Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Systematic derivation of correct variability-aware program analyses ☆

Jan Midtgaard [a,1], Aleksandar S. Dimovski [b], Claus Brabrand [b,*], Andrzej Wąsowski [b]

[a] DTU Compute, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
[b] IT University of Copenhagen, 2300 Copenhagen S, Denmark

ARTICLE INFO

Article history:
Received 27 February 2014
Received in revised form 13 April 2015
Accepted 14 April 2015
Available online xxxx

Keywords:
Software Product Lines
Software variability
Verification
Static analysis
Abstract interpretation

ABSTRACT

A recent line of work *lifts* particular verification and analysis methods to Software Product Lines (SPL). In an effort to generalize such case-by-case approaches, we develop a systematic methodology for lifting single-program analyses to SPLs using abstract interpretation. Abstract interpretation is a classical framework for deriving static analyses in a compositional, step-by-step manner. We show how to take an analysis expressed as an abstract interpretation and lift each of the abstract interpretation steps to a family of programs (SPL). This includes schemes for lifting domain types, and combinators for lifting analyses and Galois connections. We prove that for analyses developed using our method, the soundness of lifting follows by construction. The resulting *variational abstract interpretation* is a conceptual framework for understanding, deriving, and validating static analyses for SPLs. Then we show how to derive the corresponding variational dataflow equations for an example static analysis, a constant propagation analysis. We also describe how to approximate variability by applying variability-aware abstractions to SPL analysis. Finally, we discuss how to efficiently implement our method and present some evaluation results.

1. Introduction

The methodology of *Software Product Lines* (SPLs) [1] enables systematic development of a *family* of related programs, known as *variants*, from a common code base by maximizing reuse in order to decrease development cost and time-to-market. Each variant in an SPL is specified in terms of *features* selected for that particular variant. The SPL method has grown in popularity over the last 20 years, especially in the domain of embedded systems, including safety critical systems with stringent quality requirements on produced code.

While program families can be implemented using domain-specific languages and general-purpose model transformation [2], often it is possible to use simpler methods that are more easily amenable to testing and analysis. The most popular [3] implementation method in practice relies on a simple form of two-staged computation in preprocessor style: the programming language used (often C) is enriched with the ability to express simple compile-time computations (often

☆ Supported by *The Danish Council for Independent Research* (grant no. 0602-02327B) under the Sapere Aude scheme, project VARIETE.
* Corresponding author.
E-mail address: brabrand@itu.dk (C. Brabrand).
[1] A major part of this work was carried out while the author was at Dept. of Computer Science, Aarhus University, 8200 Aarhus N, Denmark.

http://dx.doi.org/10.1016/j.scico.2015.04.005
0167-6423/

C preprocessor), e.g., it can be enriched with '`#if` *A*' statements in which *A* represents a feature. At build-time, the source code is first configured, a variant describing a particular product is derived by selecting a set of features relevant for it, and only then is this variant compiled or interpreted.

In this two-stage process the compiler handles only the second stage artifacts—the code of the actual product variant. Consequently, all its static analysis mechanisms (such as type checking, data and control-flow analyses) do not analyze the entire program source code, but only the variant specialized for a particular product. This is entirely unacceptable for analyses that aim at identifying program errors. Often, it is not feasible for the vendor shipping the code to analyze each of the variants separately, due to a combinatorial explosion of the number of products (variants). For example, if variability is used to provide personalization of software for various users, it suffices to have 33 independent features to yield more configurations than people on the planet (2^{33}). As little as 320 optional features yield more configurations than the number of atoms in the universe. Now, we have the Linux kernel code base with more than 11,000 features [4]. The problem is particularly burning when run-time errors remain disguised because exhaustive analysis is not possible [5].

In the last decade, many existing program analysis and verification techniques have been *lifted* to work on program families leading to the emergence of so-called *family-based* or *variability-aware analyses* [6]. The main advantage of these analyses is that they do *not* work in two stages, i.e. they do not generate and analyze individual variants separately, but directly analyze the entire code base—all configuration variants at once—at a cost much lower than the accumulated cost of analyzing each of the product variants separately.

Unfortunately, along with the growth of the collection of available lifted analysis methods, a more fundamental worry became increasingly clear: does the variability challenge require redevelopment of the entire language and compiler engineering theory? In response, the industry initiated standardization efforts to codify common understanding of what variability in languages is (for example [7]). In research, a number of papers have started to appear that tackle the more fundamental question of "what is variability in a programming language?" [8]. As part of this larger effort, we attack the problem by developing a systematic understanding of (1) how a single-program analysis relates to the lifted family-based analysis, (2) how programming language definitions (including semantics) are enriched with variability and (3) how a program analysis developed formally for a single program can be systematically lifted into a correct analysis for a family of programs.

