Variability Abstractions: Trading Precision for Speed in Family-Based Analyses*

Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski

IT University of Copenhagen
Denmark
{adim,brabrand,wasowski}@itu.dk

Abstract

Family-based (lifted) data-flow analysis for Software Product Lines (SPLs) is capable of analyzing all valid products (variants) without generating any of them explicitly. It takes as input only the common code base, which encodes all variants of a SPL, and produces analysis results corresponding to all variants. However, the computational cost of the lifted analysis still depends inherently on the number of variants (which is exponential in the number of features, in the worst case). For a large number of features, the lifted analysis may be too costly or even infeasible. In this paper, we introduce variability abstractions defined as Galois connections and use abstract interpretation as a formal method for the calculational-based derivation of approximate (abstracted) lifted analyses of SPL programs, which are sound by construction. Moreover, given an abstraction we define a syntactic transformation that translates any SPL program into an abstracted version of it, such that the analysis of the abstracted SPL coincides with the corresponding abstracted analysis of the original SPL. We implement the transformation in a tool, that works on Object-Oriented Java program families, and evaluate the practicality of this approach on three Java SPL benchmarks.

1998 ACM Subject Classification F.3.2 [Logics and Meanings of Programs] Semantics of Programming Languages – Program analysis

Keywords and phrases Software Product Lines, Family-Based Program Analysis, Abstract Interpretation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.247

1 Introduction and Motivation

Software Product Lines (SPLs) are an effective strategy for developing and maintaining a family of related programs. Any valid program (*variant*) of an SPL is specified in terms of features selected. A *feature* is a distinctive aspect, quality, or characteristic from the problem-domain of a system. SPLs have been adopted by the industry because of improvements in productivity and time-to-market [7]. While there are many implementation strategies, many industrial product lines are implemented using annotative approaches such as conditional compilation; in particular, via the C-preprocessor `#ifdef` construct [17].

Recently, formal analysis and verification of SPLs have been a topic of considerable research (see [23] 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 grows exponentially with the number of features, the need for efficient

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

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 247–270

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

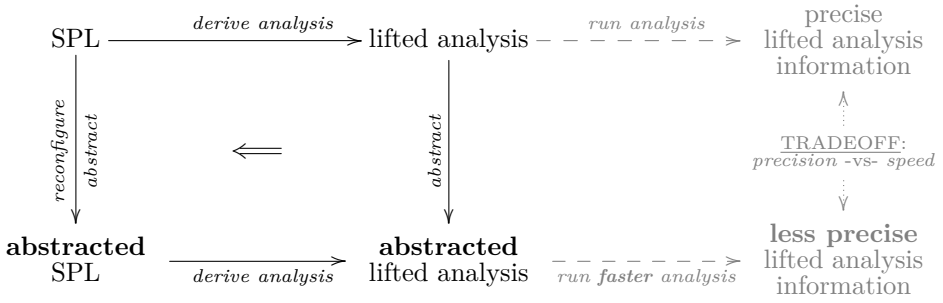


Figure 1 Diagram illustrating the role and intended usage of the `reconfigurator` transformation. Instead of abstracting an already existing (or derived) lifted analysis, our transformation allows abstraction to be applied directly to the SPL. The resulting "abstracted SPL" can then be analyzed using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

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 [18], lifted data-flow analysis [5, 4], lifted model checking [6]. They are also known as family-based (*variability-aware* or feature-sensitive) 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 capable of pin-pointing errors directly in the product line, as opposed to reporting errors in an individual product derived from the SPL.

There are two ways to speed up analyses: improving *representation* and increasing *abstraction*. The former has received considerable attention in the field of family-based analysis. In this paper, we investigate the latter. We consider a range of abstractions at the *variability level* that may tame the combinatorial explosion of configurations and reduce it to something more tractable by manipulating the configuration space of a program. Such variability abstractions enable deliberate trading of precision for speed in family-based 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, modular, and compositional declarative specification of abstractions. We propose two basic abstraction operators (*project* and *join*) and two compositional abstraction operators (*sequential composition* and *parallel composition*). Each abstraction expresses a compromise between precision and speed in the induced abstracted analysis. We show how to apply each of these abstractions to data-flow lifted analyses, to derive their corresponding efficient and sound (correct) abstracted lifted analysis based on the calculational approach of abstract interpretation developed in [9]. Moreover, we present a method for extracting data-flow equations corresponding to each abstracted analysis. Note that the approach is applicable to *any* analysis phrased as an abstract interpretation; in particular, it is not limited to data-flow analysis.

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 as input and derive a "lifted analysis" (rightward

arrow). We can then run that lifted analysis (next rightward dashed arrow) and obtain our "*precise* lifted analysis information". (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 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 analyses that are specifiable via abstract interpretation; including, but not limited to: data-flow analysis [10], model checking [12], type systems [8] and testing [15].

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 and produces an abstracted SPL program such 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 analysis. 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 paper makes the following contributions:

- **C1:** *Variability abstraction* as a method for trading precision for speed in family-based analysis (based on abstract interpretation);
- **C2:** A *calculus* for modular specification of variability abstractions;
- **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:** A stand-alone *transformation*, `reconfigurator`, based on the above ideas;
- **C5:** An *evaluation* of the above ideas; in particular, an evaluation of the tradeoff between precision and speed in family-based analyses.

We direct this work to program analysis and software engineering researchers. The method of *variability abstractions* (**C1**–**C3**) 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 data-flow analyses (e.g., to model checking, type systems, verification, and testing). The `reconfigurator` (**C4**) and the evaluation lessons (**C5**) are relevant for software engineers working on preprocessor-based product lines and who would like to speed up existing analyzers.

We proceed by introducing the basics of lifting analyses in Section 2. Section 3 defines a calculus for specification of variability abstractions. Section 4 explains how to apply an abstraction to a lifted analysis. It uses constant propagation as an example. The `reconfigurator` is described in Section 5 along with correctness for our example analysis.

Section 6 presents the evaluation on three Java Object-Oriented SPLs. Finally, we discuss the relation to other works and conclude.

2 Program Families and Lifted Analyses

In this section we summarize the prerequisites for presenting our work. We define *features*, *configurations*, *feature expressions*, and a *feature model* which designates a set of *valid* configurations. Hereafter, we describe a simple imperative language $\overline{\text{IMP}}$ for writing program families. Finally, we briefly sketch a lifted constant propagation analysis for this language, formally derived in [19]. We focus on constant propagation for presentation purposes; our approach is generically applicable to any lifted analysis phrased as an abstract interpretation.

Features, Configurations, and Feature Expressions. Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite set of *features*, each of which may be *enabled* or *disabled* in a particular program variant. A *feature expression*, *FeatExp* formula, is a propositional logic formula over $\mathbb{F}$, defined inductively by:

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

A truth assignment or *valuation* is a mapping v assigning a truth value to all features. Every feature expression evaluates to some truth value under the valuation v. 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. 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 $\varphi \not\models \theta$.

Feature Model. A feature model describes the set of *valid* configurations (variants) of a product line in terms of features and relationships among them. For our purposes a feature model can be equated to a propositional formula [2], say $\psi \in \mathit{FeatExp}$, 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 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 $\mathit{true} \cdot A = A$ and $\mathit{false} \cdot A = \neg A$, 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 corresponding 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 $\mathbb{K}_\psi = \{k_{v_1}, \ldots, k_{v_n}\}$. For example, the set of features, $\mathbb{F} = \{A, B\}$, and the feature model, $\psi = A \vee B$, yield the following set of *valid* configurations: $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$.

