttt{\#ifdef } (A) \texttt{ x := } 1 \texttt{ \#else x := y} + 1 \texttt{ \#endif}$$

The coarsest join abstraction will report that the variable `x` is not live at the exit from the first assignment `x := 5` for all variants. Therefore, the assignment `x := 5` is redundant and can be eliminated. So in this case, we can successfully analyze the above code using the coarsest and fastest abstraction.

Lifted static analysis. Brabrand et al. [9] show how to lift any data-flow analysis from the monotone framework. The obtained lifted data-flow analysis are much faster than ones based on $|\mathbb{K}_\psi|$ runs of the naive generate and analyze strategy. Another efficient implementation of lifted analysis formulated within the IFDS framework [39] for inter-procedural distributive environments was proposed in SPL^{LIFT} [10]. It uses binary decision diagrams to represent shared feature constraints. The authors have found that the running time of analysing all variants in a family is close to the analysis of a single-program. In such case, further benefit of applying abstraction, as presented in this paper, is unlikely to bring any significant improvement. However, notice that the method of SPL^{LIFT} is limited only to distributive data-flow analysis encoded within the IFDS framework. Many analyses, including constant propagation and interval, are not distributive and cannot be expressed in IFDS.

The formal developments in this paper are based on *variational abstract interpretation*, a formal methodology for systematic derivation of lifted analyses for `#ifdef`-based product lines, proposed in [11]. The method is based on the calculational approach to abstract interpretation of Cousot [22], applied and contextualized to product lines. In that work [11], calculations are used to derive a directly operational *lifted analysis* which is *correct* by construction. In the present paper, we assume that lifted analyses exist (possibly obtained using the methodology of [11]), and focus on abstracting variability. We devise an expressive calculus G^{va} for specifying abstraction operators. Also, all variability abstractions specifiable in our calculus are now automatically executable as source-to-source transformations. Implementing abstractions as program transformations looks similar to the framework defined in [40] for designing source-to-source program transformations by abstract interpretation of program semantics. The proposed calculus G^{va} brings new variability-aware abstractions to the calculus suggested by Cousot and Cousot in [41]. The basic operators defined here are tailored to deal with tuples. For example, the join abstraction appears in [41] as well, but it is used for abstracting set of sets. On the other hand, the join abstraction defined here is used for abstracting tuples and it is applied for deriving abstracted lifted analyses. The approach presented here is further extended in [42], where a technique for automatic generation of suitable variability abstractions is presented. It uses a pre-analysis to estimate the impact of variability-specific parts of the program family on the ultimate analysis's precision. The obtained results from running the pre-analysis are then used for constructing a suitable abstracted lifted analysis.

A good collection of analyses that have been lifted manually is presented in the survey [6]. Besides the family-based (lifted) strategy, the survey [6] identifies a *sampling strategy* as a suitable way of analyzing product lines (see also [43]). In the sampling strategy, only a random subset of products is analyzed. We

remark that once the sample is selected, our projection operator $\boldsymbol{\alpha}_{\varphi}^{\text{proj}}$ can be used to realize the sampling strategy in a simultaneous way by exploiting an existing family-based analysis. In fact, the abstraction specification framework of Section 4 allows specifying any analysis in the spectrum between a fully family-based analyses, and a single variant (*single product-based*) analysis described in [6]. We can specify abstractions that select (sample) any subsets of configurations and then analyze this subset with selected choice of precision, either all variants precisely, like in sampling, or confounding some executions for efficiency. In this sense, we show how to design analyses placed anywhere in the design spectrum painted in [6]. Consider the *feature-based* analysis strategy as an example. In this strategy an analysis explores the program code feature-by-feature (as opposed to configuration-by-configuration). Analyses following this strategy can now be systematically obtained using our abstractions, by projecting away (ignoring) all but one feature and running a single program analysis on the result. This is quite remarkable. It has been well recognized that designing such analyses is very difficult, yet now there exists a systematic way of doing that, so it is no longer an impenetrable art.