We develop a systematic methodology for lifting single-program analyses using abstract interpretation [9]. Abstract interpretation is a unifying theory of sound abstraction and approximation of structures; a well established general framework, which can express many analyses (including data-flow analyses [9], control-flow analyses [10], model checking [11,12], and type checking [13]). Our method exploits knowledge about a single-program analysis to obtain a family-based analysis. The family-based analyses derived using this method are not only sound, but also formally and intimately related to their single program origins. The method is applicable to any analysis expressible as an abstract interpretation, but our focus here is on the constant propagation analysis. The following contributions are made:

(C1) A systematic method for compositional derivation of family-based analyses based on abstract interpretation.
(C2) The correctness (soundness) of the obtained family-based analyses follows by construction.
(C3) Understanding of the structure of the space of family-based analyses (how single-program analyses induce family-based analyses, and which of their abstraction components can be reused at the family level).
(C4) Understanding of individual family-based analyses (in particular, precisely where analysis precision is lost).
(C5) Transfer of the usual benefits of abstract interpretation to family-based analyses (for example, techniques for trading precision for speed and methods for proving analyses to be semantically sound).
(C6) A step-by-step example-driven demonstration of how to derive a family-based analysis.

This work represents an extended and revised version of [14]. Compared to the earlier work, we provide formal and carefully explained proofs of all theorems. We use a running example throughout the paper in order to clarify and improve the presentation of the proposed method and the introduced concepts. In addition, we discuss on an efficient implementation of this method and support our claims by some practical results.

The work is organized as follows. First, a simple imperative language and its operational semantics are presented in Section 2. Then in Section 3, we present a systematic derivation of constant propagation analysis for this language, which is based on the calculational approach to abstract interpretation [15]. In Section 4, we show how the entire derivation process and result can be lifted to the family level for analyzing Software Product Lines. An alternative way to derive lifted analyses of family programs is described in Section 5. We then discuss how the proposed lifted analyses can be efficiently implemented in Section 6. In the end, we discuss related work, and conclude by presenting some ideas for future work.

2. A programming language

We begin by defining the programming language that we want to analyze. Then, we present its operational semantics as we aim to develop a provably sound analysis. Finally, we introduce static variability into the language, and into its formal semantics.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma}\ \textsc{Skip} \qquad \frac{\mathcal{E}(e,\sigma) = v}{\langle \texttt{x} := e, \sigma\rangle \to \sigma[\texttt{x} \mapsto v]}\ \textsc{Assign}$$

$$\frac{\langle s_0, \sigma\rangle \to \langle s_0', \sigma'\rangle}{\langle s_0\ ;\ s_1, \sigma\rangle \to \langle s_0'\ ;\ s_1, \sigma'\rangle}\ \textsc{Seq1} \qquad \frac{\langle s_0, \sigma\rangle \to \sigma'}{\langle s_0\ ;\ s_1, \sigma\rangle \to \langle s_1, \sigma'\rangle}\ \textsc{Seq2}$$

$$\frac{\mathcal{E}(e,\sigma) = v \qquad v \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{else}\ s_1, \sigma\rangle \to \langle s_0, \sigma\rangle}\ \textsc{If1}$$

$$\frac{\mathcal{E}(e,\sigma) = v \qquad v = 0}{\langle \texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{else}\ s_1, \sigma\rangle \to \langle s_1, \sigma\rangle}\ \textsc{If2}$$

$$\frac{\mathcal{E}(e,\sigma) = v \qquad v \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ s, \sigma\rangle \to \langle s\ ;\ \texttt{while}\ e\ \texttt{do}\ s, \sigma\rangle}\ \textsc{Wh1} \qquad \frac{\mathcal{E}(e,\sigma) = v \qquad v = 0}{\langle \texttt{while}\ e\ \texttt{do}\ s, \sigma\rangle \to \sigma}\ \textsc{Wh2}$$

Fig. 1. Small-step operational semantics for IMP.

2.1. IMP: a language for single programs

We use a simple imperative language, called IMP [16,17], which represents a regular general-purpose programming language, aimed at the development of single programs (as opposed to program families). IMP is a well established minimal language, used in teaching and research. We stress that IMP is used only for presentational purposes, and that the introduced systematic methodology is not limited to IMP or its features.

Syntax IMP is structured as a traditional imperative language into two syntactic categories: expressions and statements. Expressions include integer constants, variables, and binary operations, and statements include a "do-nothing" statement `skip`, assignments, statement sequences, conditional statements, and while loops. Its abstract syntax is summarized using the following context-free grammar:

$$\begin{array}{lcl} e & ::= & n \mid \texttt{x} \mid e_0 \oplus e_1 \\ s & ::= & \texttt{skip} \mid \texttt{x} := e \mid s_0\ ;\ s_1 \mid \\ & & \texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{else}\ s_1 \mid \texttt{while}\ e\ \texttt{do}\ s \end{array}$$

In the above, n stands for an integer constant, `x` stands for a variable name, and $\oplus$ stands for a binary operator. The precise choice of available operators is immaterial for the remainder of the paper. We denote by *Stm* and *Exp* the set of all statements, s, and expressions, e, generated by the above grammar. We use parentheses to resolve ambiguities in different syntactic categories. For this purpose we will write $\{\ldots\}$ for statements, and $(\ldots)$ for expressions.