The Programming Language. $\overline{\text{IMP}}$ is an extension of the imperative language IMP [25] 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 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:

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

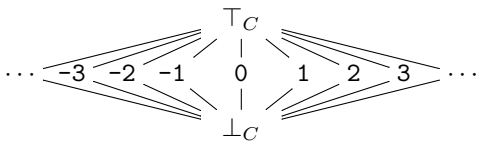
where n ranges over integers, $\mathtt{x}$ ranges over variable names *Var*, and $\oplus$ over binary arithmetic operators. The set of all generated statements s (respectively expressions e) is denoted by *Stm* (respectively *Exp*). Notice that $\overline{\text{IMP}}$ is only used for presentational purposes as a well established minimal language. Still, the introduced methodology is not limited to $\overline{\text{IMP}}$ or its features. In fact, we evaluate our approach on Object-Oriented program families written in Java.

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. All "`#if` (θ) s" statements are appropriately resolved in the generated valid product, i.e. s is included in it iff $k \models \theta$. Then, the obtained variant is executed (compiled) using the standard IMP semantics [25].

Analysis Framework. 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 (so on all the variants simultaneously).

Suppose that we have a monotone data-flow analysis framework for single programs, and we want to lift the analysis to the family level setting. Let $\langle \mathbb{X}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ be a complete lattice with finite height, which represents a property domain suitable for performing a computable analysis for single programs. By using variational abstract interpretation [19], we can derive the corresponding lifted analysis for $\overline{\text{IMP}}$, whose lifted property domain is $\langle \mathbb{X}^{\mathbb{K}_\psi}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$ where $\mathbb{K}_\psi$ is the set of valid configurations. In the following, we use constant propagation analysis as an example to demonstrate our approach for deriving computationally cheap abstracted lifted analysis. Still, the approach is by no means limited to constant propagation, but it is applicable to any static analysis from the monotone data-flow analysis framework [20] that can be lifted.

Constant Propagation Analysis. We first define a constant propagation lattice $\langle \mathit{Const}, \sqsubseteq_C \rangle$, whose partial ordering $\sqsubseteq_C$ is given by:



In this domain $\top_C$ 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, we have $0 \sqcup_C 1 = \top_C$, $\top_C \sqcap_C 1 = 1$, 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*. Thus $a(\mathtt{x})$ informs whether the variable $\mathtt{x}$ is a constant, and, in this case, what is its value. We write $\mathbb{A} = \mathit{Var} \to \mathit{Const}$ meaning 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 [25]. For example, for $a, a' \in \mathbb{A}$ we have $a \sqsubseteq_\mathbb{A} a'$ iff $\forall \mathtt{x} \in \mathit{Var}$, $a(\mathtt{x}) \sqsubseteq_C a'(\mathtt{x})$. We omit the subscripts C and $\mathbb{A}$ whenever they are clear in 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

$$
\begin{aligned}
\overline{\mathcal{A}}[\![\texttt{skip}]\!] &= \lambda\overline{a}.\,\overline{a}\\
\overline{\mathcal{A}}[\![\texttt{x := } e]\!] &= \lambda\overline{a}.\,\prod_{k\in\mathbb{K}_\psi}(\pi_k(\overline{a}))[\texttt{x}\mapsto \pi_k(\overline{\mathcal{A}'}[\![e]\!]\overline{a})]\\
\overline{\mathcal{A}}[\![s_0 \texttt{ ; } s_1]\!] &= \overline{\mathcal{A}}[\![s_1]\!]\circ\overline{\mathcal{A}}[\![s_0]\!]\\
\overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] &= \lambda\overline{a}.\,\overline{\mathcal{A}}[\![s_0]\!]\overline{a}\,\dot{\sqcup}\,\overline{\mathcal{A}}[\![s_1]\!]\overline{a}\\
\overline{\mathcal{A}}[\![\texttt{while } e \texttt{ do } s]\!] &= \mathrm{lfp}\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a}\,\dot{\sqcup}\,\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\,\overline{a})\\
\overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!] &= \lambda\overline{a}.\,\prod_{k\in\mathbb{K}_\psi}\begin{cases}\pi_k(\overline{\mathcal{A}}[\![s]\!]\overline{a}) & \text{if } k\models\theta\\ \pi_k(\overline{a}) & \text{if } k\not\models\theta\end{cases}\\
\overline{\mathcal{A}'}[\![n]\!] &= \lambda\overline{a}.\,\prod_{k\in\mathbb{K}_\psi}\texttt{n}\\
\overline{\mathcal{A}'}[\![\texttt{x}]\!] &= \lambda\overline{a}.\,\prod_{k\in\mathbb{K}_\psi}\pi_k(\overline{a})(\texttt{x})\\
\overline{\mathcal{A}'}[\![e_0\oplus e_1]\!] &= \lambda\overline{a}.\,\prod_{k\in\mathbb{K}_\psi}\pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\overline{a})\,\widehat{\oplus}\,\pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\overline{a})
\end{aligned}
$$

Figure 2 Definitions of $\overline{\mathcal{A}}[\![s]\!] : (\mathbb{A}\to\mathbb{A})^{\mathbb{K}_\psi}$ and $\overline{\mathcal{A}'}[\![e]\!] : (\mathbb{A}\to \mathit{Const})^{\mathbb{K}_\psi}$.

$|\mathbb{K}_\psi|$-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 $\dot{\sqcup},\dot{\sqcap},\dot{\bot},\dot{\top}$. E.g., $\dot{\top}=\prod_{k\in\mathbb{K}_\psi}\top_{\mathbb{A}}=(\top_{\mathbb{A}},\ldots,\top_{\mathbb{A}})$.

The lifted analysis $\overline{\mathcal{A}}[\![s]\!]$ should be a function from $\mathbb{A}^{\mathbb{K}_\psi}$ to $\mathbb{A}^{\mathbb{K}_\psi}$. However, using a tuple of $|\mathbb{K}_\psi|$ independent simple functions of type $\mathbb{A}\to\mathbb{A}$ is sufficient. 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. Thus, 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 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 [19] by following the steps of the calculational approach to abstract interpretation [9]: 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 [19] as well.

The (transfer) function $\overline{\mathcal{A}}[\![s]\!]$ captures the effect of analysing the statement s in an input 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 `if` case results in the least upper bound (join) of the effects from the two corresponding branches, and it abstracts away the analysis information at the guard (condition) point. For the `while` statement, we compute the least fixed point

of a functional[1] 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 [19, 10]. 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:

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

We lift the above operation configuration-wise, and in this way obtain a new operation $\dot{\widehat{\oplus}}$ on tuples of *Const* values.

▶ **Example 1.** Consider the $\overline{\text{IMP}}$ program S_1:

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

with the set $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. By using the rules of Fig. 2, we can calculate $\overline{\mathcal{A}}[\![S_1]\!]$ for a store in which `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:

$$\big([\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top]\big) \overset{\overline{\mathcal{A}}[\![\mathtt{x}:=0]\!]}{\longmapsto} \big([\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 0]\big)$$
$$\overset{\overline{\mathcal{A}}[\![\#\text{if } (A)\ \mathtt{x}:=\mathtt{x}+1]\!]}{\longmapsto} \big([\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 0]\big) \overset{\overline{\mathcal{A}}[\![\#\text{if } (B)\ \mathtt{x}:=1]\!]}{\longmapsto} \big([\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1]\big)$$

After evaluating S_1, the variable `x` has the constant value 1 for all valid configurations. Observe that in the above lifted stores many components are the same, i.e. many configurations have equivalent analysis information. Such lifted stores can be more compactly represented using sharing (e.g., bit vectors or formulae), which in effect will result in more efficient implementation of the lifted analysis.

Let S_2 be a program obtained from S_1, such that `#if` (B) `x` $:= 1$ is replaced with `#if` (B) `x` $:= \mathtt{x} - 1$. Then, we have:

$$\overline{\mathcal{A}}[\![S_2]\!]\big([\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top],[\mathtt{x}\mapsto\top]\big) = \big([\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto -1]\big)$$

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

3 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 if using symbolic representations or sharing to avoid direct storage of $|\mathbb{K}_\psi|$-sized tuples as done in [5]. We want to replace $\mathbb{A}^{\mathbb{K}_\psi}$ with a smaller domain obtained by abstraction and perform an approximate, but feasible, lifted analysis.