Other lifted techniques. Various lifted techniques have been proposed which lift existing single-program verification techniques to work on the level of program families. TypeChef [44] and SuperC [45] 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 [7], whereas variational lambda calculus was studied in [8]. Recently, researchers have introduced lifted model checking [12, 13] for verifying variability intensive systems. The method works at the family level and thus does not explicitly check all products one by one. In particular, transition systems enriched with features are used for compact modelling of variability-intensive systems, where system parts that vary are annotated using features. The SNIP, a specifically designed family-based model checker, is implemented for efficient verification of temporal properties of such systems. The input language to this tool is fPromela, which is a feature-aware extension of the well-known SPIN's language Promela. In [19, 20], we describe how variability abstractions can be successfully applied in the context of lifted model checking as opposed to lifted static analysis. This allows to efficiently verify some interesting properties of variability intensive systems by only a few calls to an off-the-shelf (single-system) model checker, such as SPIN. An automatic abstraction refinement procedure for family-based model checking is then proposed in [46], which works until a genuine counterexample is found or the property satisfaction is shown for all variants in the family. The application of variability abstractions for verifying real-time variational systems is described in [21]. In [14, 15], specifically designed family-based model checking algorithms are used for verifying symbolic game semantics models [47, 48] extracted from open second-order programs with `#ifdef`-s that contain undefined

components.

10. Conclusion

We have defined variability-aware abstractions given as Galois connections, and used them to derive efficient and correct-by-construction abstract analyses of program families. We have designed a calculus for the abstractions, and shown how abstractions specified in this language can be applied not only on analyses, but also on programs, obtaining a convenient implementation strategy of the abstractions in form of a source-to-source `reconfigurator` transformation.

We have proved the main results (Theorem 5.1 and Theorem 6.1) for constant propagation analysis and extracted a general proof methodology that holds for any other monotone and computable analysis that can be lifted. We have derived the abstracted definition of `#if` with the lowest precision. Improvements of the precision are possible once the analysis is known.

We evaluated the method on three Java-based product lines. We found that the abstractions improve performance of analyses independently of improvements in the data representations used in the implementations of these analyses. This indicates that the proposed abstraction strategies will be instrumental in tackling error finding analysis in large configurable software systems, like the Linux kernel. Indeed we have developed these techniques with the intention of scaling error finding tools to such challenging cases in future. Besides this, we would like to experiment with applying these abstraction techniques to alternative quality assurance methods including model checking, and testing.

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[2] T. Berger, S. She, R. Lotufo, A. Wasowski, K. Czarnecki, A study of variability models and languages in the systems software domain, IEEE Trans. Software Eng. 39 (12) (2013) 1611–1640. doi:10.1109/TSE.2013.34.
URL http://dx.doi.org/10.1109/TSE.2013.34

[3] T. Berger, R. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, S. She, Variability mechanisms in software ecosystems, Information & Software Technology 56 (11) (2014) 1520–1535. doi:10.1016/j.infsof.2014.05.005.
URL http://dx.doi.org/10.1016/j.infsof.2014.05.005

[4] H. V. Nguyen, C. Kästner, T. N. Nguyen, Exploring variability-aware execution for testing plugin-based web applications, in: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, 2014, pp. 907–918. doi:10.1145/2568225.2568300.
URL http://doi.acm.org/10.1145/2568225.2568300

[5] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: 30th International Conference on Software Engineering (ICSE 2008), ACM, 2008, pp. 311–320. doi:10.1145/1368088.1368131.
URL http://doi.acm.org/10.1145/1368088.1368131

[6] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (1) (2014) 6:1–6:45.

[7] C. Kästner, S. Apel, T. Thüm, G. Saake, Type checking annotation-based product lines, ACM Trans. Softw. Eng. Methodol. 21 (3) (2012) 14:1–14:39. doi:10.1145/2211616.2211617.
URL http://doi.acm.org/10.1145/2211616.2211617

[8] S. Chen, M. Erwig, E. Walkingshaw, An error-tolerant type system for variational lambda calculus, in: ACM SIGPLAN International Conference on Functional Programming, ICFP'12, 2012, pp. 29–40. doi:10.1145/2364527.2364535.
URL http://doi.acm.org/10.1145/2364527.2364535

[9] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, P. Borba, Intraprocedural dataflow analysis for software product lines, Trans. Aspect-Oriented Software Development 10 (2013) 73–108. doi:10.1007/978-3-642-36964-3_3.
URL https://doi.org/10.1007/978-3-642-36964-3_3

[10] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, M. Mezini, SPL$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, 2013, pp. 355–364. doi:10.1145/2491956.2491976.
URL http://doi.acm.org/10.1145/2491956.2491976

[11] J. Midtgaard, A. S. Dimovski, C. Brabrand, A. Wasowski, Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. 105 (2015) 145–170. doi:10.1016/j.scico.2014.10.002.
URL http://dx.doi:10.1016/j.scico.2015.04.005