Semantics A state of an IMP program is an abstraction of memory storage (a *store*) mapping variables to values (integer numbers), $Val = \mathbb{Z}$. We write $Store = Var \to Val$ to denote the set of all possible stores. IMP expressions are computed in a given store, denoted by σ below. A function $\mathcal{E} : Exp \times Store \to Val$ defined below by structural induction on e, maps an expression and a store to a value, thereby formalizing evaluation of expressions.

$$\begin{aligned} \mathcal{E}(n,\sigma) &= n \\ \mathcal{E}(\texttt{x},\sigma) &= \sigma(\texttt{x}) \\ \mathcal{E}(e_0 \oplus e_1,\sigma) &= \mathcal{E}(e_0,\sigma) \oplus \mathcal{E}(e_1,\sigma) \end{aligned}$$

Fig. 1 presents a small-step operational semantics for IMP. Following the convention popularized by C, we model Boolean values as integers, with zero interpreted as false and everything else as true (see rules If2 and Wh2, respectively, If1 and Wh1). Note that there are two types of rules. First, we have the typical small-step rules (for instance, Seq1 or Seq2), which rewrite a complex statement into a simpler one, possibly updating the store. Second, there are the completion rules, which execute a statement to completion producing a new store (for instance, Skip or Wh2).

Example 1. Let us consider the following IMP program S taken from [18]:

```
z := 3;
x := 1;
while (x<5) do {
    if (x=1) then y := 7 else y := z + 4;
    x := x + 1 }
```

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

where the body of `while` is marked with curly braces {...}. We evaluate S in the initial store $\sigma = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0, \mathtt{z} \mapsto 0]$. By using the rules in Fig. 1, we obtain the following final store:

$$\langle S, \sigma \rangle \rightarrow^* [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]$$

where $\rightarrow^*$ is defined to be the transitive and reflexive closure of $\rightarrow$. We will use the program S as a running example in the single-program analysis.

2.2. $\overline{\text{IMP}}$: a language for program families

Implementation of SPL architectures [1] relies on the existence of a variability mechanism [2] that allows early, or *staged*, configuration of program functionality (i.e., ability to configure program behavior at build-time or compile-time). This way, we can use a common code base to encode multiple variations of a software product, maximizing code reuse. An individual product is derived by specializing the multi-staged program, i.e. the common code base, at product derivation time, before it is built.

A simple form of two-staged computation involving a C-style preprocessor is the most common variability mechanism in practice [3]. We will now lift IMP from describing single programs to program families, admitting two-staged computation in this style. The compile-time computation is controlled by a product configuration k—a set of product *features* that should be included in the build process. A finite set $\mathbb{F}$ of Boolean variables, A, describes available features, $A \in \mathbb{F}$. A configuration, k, is a subset of *available* features: $k \subseteq \mathbb{F}$. We write $\mathbb{K}$ for the set of all valid configurations for a family program. We consider only valid configurations in the remainder of the paper.

The set of valid configurations is typically described by a *feature model* [19] or a configuration model in another similar notation [20]. In this paper we disregard syntactic representations of the set $\mathbb{K}$, as we are concerned with mathematical proofs more than with implementation details (so the set-theoretic view is simple and convenient). In practice, syntax of feature models can be easily related to sets of valid configurations [21]. An exhaustive account of feature modeling and domain modeling can be found in [2].

Syntax The programming language $\overline{\text{IMP}}$ is our two-stage extension of IMP. Its abstract syntax includes the same expression and statement productions as IMP, but we add a new compile-time-conditional statement, with keyword `#if`. It takes a condition over features (φ) and a statement (s) that should be executed (included in the product) if the condition is satisfied by the product configuration.

$$\begin{array}{rcl} s & ::= & \ldots \mid \#\mathtt{if}\ \varphi\ s \\ \varphi & ::= & A \in \mathbb{F} \mid \neg\varphi \mid \varphi_0 \wedge \varphi_1 \end{array}$$

We also add a syntactic category of Boolean expressions (φ) to write compile-time propositional logic formulae over features. We write, *FeatExp*, for the set of all Boolean expressions over features, and $\overline{Stm}$ for the set of all statements of $\overline{\text{IMP}}$. To stress the variability aspect, we write $\bar{s}$ to denote a statement from $\overline{Stm}$ (despite the notational overhead). The set of expressions *Exp* remains the same as for IMP. Observe that adding preprocessor directives to the abstract syntax of IMP was essentially a mechanical transformation of the grammar that will look similar for other, more complex languages.

Remark. The C preprocessor uses the following keywords: `#if`, `#ifdef`, and `#ifndef` to start a conditional construct; `#elif` and `#else` to create additional branches; and `#endif` to end a construct. Any of such preprocessor conditional constructs can be desugared and represented only by `#if` construct we use in this work, e.g. `#ifdef` φ s_0 `#else` s_1 `#endif` is translated into `#if` φ s_0 ; `#if` $\neg\varphi$ s_1.