---

[1] The functional of the `while` rule is: $\lambda\overline{\Phi}.\,\lambda\overline{a}.\,\overline{a} \dot{\sqcup} \overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\,\overline{a})$.

[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 $\theta \in FeatExp$.

3.1 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 constructors, join and projection, along with a sequential and parallel composition of abstractions. The set of abstractions *Abs* is generated by the following grammar:

$$\alpha \ ::= \ \boldsymbol{\alpha}^{\text{join}} \mid \boldsymbol{\alpha}^{\text{proj}}_{\varphi} \mid \alpha \circ \alpha \mid \alpha \otimes \alpha \tag{1}$$

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

Join. Consider the following scenario. An analysis is run interactively, while a developer is typing in a development environment. The analysis finds simple errors and warnings. In this scenario, the analysis must be fast and it should consider all legal configurations $\mathbb{K}$. It is not problematic if some spurious errors are introduced, since, like previously, 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 variants that participate in all products.

The *join* abstraction gathers the information about all valid 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 follows:

$$\boldsymbol{\alpha}^{\text{join}}(\overline{a}) = \big(\textstyle\bigsqcup_{k\in\mathbb{K}} \pi_k(\overline{a})\big) \quad \text{and} \quad \boldsymbol{\gamma}^{\text{join}}(a) = \prod_{k\in\mathbb{K}} a \tag{2}$$

We overload abstraction names (α) to apply not only to domain elements but also to sets of features, sets of configurations, and, later, to program code. The new set of valid configurations is $\boldsymbol{\alpha}^{\text{join}}(\mathbb{K}) = \{\bigvee_{k\in\mathbb{K}} k\}$. Thus, we have only one valid configuration denoted by the 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 2.** $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{join}}(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ *is a Galois connection* [3] [4].

▶ **Example 3.** Let us return to the scenario of using *join* for improving analysis performance. Assume that the feature model is given by $\psi = A \vee B$ with valid configurations $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Now, the final stores we obtain by analyzing programs S_1 and S_2 from Example 1 are $\overline{a}_{S_1} = \big([\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto 1]\big)$ and $\overline{a}_{S_2} = \big([\mathtt{x}\mapsto 0],[\mathtt{x}\mapsto 1],[\mathtt{x}\mapsto -1]\big)$. Applying the join abstraction we obtain $\boldsymbol{\alpha}^{\text{join}}(\overline{a}_{S_1}) = \big([\mathtt{x}\mapsto 1]\big)$ and $\boldsymbol{\alpha}^{\text{join}}(\overline{a}_{S_2}) = \big([\mathtt{x}\mapsto \top]\big)$. In both cases the state representation has been significantly decreased. In the former case, the abstraction promptly notices that `x` is a constant regardless of the configuration. In the latter case, the abstraction looses precision by saying that `x` is not a constant in general, even if it was a constant in each of the configurations considered. We will continue using stores $\overline{a}_{S_1}$ and $\overline{a}_{S_2}$ in the subsequent examples. ◀

---

[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 $l \in L, m \in M$.

[4] The proofs of all theorems in this section can be found in [14, App. A].

Projection. In industrial practice the number of products actually deployed is often only a small subset of $\mathbb{K}$ [3]. In such case, analyzing all legal (valid) configurations seems unnecessary, and performance of analyses can be improved by abstracting many products away. This is achieved by a configuration projection, which removes configurations that do not satisfy a given constraint, for instance a disjunction of product configurations of interest. 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 formula over feature names. 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:

$$\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\overline{a}) = \prod_{k\in\mathbb{K},k\models\varphi} \pi_k(\overline{a}) \tag{3}$$

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

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

▶ **Theorem 4.** $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}\rangle \underset{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}{\overset{\boldsymbol{\gamma}^{\text{proj}}_{\varphi}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq}\rangle$ *is a Galois connection.*

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 5.** 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 store $\overline{a}_{S_2}$ with the set of valid configurations $\mathbb{K}_{\psi}$ from Example 3. The set of deployed products is defined by formula $\varphi = A$ (so all possible programs with feature A are marketed). By definition of projection (3), we have: $\boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_2}) = \big(\pi_{A\wedge B}(\overline{a}_{S_2}), \pi_{A\wedge\neg B}(\overline{a}_{S_2})\big) = \big([\mathtt{x}\mapsto 0], [\mathtt{x}\mapsto 1]\big)$, and $\boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_2}) = (\pi_{\neg A\wedge B}(\overline{a}_{S_2})) = ([\mathtt{x}\mapsto -1])$. The state representation is effectively decreased to two, respectively one, components. ◀

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, which cannot be expressed using a simple strengthening of a global feature model.

Sequential Composition. We use composition to build complex abstractions out of the basic ones, which also allows us to keep the number of operators in the framework and in the implementation low.

Recall that a composition of two Galois connections is also a Galois connection [11]. 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 $\overline{a}' \in \mathbb{A}^{(\alpha_2\circ\alpha_1)(\mathbb{K})}$. Also $(\alpha_2 \circ \alpha_1)(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$.

▶ **Example 6.** 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: $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(\overline{a}_{S_2}) = (\pi_{A\wedge B}(\overline{a}_{S_2}) \sqcup \pi_{A\wedge\neg B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto \top])$ and $\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\neg A}(\overline{a}_{S_2}) = (\pi_{\neg A\wedge B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto -1])$. ◀

Parallel Composition. Consider a product line where two disjoint groups of products share the same code base: one group is correctness critical, the other comprises correctness 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 correctness 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 correctness 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 overall, 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 $\overline{a}_1 \times \overline{a}_2 \in \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})$ as:

$$\overline{a}_1 \times \overline{a}_2 = \prod_{k \in \alpha_1(\mathbb{K}) \cup \alpha_2(\mathbb{K})} \begin{cases} \pi_k(\overline{a}_1) & \text{if } k \in \alpha_1(\mathbb{K}) \setminus \alpha_2(\mathbb{K}) \\ \pi_k(\overline{a}_1) \sqcup \pi_k(\overline{a}_2) & \text{if } k \in \alpha_1(\mathbb{K}) \cap \alpha_2(\mathbb{K}) \\ \pi_k(\overline{a}_2) & \text{if } k \in \alpha_2(\mathbb{K}) \setminus \alpha_1(\mathbb{K}) \end{cases} \tag{6}$$

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

$$(\alpha_1 \otimes \alpha_2)(\overline{a}) = \alpha_1(\overline{a}) \times \alpha_2(\overline{a}) \tag{7}$$

$$(\gamma_1 \otimes \gamma_2)(\overline{a}') = \gamma_1(\pi_{\alpha_1(\mathbb{K})}(\overline{a}')) \sqcap \gamma_2(\pi_{\alpha_2(\mathbb{K})}(\overline{a}')) \ , \text{ where} \tag{8}$$

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

▶ **Theorem 7.** $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\alpha_1 \otimes \alpha_2}{\overset{\gamma_1 \otimes \gamma_2}{\leftrightarrows}} \langle \mathbb{A}^{(\alpha_1 \otimes \alpha_2)(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ *is a Galois connection.*

▶ **Example 8.** Let us assume that for products with feature A we need precise analysis results, and for products without this feature we do not need so precise results. We are interested in analyzing products with A thoroughly, while the analysis of the products without A can be speeded up. To this end we build the following abstraction: $\boldsymbol{\alpha}^{\text{proj}}_{A} \otimes (\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\neg A})$. ◀