[12] A. Classen, M. Cordy, P. Heymans, A. Legay, P. Schobbens, Model checking software product lines with SNIP, STTT 14 (5) (2012) 589–612. doi:10.1007/s10009-012-0234-1.
URL http://dx.doi.org/10.1007/s10009-012-0234-1

[13] A. Classen, P. Heymans, P. Schobbens, A. Legay, Symbolic model checking of software product lines, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, ACM, 2011, pp. 321–330. doi:10.1145/1985793.1985838.
URL http://doi.acm.org/10.1145/1985793.1985838

[14] A. S. Dimovski, Symbolic game semantics for model checking program families, in: Model Checking Software - 23rd International Symposium, SPIN 2016, Proceedings, Vol. 9641 of LNCS, Springer, 2016, pp. 19–37. doi:10.1007/978-3-319-32582-8_2.
URL https://doi.org/10.1007/978-3-319-32582-8_2

[15] A. S. Dimovski, Verifying annotated program families using symbolic game semantics, Theor. Comput. Sci. 706 (2018) 35–53. doi:10.1016/j.tcs.2017.09.029.
URL http://www.sciencedirect.com/science/article/pii/S0304397517306977

[16] A. F. Iosif-Lazar, J. Melo, A. S. Dimovski, C. Brabrand, A. Wasowski, Effective analysis of c programs by rewriting variability, Programming Journal 1 (1) (2017) 1. doi:10.22152/programming-journal.org/2017/1/1.
URL https://doi.org/10.22152/programming-journal.org/2017/1/1

[17] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, S. Apel, Variability encoding: From compile-time to load-time variability, J. Log. Algebr. Meth. Program. 85 (1) (2016) 125–145. doi:10.1016/j.jlamp.2015.06.007.
URL http://dx.doi.org/10.1016/j.jlamp.2015.06.007

[18] A. F. Iosif-Lazar, A. S. Al-Sibahi, A. S. Dimovski, J. E. Savolainen, K. Sierszecki, A. Wasowski, Experiences from designing and validating a software modernization transformation (E), in: 30th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE 2015, 2015, pp. 597–607. doi:10.1109/ASE.2015.84.
URL http://dx.doi.org/10.1109/ASE.2015.84

[19] A. S. Dimovski, A. S. Al-Sibahi, C. Brabrand, A. Wasowski, Family-based model checking without a family-based model checker, in: Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings, Vol. 9232 of LNCS, Springer, 2015, pp. 282–299. doi:10.1007/978-3-319-23404-5_18.
URL http://dx.doi.org/10.1007/978-3-319-23404-5_18

[20] A. Dimovski, A. S. Al-Sibahi, C. Brabrand, A. Wasowski, Efficient family-based model checking via variability abstractions, STTT 19 (5) (2017) 585–603. doi:10.1007/s10009-016-0425-2.
URL https://doi.org/10.1007/s10009-016-0425-2

[21] A. S. Dimovski, A. 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, Vol. 10460 of LNCS, Springer, 2017, pp. 249–268. doi:10.1007/978-3-319-63121-9_13.
URL https://doi.org/10.1007/978-3-319-63121-9_13

[22] P. Cousot, The calculational design of a generic abstract interpreter, in: M. Broy, R. Steinbrüggen (Eds.), Calculational System Design, NATO ASI Series F. IOS Press, Amsterdam, 1999, pp. 1–88.

[23] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, Secaucus, USA, 1999.

[24] A. S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions: Trading precision for speed in family-based analyses, in: 29th European Conference on Object-Oriented Programming, ECOOP 2015, Vol. 37 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 247–270. doi:10.4230/LIPIcs.ECOOP.2015.247.
URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.247

[25] J. Choi, Programming in the large for the internet of things (invited talk), in: 29th European Conference on Object-Oriented Programming, ECOOP 2015, 2015, pp. 2–2. doi:10.4230/LIPIcs.ECOOP.2015.2.
URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.2

[26] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) feasibility study, Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990).

[27] D. S. Batory, Feature models, grammars, and propositional formulas, in: Software Product Lines, 9th International Conference, SPLC 2005, Proceedings, Vol. 3714 of LNCS, Springer, 2005, pp. 7–20. doi:10.1007/11554844_3.
URL https://doi.org/10.1007/11554844_3

[28] G. Winskel, The Formal Semantics of Programming Languages, Foundation of Computing Series, The MIT Press, 1993.