Conditional constructs can be defined at the level of expressions as well, but in that case we must have a conditional-compilation expression with a choice between two different elements, e.g. $(\#\varphi\ ?\ e_0 : e_1\#)$, since there is no unit element for expressions. However, conditional constructs defined on arbitrary language elements could be translated into constructs that respect the appropriate syntactic structure of the language by code duplication [22,23]. We have made the choice to introduce variability at the level of statements purely for pedagogical reasons. This allows us to keep the presentation focussed and improves readability of definitions and proofs.

Semantics: from $\overline{\text{IMP}}$ to IMP $\overline{\text{IMP}}$'s semantics has two stages: first, given a configuration k compute an IMP program for a given product variant; second, execute the IMP program using regular IMP semantics. Below we present the first stage of $\overline{\text{IMP}}$'s semantics.

We capture the meaning of static conditional expressions over features using a *satisfiability relation*, $\models \subseteq \mathbb{K} \times \mathit{FeatExp}$, defined as:

$$\begin{array}{rcl} k \models A & \text{iff} & A \in k \\ k \models \neg\varphi & \text{iff} & k \not\models \varphi \\ k \models \varphi_0 \wedge \varphi_1 & \text{iff} & k \models \varphi_0 \wedge k \models \varphi_1 \end{array}$$

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

$$
\begin{aligned}
P[\![\texttt{skip}]\!]_k &= \texttt{skip}\\
P[\![\texttt{x} := e]\!]_k &= \texttt{x} := e\\
P[\![s_0 \texttt{ ; } s_1]\!]_k &= P[\![s_0]\!]_k \texttt{ ; } P[\![s_1]\!]_k\\
P[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]_k &= \texttt{if } e \texttt{ then } P[\![s_0]\!]_k \texttt{ else } P[\![s_1]\!]_k\\
P[\![\texttt{while } e \texttt{ do } s]\!]_k &= \texttt{while } e \texttt{ do } P[\![s]\!]_k\\
P[\![\texttt{\#if } \varphi\ s]\!]_k &= \begin{cases} P[\![s]\!]_k & k \vDash \varphi \\ \texttt{skip} & k \nvDash \varphi \end{cases}
\end{aligned}
$$

Fig. 2. Preprocessor from $\overline{\text{IMP}}$ to IMP for configuration k.

The semantics of the first stage of the computation—a simple preprocessor from $\overline{\text{IMP}}$ to IMP, is specified by the function $P : \overline{Stm} \to \mathbb{K} \to Stm$ in Fig. 2. The semantic function P copies all basic statements of $\overline{\text{IMP}}$ that are also IMP statements, and recursively pre-processes all sub-statements of compound statements. The last case checks whether a feature constraint is satisfied and, if so, it includes the guarded statement. Otherwise it reduces to `skip`, which has the effect of removing the guarded statement. Again, observe that the above rules are independent of the semantics of IMP, so specifying the semantics of the preprocessor is essentially a mechanical process.

Example 2. We now slightly modify the program S from Example 1 by adding some `#if` statements, and in this way obtain a new $\overline{\text{IMP}}$ program $\overline{S}$:

```
z := 3;
#if (A) x := 1;
#if (B) y := 7;
while (x < 5) do {
    if (x = 1) then y := 7 else y := z + 4;
    x := x + 1}
```

Let the set of all possible valid configurations be $\mathbb{K} = \{\{A\}, \{B\}, \{A, B\}\}$. Then the result of preprocessing $\overline{S}$ in configuration A, $P[\![\overline{S}]\!]_{\{A\}}$, is the program S from Example 1. We also have that $P[\![\overline{S}]\!]_{\{B\}}$ is the program:

```
z := 3;
y := 7;
while (x < 5) do {
    if (x = 1) then y := 7 else y := z + 4;
    x := x + 1}
```

We denote the above single program as S_B. We will use the program $\overline{S}$ as a running example in the family-program analysis.

3. Background: how to derive a single-program analysis

This section represents a brief summary of the ideas and concepts of abstract interpretation, and in particular of its calculational approach. For more elaborate treatments of these concepts, which are central in this paper, we refer to [15,17]. In [24] we give the proofs of all results presented here. We leave $\overline{\text{IMP}}$ aside in this section and work only with single programs and IMP in the following. We will systematically *derive* static analyses for IMP in a step-by-step compositional manner, using abstract interpretation.

3.1. Collecting semantics

We first introduce a *collecting semantics* for IMP, which is the starting point in abstract interpretation. A collecting semantics takes a program as an argument and then defines how to "collect" information of interest in the given program. It can be seen as an analysis that does not introduce any imprecision. Such an analysis is obviously *uncomputable*, i.e. it cannot be computed statically since IMP is a Turing complete language. Then, we introduce the notion of a *Galois connection*—a pair of functions capturing information loss between two domains. Finally, we demonstrate how to combine collecting semantics and Galois connections to derive approximate, albeit computable analyses, which can statically determine dynamic properties of programs. We use a constant propagation analysis for IMP to demonstrate this approach.