3.2 Derived Abstractions

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

Join-Project. Recall the construction of Example 6, where we combined projection with a join in order to confound a subset of legal configurations. This pattern has occurred so often in our exercises that we introduced a syntactic sugar for it. For a formula φ over features, the abstraction $\boldsymbol{\alpha}^{\text{join}}_{\varphi}$ 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

$$\boldsymbol{\alpha}^{\text{join}}_{\varphi} = \boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\varphi} \quad \text{and} \quad \boldsymbol{\gamma}^{\text{join}}_{\varphi} = \boldsymbol{\gamma}^{\text{proj}}_{\varphi} \circ \boldsymbol{\gamma}^{\text{join}} \tag{9}$$

where $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}}{\overset{\boldsymbol{\gamma}^{\text{proj}}_{\varphi}}{\leftrightarrows}} \langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ and $\langle \mathbb{A}^{\boldsymbol{\alpha}^{\text{proj}}_{\varphi}(\mathbb{K})}, \dot{\sqsubseteq} \rangle \underset{\boldsymbol{\alpha}^{\text{join}}}{\overset{\boldsymbol{\gamma}^{\text{join}}}{\leftrightarrows}} \langle \mathbb{A}^{(\boldsymbol{\alpha}^{\text{join}} \circ \boldsymbol{\alpha}^{\text{proj}}_{\varphi})(\mathbb{K})}, \dot{\sqsubseteq} \rangle$ are Galois connections. Now the compositions in Example 6 can be written simply as $\boldsymbol{\alpha}^{\text{join}}_{A}$ and $\boldsymbol{\alpha}^{\text{join}}_{\neg A}$.

Ignoring features. Consider a scenario, where a configurable third-party component is integrated into a product line. The code base is large, and a static analysis does not scale to this size. In a compile-analyze-test cycle errors appear most often in the newly written code, and are thus relatively little influenced by how the features of the third party component are configured. Lowering precision on analyzing external components can allow finding errors faster. This scenario can be realized using a feature projection, which simplifies feature domains by confounding executions differing only on uninteresting features.

Before defining feature projection, let us consider a simpler case of ignoring a single feature $A \in \mathbb{F}$ that is not directly relevant for current analysis. The *ignore feature* abstraction merges any configurations that only differ with regard to A, and are identical with regard to remaining features, $\mathbb{F} \backslash \{A\}$. We write $k \backslash_A$ for a formula obtained by eliminating the feature A from k. The new set of configurations is given by $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}) = \{\bigvee_{k \in \mathbb{K}, k\backslash_A \equiv k'} k \mid k' \in \{k\backslash_A \mid k \in \mathbb{K}\}\}$. The abstraction $\boldsymbol{\alpha}^{\text{fignore}}_{A} : \mathbb{A}^{\mathbb{K}} \to \mathbb{A}^{\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K})}$ and concretization functions $\boldsymbol{\gamma}^{\text{fignore}}_{A} : \mathbb{A}^{\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K})} \to \mathbb{A}^{\mathbb{K}}$ are:

$$\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}) = \prod_{k' \in \boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K})} \bigsqcup_{k \in \mathbb{K}, k \models k'} \pi_k(\overline{a}) \tag{10}$$

$$\boldsymbol{\gamma}^{\text{fignore}}_{A}(\overline{a}') = \prod_{k \in \mathbb{K}} \pi_{k'}(\overline{a}') \;\text{ if } k \models k' \tag{11}$$

It turns out that ignoring features can be derived from the above basic abstractions as shown in the following theorem:

▶ **Theorem 9.** *Let* $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}) = \{k'_1, \ldots, k'_n\}$. *Then:*

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

▶ **Example 10.** We consider the lifted store $\overline{a}_{S_2}$ with $\mathbb{K}_{\psi} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Then, we have $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\mathbb{K}_{\psi}) = \{(A \wedge B) \vee (\neg A \wedge B), A \wedge \neg B\}$ and $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}_{S_2}) = (\pi_{A \wedge B}(\overline{a}_{S_2}) \sqcup \pi_{\neg A \wedge B}(\overline{a}_{S_2}), \pi_{A \wedge \neg B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto \top], [\mathtt{x} \mapsto 1])$. On the other hand, we have $\boldsymbol{\alpha}^{\text{fignore}}_{B}(\mathbb{K}_{\psi}) = \{(A \wedge B) \vee (A \wedge \neg B), \neg A \wedge B\}$ and $\boldsymbol{\alpha}^{\text{fignore}}_{A}(\overline{a}_{S_2}) = (\pi_{A \wedge B}(\overline{a}_{S_2}) \sqcup \pi_{A \wedge \neg B}(\overline{a}_{S_2}), \pi_{\neg A \wedge B}(\overline{a}_{S_2})) = ([\mathtt{x} \mapsto \top], [\mathtt{x} \mapsto -1])$. ◀

Feature Projection. 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 projection operator which simply ignores a set of features $\{A_1, \ldots, A_k\} \subseteq \mathbb{F}$:

$$\boldsymbol{\alpha}^{\text{fproj}}_{\{A_1, \ldots, A_k\}} = \boldsymbol{\alpha}^{\text{fignore}}_{A_1} \circ \cdots \circ \boldsymbol{\alpha}^{\text{fignore}}_{A_k} \quad \text{and} \quad \boldsymbol{\gamma}^{\text{fproj}}_{\{A_1, \ldots, A_k\}} = \boldsymbol{\gamma}^{\text{fignore}}_{A_k} \circ \cdots \circ \boldsymbol{\gamma}^{\text{fignore}}_{A_1}$$

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!] \circ \gamma)(\overline{d}) = \alpha(\overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!](\gamma(\overline{d}))) = && \text{(by def. of } \circ\text{)}\\
&= \alpha\Big(\prod_{k\in\mathbb{K}_\psi} \begin{cases} \pi_k(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d})) & \text{if } k \models \theta \\ \pi_k(\gamma(\overline{d})) & \text{if } k \not\models \theta \end{cases}\Big) && \text{(def. of } \overline{\mathcal{A}} \text{ in Fig. 2)}\\
&\dot{\sqsubseteq} \prod_{k'\in\alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\alpha(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d}))) & \text{if } k' \models \theta \\ \pi_{k'}(\alpha(\gamma(\overline{d}))) \sqcup \pi_{k'}(\alpha(\overline{\mathcal{A}}[\![s]\!]\gamma(\overline{d}))) & \text{if } \mathrm{sat}(k'\wedge\theta) \wedge \mathrm{sat}(k'\wedge\neg\theta) \\ \pi_{k'}(\alpha(\gamma(\overline{d}))) & \text{if } k' \models \neg\theta \end{cases} && \text{(by Lemma 2 in [14, App. C])}\\
&\dot{\sqsubseteq} \prod_{k'\in\alpha(\mathbb{K}_\psi)} \begin{cases} \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } k' \models \theta \\ \pi_{k'}(\overline{d}) \sqcup \pi_{k'}(\overline{\mathcal{D}}_\alpha[\![s]\!]\overline{d}) & \text{if } \mathrm{sat}(k'\wedge\theta) \wedge \mathrm{sat}(k'\wedge\neg\theta) \\ \pi_{k'}(\overline{d}) & \text{if } k' \models \neg\theta \end{cases} && \text{(by IH and } \alpha\circ\gamma \text{ reductive)}\\
&= \overline{\mathcal{D}}_\alpha[\![\texttt{\#if } (\theta)\ s]\!]\,\overline{d}
\end{aligned}
$$

Figure 3 Calculational derivation of $\overline{\mathcal{D}}_\alpha[\![\texttt{\#if } (\theta)\ s]\!]$, the abstraction of $\overline{\mathcal{A}}[\![\texttt{\#if } (\theta)\ s]\!]$. The 'reductive' property of all Galois connections is $(\alpha \circ \gamma)(\overline{d}) \sqsubseteq \overline{d}$ for all $\overline{d}$.