[29] A. Garrido, R. E. Johnson, Refactoring C with conditional compilation, in: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada, 2003, pp. 323–326. doi:10.1109/ASE.2003.1240330.
URL http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240330

[30] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252. doi:10.1145/512950.512973.
URL http://doi.acm.org/10.1145/512950.512973

[31] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Log. Program. 13 (2–3) (1992) 103–179.

[32] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: 6th Annual ACM Symposium on Principles of Programming Languages, POPL '79, 1979, pp. 269–282. doi:10.1145/567752.567778.
URL http://doi.acm.org/10.1145/567752.567778

[33] A. S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions: Trading precision for speed in family-based analyses (extended version), CoRR abs/1503.04608 (2015) 1–50.
URL http://arxiv.org/abs/1503.04608

[34] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, V. Sundaresan, Soot - a java bytecode optimization framework, in: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, 1999, p. 13.

[35] C. Kästner, Virtual separation of concerns: Toward preprocessors 2.0, Ph.D. thesis, University of Magdeburg, Germany (May 2010).

[36] The colt project: Open source libraries for high performance scientific and technical computing in java, cERN: European Organization for Nuclear Research.
URL http://acs.lbl.gov/software/colt/

[37] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, Combination of abstractions in the astrée static analyzer, in: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Vol. 4435 of LNCS, Springer, 2006, pp. 272–300. doi:10.1007/978-3-540-77505-8_23.
URL https://doi.org/10.1007/978-3-540-77505-8_23

[38] J. Henry, D. Monniaux, M. Moy, PAGAI: A path sensitive static analyser, Electr. Notes Theor. Comput. Sci. 289 (2012) 15–25. doi:10.1016/j.entcs.2012.11.003.
URL https://doi.org/10.1016/j.entcs.2012.11.003

[39] T. W. Reps, S. Horwitz, S. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995, ACM Press, 1995, pp. 49–61. doi:10.1145/199448.199462.
URL http://doi.acm.org/10.1145/199448.199462

[40] P. Cousot, R. Cousot, Systematic design of program transformation frameworks by abstract interpretation, in: 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, 2002, pp. 178–190. doi:10.1145/503272.503290.
URL http://doi.acm.org/10.1145/503272.503290

[41] P. Cousot, R. Cousot, A galois connection calculus for abstract interpretation, in: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, 2014, pp. 3–4. doi:10.1145/2535838.2537850.
URL http://doi.acm.org/10.1145/2535838.2537850

[42] A. S. Dimovski, C. Brabrand, A. Wasowski, Finding suitable variability abstractions for family-based analysis, in: FM 2016: Formal Methods - 21st International Symposium, Proceedings, Vol. 9995 of LNCS, 2016, pp. 217–234. doi:10.1007/978-3-319-48989-6_14.
URL http://dx.doi.org/10.1007/978-3-319-48989-6_14

[43] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: case studies and experiments, in: 35th International Conference on Software Engineering, ICSE '13, IEEE Computer Society, 2013, pp. 482–491. doi:10.1109/ICSE.2013.6606594.
URL https://doi.org/10.1109/ICSE.2013.6606594

[44] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. 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, ACM, 2011, pp. 805–824. doi:10.1145/2048066.2048128.
URL http://doi.acm.org/10.1145/2048066.2048128

[45] P. Gazzillo, R. Grimm, Superc: parsing all of C by taming the preprocessor, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 2012, 2012, pp. 323–334. doi:10.1145/2254064.2254103.
URL http://doi.acm.org/10.1145/2254064.2254103

[46] A. S. Dimovski, A. Wasowski, Variability-specific abstraction refinement for family-based model checking, in: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings, Vol. 10202 of LNCS, 2017, pp. 406–423. doi:10.1007/978-3-662-54494-5_24.
URL http://dx.doi.org/10.1007/978-3-662-54494-5_24

[47] A. Dimovski, Program verification using symbolic game semantics, Theor. Comput. Sci. 560 (2014) 364–379. doi:10.1016/j.tcs.2014.01.016.
URL http://dx.doi.org/10.1016/j.tcs.2014.01.016

[48] A. S. Dimovski, Probabilistic analysis based on symbolic game semantics and model counting, in: Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Vol. 256 of EPTCS, 2017, pp. 1–15. doi:10.4204/EPTCS.256.1.
URL https://doi.org/10.4204/EPTCS.256.1