A *collecting semantics* mimics the behavior of the operational semantics (cf. Fig. 1), but with one important difference. Instead of working on *stores*, it works on *sets of stores*. In other words: our property of interest is the *possible* memories (modeled as a set of stores) that may arise at each program point. Furthermore, unknown program input can be modeled as *any* possible input (the set of stores in which a dedicated input variable can take on *any* run-time value). Finally, the set of stores is naturally ordered under the subset ordering, $\subseteq$. In this way, the collecting semantics can already be thought of as a fully precise (but uncomputable) analysis. Then the actual computable analyses can be defined as approximations of this semantics.

The collecting semantics for IMP is given in Fig. 3. Going from the operational semantics to the collecting semantics is straightforward. The function $\mathcal{C}[\![s]\!]$ captures the effect of executing statement s on a set of input stores, by computing the

$$
\begin{array}{rcl}
\mathcal{C}[\![\texttt{skip}]\!] & = & \lambda c.\, c \\
\mathcal{C}[\![\texttt{x} := e]\!] & = & \lambda c.\, \{\sigma[\texttt{x} \mapsto v] \mid \sigma \in c \ \wedge \ v \in \mathcal{C}'[\![e]\!]\{\sigma\}\} \\
\mathcal{C}[\![s_0 \ ; \ s_1]\!] & = & \mathcal{C}[\![s_1]\!] \circ \mathcal{C}[\![s_0]\!] \\
\mathcal{C}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] & = & \lambda c.\, \mathcal{C}[\![s_0]\!]\{\sigma \in c \mid 0 \notin \mathcal{C}'[\![e]\!]\{\sigma\}\} \ \cup \ \mathcal{C}[\![s_1]\!]\{\sigma \in c \mid 0 \in \mathcal{C}'[\![e]\!]\{\sigma\}\} \\
\mathcal{C}[\![\texttt{while } e \texttt{ do } s]\!] & = & \mathrm{lfp}\, \lambda\Phi.\lambda c.\, \{\sigma \in c \mid 0 \in \mathcal{C}'[\![e]\!]\{\sigma\}\} \ \cup \ \Phi(\mathcal{C}[\![s]\!]\{\sigma \in c \mid 0 \notin \mathcal{C}'[\![e]\!]\{\sigma\}\}) \\
 & & \\
\mathcal{C}'[\![n]\!] & = & \lambda c.\, \{n\} \\
\mathcal{C}'[\![\texttt{x}]\!] & = & \lambda c.\, \{\sigma(\texttt{x}) \mid \sigma \in c\} \\
\mathcal{C}'[\![e_0 \oplus e_1]\!] & = & \lambda c.\, \{v \mid v \in \{v_0\} \mathbin{\dot{\oplus}} \{v_1\} \ \wedge \ \sigma \in c \ \wedge \ v_0 \in \mathcal{C}'[\![e_0]\!]\{\sigma\} \ \wedge \ v_1 \in \mathcal{C}'[\![e_1]\!]\{\sigma\}\}
\end{array}
$$

Fig. 3. Collecting semantics $\mathcal{C}[\![s]\!] : 2^{Store} \to 2^{Store}$ and $\mathcal{C}'[\![e]\!] : 2^{Store} \to 2^{Val}$.

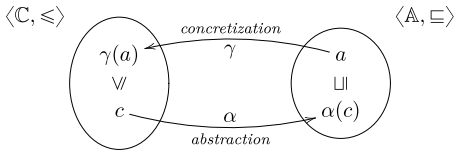

Fig. 4. A *Galois connection* between a *concrete*, $\langle \mathbb{C}, \leqslant \rangle$, and an *abstract* domain, $\langle \mathbb{A}, \sqsubseteq \rangle$.

set of possible output stores (memory contents after executing s). For instance, since the Skip rule does not modify the store, the corresponding case in the collecting semantics becomes the identity function on sets of stores: $\lambda c.\, c$. The `if` case results in the *union* of the effect from the two corresponding rules (If1 and If2) with a contribution from s_0 (for the stores where the condition evaluates to a non-zero value) and one from s_1 (for the stores where the condition evaluates to zero). The only slightly more complex case is that of the `while` statement which is now given in a standard *fixed-point formulation*. The case similarly combines the effects corresponding to the two rules (Wh1 and Wh2) although with an application of Φ to capture additional iterations of the `while` loop. Observe, that the subordinate function $\mathcal{C}'[\![e]\!]$ does the same exercise for expressions. The symbol $\dot{\oplus}$ denotes lifting of $\oplus$ to sets—an operator that produces a set of possible values of the expression for each combination of arguments from argument sets. Note that since the language is deterministic, if we evaluate an expression in a singleton set of input stores then a singleton value will be obtained, i.e. $\mathcal{C}'[\![e]\!]\{\sigma\}$ yields exactly one value (a singleton set). So we have $\mathcal{C}'[\![e]\!]\{\sigma\} = \{\mathcal{E}(e, \sigma)\}$, and we denote this fact by $(*)$. It can be proved by simple structural induction on expressions e.