It follows from the theorems of Section 3.1 that all the derived pairs of abstraction and concretization functions are Galois connections.

4 Abstracting Lifted Analyses

We will now demonstrate how to derive abstracted lifted analyses using the operators of Section 3, using the case of constant propagation for IMP programs as an example. Recall that this 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 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 abstractions presented in Section 3. We will also write $(\alpha, \gamma) \in \mathit{Abs}$ to denote a Galois connection obtained in such 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 by the fixed point transfer theorem (FPT for short) [10]: $\alpha(\mathrm{lfp} f) \sqsubseteq \mathrm{lfp} F \sqsubseteq \mathrm{lfp} F^\#$. 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 [9] 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,

$$
\begin{aligned}
&(\alpha \circ \overline{\mathcal{A}'}[\![e_0 \oplus e_1]\!] \circ \gamma)(\overline{d}) \\
&= \alpha\big(\prod_{k \in \mathbb{K}_\psi} \pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d})) \mathbin{\widehat{\oplus}} \pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))\big) && \text{(by def. of } \circ \text{, and } \overline{\mathcal{A}'} \text{ in Fig. 2)} \\
&= \alpha\big(\prod_{k \in \mathbb{K}_\psi} \pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d}) \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))\big) && \text{(by def. of } \pi_k \text{ and } \dot{\widehat{\oplus}}) \\
&= \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\alpha\big(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d}) \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d})\big)) && \text{(by def. of } \alpha) \\
&\dot{\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\alpha(\overline{\mathcal{A}'}[\![e_0]\!]\gamma(\overline{d})) \mathbin{\dot{\widehat{\oplus}}} \alpha(\overline{\mathcal{A}'}[\![e_1]\!]\gamma(\overline{d}))) && \text{(by Lemma 3 in [14, App. C])} \\
&\dot{\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d} \mathbin{\dot{\widehat{\oplus}}} \overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d}) && \text{(by IH, twice)} \\
&\dot{\sqsubseteq} \prod_{k' \in \alpha(\mathbb{K}_\psi)} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_0]\!]\overline{d}) \mathbin{\widehat{\oplus}} \pi_{k'}(\overline{\mathcal{D}'}_\alpha[\![e_1]\!]\overline{d}) && \text{(by def. of } \pi_{k'} \text{ and } \dot{\widehat{\oplus}}) \\
&= \overline{\mathcal{D}'}_\alpha[\![e_0 \oplus e_1]\!]\overline{d}
\end{aligned}
$$

Figure 4 Calculational derivation of $\overline{\mathcal{D}}_\alpha[\![e_0 \oplus e_1]\!]$.

deceivingly akin to equation reasoning.

Let us 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: $\alpha \circ \overline{\mathcal{A}}[\![s']\!] \circ \gamma \mathbin{\dot{\sqsubseteq}} \overline{\mathcal{D}}_\alpha[\![s']\!]$. The derivation in Fig. 3 begins with composing the concretization and abstraction functions with the concrete transfer function and then proceeds by expanding definitions. An (inner) induction on the structure of the abstraction α follows, delegated to the Appendix for brevity. 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. We give derivational steps for $e_0 \oplus e_1$ in Fig. 4.

The derivations for other cases are similar and can be found in [14, App. B]. The process results in the definitions of $\overline{\mathcal{D}}_\alpha[\![s]\!]$ and $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ presented in Fig. 5. Soundness of the abstracted analysis follows by construction; more precisely the complete calculation constitutes an inductive proof of the following theorem:

▶ **Theorem 11** (Soundness of Abstracted Analysis). *We have that:*

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

Monotonicity of the abstracted analysis is shown in [14, App. D].

▶ **Theorem 12** (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.*

▶ **Example 13.** Consider the program S_1 from Example 1, with $\mathbb{K}_\psi = \{A\wedge B, A\wedge\neg B, \neg A\wedge B\}$. We calculate $\overline{\mathcal{D}}_{\alpha_1}[\![S_1]\!]$ for $\alpha_1 = \boldsymbol{\alpha}^{\text{join}}_A$. Following the rules of Fig. 5, we obtain the following confounded abstract execution off all configurations containing the feature A:

$$\big([\mathtt{x} \mapsto \top]\big) \overset{\overline{\mathcal{D}}_{\alpha_1}[\![\mathtt{x}:=0]\!]}{\longmapsto} \big([\mathtt{x} \mapsto 0]\big) \overset{\overline{\mathcal{D}}_{\alpha_1}[\![\#\mathsf{if}\ (A)\ \mathtt{x}:=\mathtt{x}+1]\!]}{\longmapsto} \big([\mathtt{x} \mapsto 1]\big) \overset{\overline{\mathcal{D}}_{\alpha_1}[\![\#\mathsf{if}\ (B)\ \mathtt{x}:=1]\!]}{\longmapsto} \big([\mathtt{x} \mapsto 1]\big)$$

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

Figure 5 Definitions of $\overline{\mathcal{D}}_\alpha[\![\overline{s}]\!] : (\mathbb{A}\to\mathbb{A})^{\alpha(\mathbb{K}_\psi)}$ and $\overline{\mathcal{D}'}_\alpha[\![\overline{e}]\!] : (\mathbb{A}\to \mathit{Const})^{\alpha(\mathbb{K}_\psi)}$.

In the last step we used $\overline{\mathcal{D}}_{\alpha_1}[\![\texttt{\#if}(B)\ \texttt{x} := 1]\!]([\texttt{x}\mapsto 1]) = ([\texttt{x}\mapsto 1])\,\dot\sqcup\,\overline{\mathcal{D}}_{\alpha_1}[\![\texttt{x} := 1]\!]([\texttt{x}\mapsto 1])$ since $((A\wedge B)\vee(A\wedge\neg B))\wedge B$ and $((A\wedge B)\vee(A\wedge\neg B))\wedge\neg B$ are both satisfiable. The final result shows that the value of x is the constant 1 for every configuration that satisfies A. On the other hand, for the program S_2 and the same abstraction we obtain $\overline{\mathcal{D}}_{\alpha_1}[\![S_2]\!]([\texttt{x}\mapsto\top]) = ([\texttt{x}\mapsto\top])$, so the value of x is lost (approximated) by $\overline{\mathcal{D}}_{\alpha_1}$. ◀

We may implement the abstracted analysis in Fig. 5 directly by using Kleene's fixed point theorem to calculate fixed points of loops iteratively. But, we can also extract corresponding data-flow equations, and then apply the known iterative algorithms to calculate fixed-point solutions. We assume that the individual statements are uniquely labelled with labels ℓ. Given an abstraction α, for each statement s^ℓ we generate two abstracted stores $[\![s^\ell]\!]^\alpha_{\mathsf{in}}, [\![s^\ell]\!]^\alpha_{\mathsf{out}} : \mathbb{A}^{\alpha(\mathbb{K}_\psi)}$, which describe the input and output abstract store for all configurations before and after executing the statement s^ℓ. They are related with the definitions for abstracted analysis $\overline{\mathcal{D}}_\alpha$ given in Fig. 5 as follows: for each statement s the input store $[\![s^\ell]\!]^\alpha_{\mathsf{in}}$ is substituted for the parameter $\overline{d}$, and the output store $[\![s^\ell]\!]^\alpha_{\mathsf{out}}$ for the value of the corresponding function. The complete list of data-flow equations are given in Fig 6. We can derive data-flow equations for expressions as well, but for brevity we refer directly to $\overline{\mathcal{D}'}_\alpha[\![e]\!]$ function. The obtained data-flow equations are provably sound as shown in [14, App. E].

▶ **Theorem 14** (Soundness of Abstracted Data-Flow Equations). *For all $s\in \mathit{Stm}$ and $\alpha\in \mathit{Abs}$, such that $[\![s^\ell]\!]^\alpha_{\mathsf{in}}$ and $[\![s^\ell]\!]^\alpha_{\mathsf{out}}$ satisfy the data-flow equations in Fig. 6, it holds:*

$$\overline{\mathcal{D}}_\alpha[\![s^\ell]\!]([\![s^\ell]\!]^\alpha_{\mathsf{in}})\ \dot\sqsubseteq\ [\![s^\ell]\!]^\alpha_{\mathsf{out}}$$

5 Variability Abstraction with Syntactic Transformation

The analyses $\overline{\mathcal{A}}$ and $\overline{\mathcal{D}}_\alpha$ can be implemented either directly by using definitions of Figs. 2 and 5, or by extracting the corresponding data-flow equations. An entirely different way

$$[\![\texttt{skip}^{\ell}]\!]^{\alpha}_{\mathsf{out}} = [\![\texttt{skip}^{\ell}]\!]^{\alpha}_{\mathsf{in}}$$

$$\underline{\forall k' \in \alpha(\mathbb{K}_{\psi})}\colon \pi_{k'}([\![\texttt{x} :=^{\ell} e^{\ell_0}]\!]^{\alpha}_{\mathsf{out}}) = \pi_{k'}([\![\texttt{x} :=^{\ell} e^{\ell_0}]\!]^{\alpha}_{\mathsf{in}})^{\alpha}[\texttt{x} \mapsto \pi_{k'}(\overline{\mathcal{D}'}_{\alpha}[\![e^{\ell_0}]\!][\![\texttt{x} :=^{\ell} e^{\ell_0}]\!]^{\alpha}_{\mathsf{in}})]$$

$$\begin{aligned}
[\![s_0^{\ell_0} ;^{\ell} s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{out}} &= [\![s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{out}} \\
[\![s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{in}} &= [\![s_0^{\ell_0}]\!]^{\alpha}_{\mathsf{out}} \\
[\![s_0^{\ell_0}]\!]^{\alpha}_{\mathsf{in}} &= [\![s_0^{\ell_0} ;^{\ell} s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{in}}
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{out}} &= [\![s_0^{\ell_0}]\!]^{\alpha}_{\mathsf{out}} \dot{\sqcup} [\![s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{out}} \\
[\![s_0^{\ell_0}]\!]^{\alpha}_{\mathsf{in}} &= [\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{in}} \\
[\![s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{in}} &= [\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]^{\alpha}_{\mathsf{in}}
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]^{\alpha}_{\mathsf{out}} &= [\![s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}} \\
[\![s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}} &= [\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}} \dot{\sqcup} [\![s^{\ell_0}]\!]^{\alpha}_{\mathsf{out}}
\end{aligned}$$

$$\underline{\forall k' \in \alpha(\mathbb{K}_{\psi})}\colon \pi_{k'}([\![\texttt{\#if}^{\ell}\ (\theta)\ s^{\ell_0}]\!]^{\alpha}_{\mathsf{out}}) = \begin{cases} \pi_{k'}([\![s^{\ell_0}]\!]^{\alpha}_{\mathsf{out}}) & \text{if } k' \models \theta \\ \pi_{k'}([\![\texttt{\#if}^{\ell}\ (\theta)\ s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}}) \sqcup \pi_{k'}([\![s^{\ell_0}]\!]^{\alpha}_{\mathsf{out}}) & \text{if } \mathrm{sat}(k' \wedge \theta) \wedge \mathrm{sat}(k' \wedge \neg\theta) \\ \pi_{k'}([\![\texttt{\#if}^{\ell}\ (\theta)\ s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}}) & \text{if } k' \models \neg\theta \end{cases}$$

$$\underline{\forall k' \in \alpha(\mathbb{K}_{\psi})}\colon \pi_{k'}([\![s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}}) = \pi_{k'}([\![\texttt{\#if}^{\ell}\ (\theta)\ s^{\ell_0}]\!]^{\alpha}_{\mathsf{in}}) \quad \text{if } \mathrm{sat}(k' \wedge \theta)$$

Figure 6 Data-flow equations for abstracted constant propagation.

to implement $\overline{\mathcal{D}}_{\alpha}$ is to execute the abstraction on the source program, before running the analysis, and then running the previously existing analysis $\overline{\mathcal{A}}$ on this transformed program. We take this route as it allows to completely reuse the effort invested in designing and implementing $\overline{\mathcal{A}}$.

Any $\overline{\mathrm{IMP}}$ program s with sets of features $\mathbb{F}$ and valid configurations $\mathbb{K}$ is translated into a corresponding abstract program $\alpha(s)$ with corresponding set of features $\alpha(\mathbb{F})$ and set of valid configurations $\alpha(\mathbb{K})$. We define the translation recursively over the structure of α. The function α copies all basic statements of $\overline{IMP}$, 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 $\boldsymbol{\alpha}^{\mathrm{join}}$ (consequently written $\boldsymbol{\alpha}^{\mathrm{join}}_{Z,}$). 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 $\boldsymbol{\alpha}^{\mathrm{join}}$. The new feature name is used to simplify conditions in the transformed code. The $\boldsymbol{\alpha}^{\mathrm{join}}_{Z,}$ rewrite is defined as follows:

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

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

In effect of applying the $\boldsymbol{\alpha}^{\mathrm{join}}_{Z,}$ transformation to any program s we obtain a single variant program, i.e. a SPL with only one valid product where the feature Z is enabled. It can be analyzed with existing single-program analyses. Note that it enables performing family-based analyses with implementations of single-program analyses, albeit with loss of precision. The newly introduced statement $\texttt{lub}(s_0, s_1)$ represents the least upper bound (join) of the results obtained

by executing s_0 and s_1. This is the only language-dependent aspect of `reconfigurator`. It can have different implementations depending on the programming language and the analysis we work with. In our case, we exploit the fact that $\overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]$ ignores the branching condition (cf. Fig. 2) and use $\texttt{lub}(s_0, s_1) = \texttt{if } (n) \texttt{ then } s_0 \texttt{ else } s_1$ for some fixed integer n. Finally, observe that $\texttt{\#if } (\neg Z)\ \boldsymbol{\alpha}^{\text{join}}_{Z,\top}(s)$ is equivalent to `skip`, however it is useful to keep the statement in the program, which makes it easy to merge programs when we use compound abstractions (below).

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 $\overline{\alpha}$ are functions transforming the condition θ and the statement s. It is easy to extract $\overline{\alpha}(\theta)$ and $\overline{\alpha}(s, \theta)$ from the above rewrites for $\boldsymbol{\alpha}^{\text{join}}_{Z,\top}$ and $\boldsymbol{\alpha}^{\text{proj}}_{\varphi}$. We will use them in defining transformations for binary operators.

Now, for the case of parallel composition $\alpha_1 \otimes \alpha_2$, recall that the set $\alpha_1 \otimes \alpha_2(\mathbb{K})$ is the union of $\alpha_1(\mathbb{K})$ and $\alpha_2(\mathbb{K})$. However 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 aim, 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 given by:

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

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

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

Observe that the second case of the parallel composition transformation can only appear if the second case of a join transformation has been used somewhere in recursive rewriting of s (perhaps deep). All the other rewrites leave s intact. However, in such case 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 of abstractions $\alpha_2 \circ \alpha_1$ we use the following rewrites: $\alpha_2 \circ \alpha_1(\mathbb{F}) = \alpha_2(\alpha_1(\mathbb{F}))$, $\alpha_2 \circ \alpha_1(\mathbb{K}) = \alpha_2(\alpha_1(\mathbb{K}))$, and for the `#if` statement we have $\alpha_2 \circ \alpha_1(\texttt{\#if } (\theta)\ s) = \texttt{\#if } \left(\overline{\alpha_2}(\overline{\alpha_1}(\theta))\right)\ \overline{\alpha_2}(\overline{\alpha_1}(s, \theta), \overline{\alpha_1}(\theta))$.