Example 3. By using the rules in Fig. 3, we can calculate the collecting semantics of program S from Example 1 for an arbitrary set of input stores $c \in 2^{Store}$ as:

$$\mathcal{C}[\![S]\!]c = \{[\texttt{x} \mapsto 5, \texttt{y} \mapsto 7, \texttt{z} \mapsto 3]\}$$

and the collecting semantics of program S_B from Example 2 for the input set $c' = \{[\texttt{x} \mapsto 0, \texttt{y} \mapsto 0, \texttt{z} \mapsto 0], [\texttt{x} \mapsto 6, \texttt{y} \mapsto 0, \texttt{z} \mapsto 0]\}$ is:

$$\mathcal{C}[\![S_B]\!]c' = \{[\texttt{x} \mapsto 5, \texttt{y} \mapsto 7, \texttt{z} \mapsto 3], [\texttt{x} \mapsto 6, \texttt{y} \mapsto 7, \texttt{z} \mapsto 3]\}$$

The collecting semantics captures precisely all executions of the operational semantics. Formally:

Theorem 1 *(Correctness of collecting semantics).*

$$\forall s \in Stm, c \in 2^{Store} : \ \mathcal{C}[\![s]\!]c = \{\sigma' \mid \sigma \in c \wedge \langle s, \sigma \rangle \to^* \sigma'\}$$

Proof. By structural induction on statements s given in [24]. □

Given a statement s, we can show that the collecting semantics $\mathcal{C}[\![s]\!] : 2^{Store} \to 2^{Store}$, and in particular the fixed-point functional of the while rule: $\lambda\Phi.\lambda c.\, \{\sigma \in c \mid 0 \in \mathcal{C}'[\![e]\!]\{\sigma\}\} \cup \Phi(\mathcal{C}[\![s]\!]\{\sigma \in c \mid 0 \notin \mathcal{C}'[\![e]\!]\{\sigma\}\})$ are *monotone* functions over *complete lattices* (see [24] for proofs). Thus by Tarski's Fixed-Point Theorem, they admit a unique least fixed point. However, since these lattices have *infinite height*, it is not guaranteed that we can compute a fixed point in finite time.

3.2. Galois connection

A *Galois connection* is a pair of functions, $\alpha : \mathbb{C} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{C}$ (respectively known as the *abstraction* and *concretization* functions), connecting two partially ordered sets, $\langle \mathbb{C}, \leqslant \rangle$ and $\langle \mathbb{A}, \sqsubseteq \rangle$ (often called the *concrete* and *abstract* domain, respectively), such that:

$$\forall c \in \mathbb{C}, a \in \mathbb{A} : \quad \alpha(c) \sqsubseteq a \ \Leftrightarrow \ c \leqslant \gamma(a) \tag{1}$$

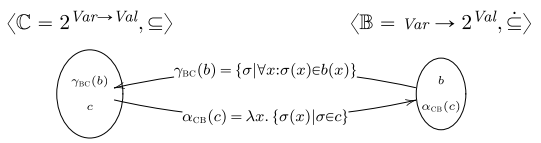

Fig. 5. Galois connection: $\langle 2^{Var \to Val}, \subseteq \rangle \underset{\alpha_{CB}}{\overset{\gamma_{BC}}{\leftrightarrows}} \langle Var \to 2^{Val}, \dot{\sqsubseteq} \rangle$.

which is often typeset as: $\langle \mathbb{C}, \leqslant \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$. Fig. 4 illustrates a Galois connection graphically. For a concrete domain $\mathbb{C}$, we define *abstraction* and *concretization* functions to and from a more abstract domain $\mathbb{A}$, where information has been abstracted away.

The seemingly innocent concept has a number of important properties [25]:

(GC1) α is *monotone*: i.e., $c \leqslant c' \Rightarrow \alpha(c) \sqsubseteq \alpha(c')$, for all $c, c' \in \mathbb{C}$;
(GC2) γ is *monotone*; i.e., $a \sqsubseteq a' \Rightarrow \gamma(a) \leqslant \gamma(a')$, for all $a, a' \in \mathbb{A}$;
(GC3) $\gamma \circ \alpha$ is *extensive*; i.e., $c \leqslant (\gamma \circ \alpha)(c)$, for all $c \in \mathbb{C}$;
(GC4) $\alpha \circ \gamma$ is *reductive*; i.e., $(\alpha \circ \gamma)(a) \sqsubseteq a$, for all $a \in \mathbb{A}$;
(GC5) If $\mathbb{A}$ and $\mathbb{C}$ are *complete lattices*, then α is a *complete join morphism* (CJM), i.e., we have $\alpha(\bigcup_{c \in \mathbb{C}} c) = \bigsqcup_{c \in \mathbb{C}} \alpha(c)$, where $\cup$ and $\sqcup$ represent lattice joins (least upper bounds) in $\mathbb{C}$ and $\mathbb{A}$, respectively.
(GC6) The composition of Galois connections is a Galois connection. If $\langle \mathbb{C}, \leqslant \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{B}, \subseteq \rangle$ and $\langle \mathbb{B}, \subseteq \rangle \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$ then $\langle \mathbb{C}, \leqslant \rangle \underset{\alpha' \circ \alpha}{\overset{\gamma \circ \gamma'}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$.