▶ **Example 15.** Consider the program S_1': $\texttt{\#if}(A)\ \texttt{x} := \texttt{x} + 1; \texttt{\#if } (B)\ \texttt{x} := 1$ with $\mathbb{F} = \{A, B\}$, $\psi = A \vee B$, and $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. Then

$$\boldsymbol{\alpha}^{\text{join}}_{Z,\top} \circ \boldsymbol{\alpha}^{\text{proj}}_{A}(S_1') = \texttt{\#if } (Z)\ \texttt{x} := \texttt{x} + 1; \texttt{\#if } (Z)\ \texttt{lub}(\texttt{x} := 1, \texttt{skip}) \tag{12}$$

The set of valid configurations after projection is changed to $\{A \wedge B, A \wedge \neg B\}$, and after join again to just $\{Z\}$. The obtained program has only one configuration, the one that satisfies Z. The projection does not change the statements of the program. The join rewrite however, simplifies the first `#if` (it is statically determined; cf. the first case of $\boldsymbol{\alpha}^{\text{join}}_{Z,\top}$ 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

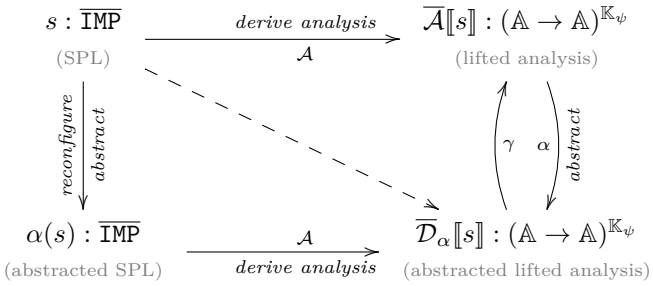


Figure 7 Illustration of *derive* vs *abstract*: $\overline{\mathcal{D}}_\alpha[\![s]\!] = \overline{\mathcal{A}}[\![\alpha(s)]\!]$.

abstracted program. Note that since Z is the only one valid configuration, the obtained program is equivalent to: $\texttt{x} := \texttt{x} + 1; \texttt{lub}(\texttt{x} := 1, \texttt{skip})$. Similarly, we can calculate: $\boldsymbol{\alpha}^{\text{join}}_{Z,'}(\boldsymbol{\alpha}^{\text{proj}}_B(S'_1)) = \texttt{\#if } (Z)\ \texttt{lub}(\texttt{x} := \texttt{x} + 1, \texttt{skip}); \texttt{\#if } (Z)\ \texttt{x} := 1$.

Now consider $((\boldsymbol{\alpha}^{\text{join}}_{Z,'} \circ \boldsymbol{\alpha}^{\text{proj}}_A) \otimes \boldsymbol{\alpha}^{\text{proj}}_B)(S'_1)$. 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 (12), and the right rewrite (projection) never changes the statements, so its result is identical to S'_1. Thus we are composing programs (12) and S'_1 using the parallel composition rewrites. Then $((\boldsymbol{\alpha}^{\text{proj}}_Z \circ \boldsymbol{\alpha}^{\text{join}}_A) \otimes \boldsymbol{\alpha}^{\text{proj}}_B)(S'_1)$ is:

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

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 any legal configuration in $\mathbb{K}'$ at most one of them does not reduce to `skip`. ◀

Now the analysis $\overline{\mathcal{A}}[\![\alpha(s)]\!]$ and $\overline{\mathcal{D}}_\alpha[\![s]\!]$ coincide up to renaming of valid configurations. So the `reconfigurator` together with an existing implementation of $\overline{\mathcal{A}}$ gives us the abstracted analysis $\overline{\mathcal{D}}_\alpha$. The above equality is illustrated by Fig. 1.

▶ **Theorem 16.**
$\forall s \in Stm, \alpha : \mathbb{A}^{\mathbb{K}_\psi} \to \mathbb{A}^{\alpha(\mathbb{K}_\psi)} \in Abs, \overline{d} \in \mathbb{A}^{\alpha(\mathbb{K}_\psi)} :\ \overline{\mathcal{D}}_\alpha[\![s]\!]\,\overline{d} = \overline{\mathcal{A}}[\![\alpha(s)]\!]\,\overline{d}$ [5].

▶ **Example 17.** Consider the program S_1 from Example 1 with $\mathbb{K}_\psi = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$. We have calculated in Example 13 that $\overline{\mathcal{D}}_{\boldsymbol{\alpha}^{\text{join}}_A}[\![S_1]\!]([\texttt{x} \mapsto \top]) = ([\texttt{x} \mapsto 1])$. We now calculate $\overline{\mathcal{A}}[\![\boldsymbol{\alpha}^{\text{join}}_{A,'Z}(S_1)]\!]([\texttt{x} \mapsto \top])$ (here $\boldsymbol{\alpha}^{\text{join}}_{A,'Z} = \boldsymbol{\alpha}^{\text{join}}_{Z,'} \circ \boldsymbol{\alpha}^{\text{proj}}_A$):

$$([\texttt{x} \mapsto \top]) \xrightarrow{\overline{\mathcal{A}}[\![\texttt{x:=0}]\!]} ([\texttt{x} \mapsto 0]) \xrightarrow{\overline{\mathcal{A}}[\![\texttt{\#if } (Z)\ \texttt{x:=x+1}]\!]} ([\texttt{x} \mapsto 1]) \xrightarrow{\overline{\mathcal{A}}[\![\texttt{\#if } (Z)\ \texttt{lub(x:=1,skip)}]\!]} ([\texttt{x} \mapsto 1]) \quad \blacktriangleleft$$

6 Evaluation

Recall that there are two ways to speed up lifted analyses: improving *representation* and increasing *abstraction*. First, we will compare the performance of the two using an unoptimized

[5] The proof of this theorem is in [14, App. F].

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

Figure 8 Characteristics of our three SPL benchmarks (average #configurations in all methods in SPL, total #features, and LOC) along with, for each SPL, its method with maximum variability (#configurations, local #features, and LOC).

lifted analysis as a baseline. Then, we demonstrate that abstraction may be used to turn previously infeasible analysis into feasible ones. Finally, we consider example scenarios that use projection and join and show that abstraction may be applied to an entire product line or when just analyzing a single method.

For our experiments, we use an existing implementation of lifted data-flow analyses for Java Object-Oriented SPLs [5]. The implementation is based on SOOT's intra-procedural data-flow analysis framework [24] for analyzing Java programs. It uses CIDE (Colored IDE) [16] to annotate statements using background colors rather than `#ifdef` directives. Every feature is thus associated with a unique color.

We will consider an unoptimized lifted intra-procedural analysis, known as $\mathcal{A}2$ (from [5]), that uses $|\mathbb{K}_\psi|$-tuples of analysis information, one analysis value per configuration. Also, we consider $\mathcal{A}3$ (from [5]) which is the same analysis as $\mathcal{A}2$, but with improved representation via sharing of analysis-equivalent configurations using a high-performance bit vector library [22]. 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. Note that $\mathcal{A}2$ corresponds to $\overline{\mathcal{A}}$ in Fig. 2 and we will thus refer to it as $\overline{\mathcal{A}}$, while we will use $\overline{\mathcal{S}}$ for the analysis with sharing ($\mathcal{A}3$ in [5]). 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. 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), $\overline{\mathcal{D}}_{N/2}$ (medium abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction). It thus quantifies the potential of abstractions.

For our experiment, we have chosen two analyses: *reaching definitions* and *uninitialized variables*, for which we derived the corresponding definitions of abstracted lifted analysis. We use three SPL benchmarks [16]: 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. Fig. 8 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.

Performance. Fig. 9 shows the time it takes to run each of our three maximum variability methods, as a relative comparion between $\overline{\mathcal{A}}$ (baseline) and $\overline{\mathcal{S}}$ (sharing) *vs* $\overline{\mathcal{D}}_{N/2}$ (medium

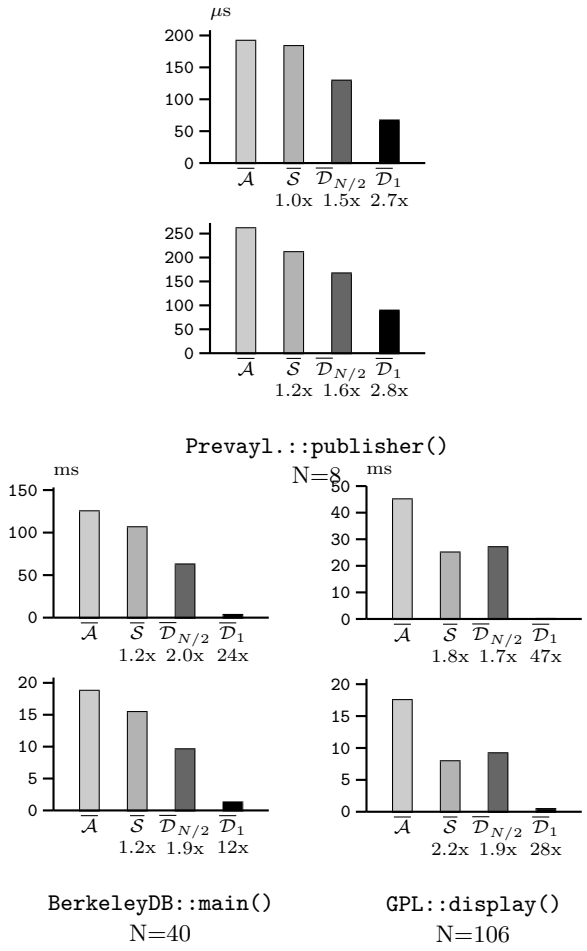


Figure 9 Analysis time for *reaching definitions* (above) and *uninitialized variables* (below): $\overline{\mathcal{A}}$ (baseline) and $\overline{\mathcal{S}}$ (sharing) vs. $\overline{\mathcal{D}}_{N/2}$ (medium abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction).

abstraction) and $\overline{\mathcal{D}}_1$ (maximum abstraction). The experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU with 8 GB memory. All times are reported as averages over ten runs with the highest and lowest number removed. For each benchmark method, we give the speed up factor relative to the baseline (normalized with factor 1) and recall the number of configurations, N.

Our experiment confirms previous results that sharing is indeed effective and especially so for larger values of N [5]. 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 twice as fast (for N=106). We also observe that abstraction is not surprisingly significantly faster than unabstracted analyses (i.e., $\overline{\mathcal{D}}$ *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 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.

From Infeasible to Feasible Analysis. Of course, for very large values of N, analyses may become impractically slow or infeasible. As an experiment, we took a large method

```
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) { ..  } ..
}
```

Figure 10 Code fragment extracted from `BerkeleyDB::main()` with N=40.

(`processFile()` from `BerkeleyDB`) and kept adding unconstrained variability manually. 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 can not only speed up analyses, but also turn previously infeasible analyses feasible.

Projection on 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.

Join on One Method. Figure 10 shows a fragment extracted from `BerkeleyDB`'s `main()` method with N=40 valid configurations. A local variable, `doAction` 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.

7 Related Work

Static analyses can be accelerated by devicing more efficient representations or by introducing abstraction. In family-based analysis for software product lines the representation improvements primarily rely on sharing state information for variants with analysis-equivalent information (which implies reducing redundant computation). This can optimize the analyses considerably [5, 6, 18]. However, in the worst case, the number of variants that a lifted analysis has to consider is still inherently exponential in the number of features, $|\mathbb{F}|$. Thus with a large number of features lifted analyses may become impractical or even infeasible. In this work we have taken the alternate route of using abstraction. Our experiments show that abstraction introduces speed-ups independently of representation gains. Thus our results can be beneficially combined with efficient representations.

An efficient implementation of lifted analysis formulated within the IFDS framework [21] for inter-procedural distributive environments was proposed in SPL$^{\text{LIFT}}$ [4]. 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$^{\text{LIFT}}$ is limited only to distributive data-flow analysis encoded within the IFDS framework. Many analyses, including constant propagation, are not distributive and hence cannot be expressed in IFDS. Let alone static analyses that are not expressible as data-flow analyses (including type checking, model-checking, etc).

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 [19]. The method is based on the calculational approach to abstract interpretation of Cousot [9], applied and contextualized to product lines. In that work [19], 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 [19]), and focus on abstracting variability using them. We devise an expressive calculus for specifying abstraction operators. Also, all abstractions specifiable in our calculus are now automatically executable. Implementing abstractions as program transformations looks similar to the framework defined in [13] for designing source-to-source program transformations by abstract interpretation of program semantics.

A good collection of analyses that have been lifted manually is presented in the survey [23]. We should remark, that the join operation $\boldsymbol{\alpha}^{\text{join}}$ allows applying single program analyses to program families, even if with precision loss. In that sense, the our approach is the first ever method that can *automatically* lift single program analyses to work on program families. Besides the family-based strategy, the survey [23] identifies a *sampling strategy* as a suitable way of analyzing product lines (see also [1]). In the sampling strategy only a random subset of products is analyzed. We remark that once the sample is selected, our projection operator $\boldsymbol{\alpha}^{\text{fproj}}_{\varphi}$ can be used to realize the sampling strategy in a simultanous way by exploiting an existing family-based analysis.

In fact, the abstraction specification framework of Section 3 allows specifying any analysis in the spectrum between a fully family-based analyses, and a single variant, *single product-based*, analysis. 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 [23]. 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.

8 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 11 and Theorem 16) 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.

The `reconfigurator` transformation presently requires that the programming language is able to express *sequential composition* (e.g., "`;`" in `IMP`) and *join of statements* (i.e., `lub` as in "$\sqcup$") with respect to the analysis in question. It would be interesting to consider lifting those assumptions in future, and apply this method to more modeling and programming languages.

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 Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE'13*, pages 482–491, 2013.

2 Don Batory. Feature models, grammars, and propositional formulas. In *9th Int'l Software Product Lines Conf., SPLC'05*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

3 Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. Three cases of feature-based variability modeling in industry. In *17th Int'l Conf. Model-Driven Engineering Languages and Systems, MODELS'14*, pages 302–319, 2014.

4 Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: Statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI'13*, pages 355–364, 2013.

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

6 Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *33th International Conference on Software Engineering, ICSE'11*, pages 321–330, 2011.

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

8 Patrick Cousot. Types as abstract interpretations. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97*, pages 316–331, 1997.

9 Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

10 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *6th Annual ACM Symposium on Principles of Programming Languages, POPL'79*, pages 269–282, 1979.

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

12 Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6(1):69–95, 1999.

13 Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'02*, pages 178–190, 2002.

14 Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses (extended version). *CoRR*, abs/1503.04608, 2015.

15 Eric Goubault, Dominique Guilbaud, Anne Pacalet, Basile Starynkevitch, and Franck Védrine. A simple abstract interpreter for threat detection and test case generation. In *WAPATV'01, with ICSE'01*, Toronto, 2001.

16 Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.

17 Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *30th International Conference on Software Engineering, ICSE'08*, pages 311–320, Leipzig, Germany, 2008. ACM.

18 Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.

19 Jan Midtgaard, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of static analyses for software product lines. In *13th International Conference on Modularity, MODULARITY'14, 2014*, pages 181–192, 2014.

20 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Secaucus, USA, 1999.

21 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95*, POPL'95, pages 49–61, 1995.

22 The colt project: Open source libraries for high performance scientific and technical computing in java. CERN: European Organization for Nuclear Research.

23 Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.

24 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, page 13, 1999.

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