Due to this last closure property, abstraction can be split into several steps by composing successive Galois connections that incrementally over-approximate information. Collectively, properties (GC1)–(GC4) are equivalent to (1).

3.3. Deriving an abstracted analysis via a Galois connection

Let us now return to our IMP language and show how to use a Galois connection to approximate information yielding a less precise analysis (although, in this case, still intractable). Recall that the collecting semantics of a statement s works on sets of stores: it transforms sets of stores to sets of stores. Fig. 5 defines a Galois connection to abstract away information from sets of stores to multi-valued stores, so from $2^{Store} = 2^{Var \to Val}$ to $Var \to 2^{Val}$. Multi-valued stores are less precise than sets of stores, because they lose relational information about the values of different variables. Consider the (concrete) store set, $c = \{[x \mapsto 1, y \mapsto 2], [x \mapsto 2, y \mapsto 1]\}$, as an example. The abstraction function, α_{CB}, abstracts the store set c into $b = \alpha_{CB}(c) = [x \mapsto \{1, 2\}, y \mapsto \{1, 2\}]$. The abstract store b will now have "forgotten" which values of variable x go with which values of y. Now if the next statement computes, say, multiplication $x * y$, the analysis will *conservatively over-approximate* the set of possible values to $\{1, 2, 4\}$, admitting *spurious values*, 1 and 4, in addition to the precise answer: $\{2\}$. The approximate response is bound to include the precise answer; in other words, we have a *sound* analysis: $\{2\} \subseteq \{1, 2, 4\}$.

Not only *values* can be abstracted from $\mathbb{C}$ to $\mathbb{A}$ in a Galois connection $\langle \mathbb{C}, \leqslant \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$. In fact, also *functions* defined on the concrete domain, $f : \mathbb{C} \to \mathbb{C}$, can be abstracted to work on the abstract domain, $\alpha \circ f \circ \gamma = F : \mathbb{A} \to \mathbb{A}$. This process transforms an argument, $a \in \mathbb{A}$, in three simple steps: (1) *concretize* a, $\gamma(a) \in \mathbb{C}$; (2) *apply* f, $(f \circ \gamma)(a) \in \mathbb{C}$; and (3) *abstract* the result, $(\alpha \circ f \circ \gamma)(a) \in \mathbb{A}$. Also, if f is monotone, then its composition with a monotone α and γ is monotone. In general, any monotone over-approximation of the composition $\alpha \circ f \circ \gamma$ is sufficient for a sound analysis.

Cousot and Cousot [9] observed that even *fixed points* transfer from $\mathbb{C}$ to $\mathbb{A}$. If $\mathbb{C}$ and $\mathbb{A}$ are *complete lattices* and f is a *monotone* function on $\mathbb{C} \to \mathbb{C}$, then by the *fixed-point transfer theorem* [9]:

$$\alpha(\mathrm{lfp}\, f) \;\sqsubseteq\; \mathrm{lfp}\, F \;\sqsubseteq\; \mathrm{lfp}\, F^{\#} \tag{2}$$

where $F = \alpha \circ f \circ \gamma$ and $F^{\#}$ is some monotone, conservative *over*-approximation of F; formally: $F \dot{\sqsubseteq} F^{\#}$ (i.e., $\forall a \in \mathbb{A} : F(a) \sqsubseteq F^{\#}(a)$). Note that F represents the *best possible approximation* of f over the chosen abstract domain [25]. The above version of the fixed-point transfer theorem still lets us approximate the desired fixed point. Under the stronger assumption that $\alpha \circ f = F \circ \alpha$ then a stronger version of the theorem guarantees that no approximation of the fixed point is taking place: $\alpha(\mathrm{lfp}\, f) = \mathrm{lfp}\, F$ [9].

The approach to abstract interpretation adopted in this paper, known as the *calculational approach* [15], advocates simple algebraic manipulation to obtain a *direct expression* for the function, F (if, indeed, it exists); or, a *sound approximation* thereof, $F^{\#}$. It is thus a *systematic* (as in *"pen and paper"*) rather than *automatic* (as in *"computer generated"*) approach for *deriving* analyses.

In our case, we derive from, $\mathcal{C}[\![s]\!] : (2^{Var \to Val}) \to (2^{Var \to Val})$, a function working on the abstracted domain, $\alpha_{CB} \circ \mathcal{C}[\![s]\!] \circ \gamma_{BC} : (Var \to 2^{Val}) \to (Var \to 2^{Val})$. This step is crucial—we use the Galois connection to derive a more abstract semantics (and thus a more approximating analysis) from the less abstract semantics (here the collecting semantics). We want to obtain a *direct*

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{B}[\![\texttt{skip}]\!] & = & \lambda b.\, b & & & \\
\mathcal{B}[\![\texttt{x} := e]\!] & = & \lambda b.\, b[\texttt{x} \mapsto \mathcal{B}'[\![e]\!]b] & \mathcal{B}'[\![n]\!] & = & \lambda b.\, \{n\} \\
\mathcal{B}[\![s_0 \texttt{ ; } s_1]\!] & = & \mathcal{B}[\![s_1]\!] \circ \mathcal{B}[\![s_0]\!] & \mathcal{B}'[\![\texttt{x}]\!] & = & \lambda b.\, b(\texttt{x}) \\
\mathcal{B}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] & = & \lambda b.\, \mathcal{B}[\![s_0]\!]b \,\dot\cup\, \mathcal{B}[\![s_1]\!]b & \mathcal{B}'[\![e_0 \oplus e_1]\!] & = & \lambda b.\, \mathcal{B}'[\![e_0]\!]b \,\dot\oplus\, \mathcal{B}'[\![e_1]\!]b \\
\mathcal{B}[\![\texttt{while } e \texttt{ do } s]\!] & = & \mathrm{lfp}\, \lambda \Phi.\, \lambda b.\, b \,\dot\cup\, \Phi(\mathcal{B}[\![s]\!]b) & & &
\end{array}
$$

Fig. 6. Systematically derived *over-approximated* abstracted collecting semantics, $\mathcal{B}[\![s]\!] : (Var \to 2^{Val}) \to (Var \to 2^{Val})$ and $\mathcal{B}'[\![e]\!] : (Var \to 2^{Val}) \to 2^{Val}$.

expression for an over-approximation of $\alpha_{\mathrm{CB}} \circ \mathcal{C}[\![s]\!] \circ \gamma_{\mathrm{BC}}$ which we henceforth abbreviate, $\mathcal{B}[\![s]\!]$. Technically, the derivation is done by structural induction on s. Note that operators $\dot\cup$ and $\dot\subseteq$ are extended to functions: $f \,\dot\cup\, g = \lambda x.\, f(x) \cup g(x)$ and $f \,\dot\subseteq\, g = \forall x.\, f(x) \subseteq g(x)$.

Let us consider the derivation steps for the '`if`' statement. Since the derivations for most of the other cases are similar to this one, we present the calculation of $\mathcal{B}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]$ in detail. By using the definition of $\mathcal{C}$ for '`if`' in Fig. 3, a subsequent β-reduction, and the rule (GC5), we obtain:

$$
\begin{array}{l}
(\alpha_{\mathrm{CB}} \circ \mathcal{C}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] \circ \gamma_{\mathrm{BC}})(b) \\
\quad = \alpha_{\mathrm{CB}}(\mathcal{C}[\![s_0]\!]\{\sigma \in \gamma_{\mathrm{BC}}(b) \mid 0 \notin \mathcal{C}'[\![e]\!]\{\sigma\}\} \cup \\
\qquad\quad \mathcal{C}[\![s_1]\!]\{\sigma \in \gamma_{\mathrm{BC}}(b) \mid 0 \in \mathcal{C}'[\![e]\!]\{\sigma\}\}) \quad \text{(by def. of } \mathcal{C} \text{ in Fig. 4, and } \beta\text{-red.)} \\
\quad = \alpha_{\mathrm{CB}}(\mathcal{C}[\![s_0]\!]\{\sigma \in \gamma_{\mathrm{BC}}(b) \mid 0 \notin \mathcal{C}'[\![e]\!]\{\sigma\}\}) \,\dot\cup \\
\qquad\quad \alpha_{\mathrm{CB}}(\mathcal{C}[\![s_1]\!]\{\sigma \in \gamma_{\mathrm{BC}}(b) \mid 0 \in \mathcal{C}'[\![e]\!]\{\sigma\}\}) \quad (\alpha_{\mathrm{CB}} \text{ is a CJM and (GC5)}) \\
\quad \,\dot\subseteq\, \alpha_{\mathrm{CB}}(\mathcal{C}[\![s_0]\!](\gamma_{\mathrm{BC}}(b))) \,\dot\cup\, \alpha_{\mathrm{CB}}(\mathcal{C}[\![s_1]\!](\gamma_{\mathrm{BC}}(b))) \quad \text{(over-approximation)} \\
\quad \,\dot\subseteq\, \mathcal{B}[\![s_0]\!]b \,\dot\cup\, \mathcal{B}[\![s_1]\!]b \quad \text{(by IH, twice)}
\end{array}
$$

In the last two steps above, we first over-approximate the arguments of functions $\alpha_{\mathrm{CB}} \circ \mathcal{C}[\![s_0]\!]$ and $\alpha_{\mathrm{CB}} \circ \mathcal{C}[\![s_1]\!]$ by neglecting the value of the condition e, and then, we apply the inductive hypothesis (IH for short) twice to s_0 and s_1. So we calculated that $\mathcal{B}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] = \lambda b.\, \mathcal{B}[\![s_0]\!]b \,\dot\cup\, \mathcal{B}[\![s_1]\!]b$, which is an over-approximation of $\alpha_{\mathrm{CB}} \circ \mathcal{C}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] \circ \gamma_{\mathrm{BC}}$.

The function $\mathcal{B}'[\![e]\!] : (Var \to 2^{Val}) \to 2^{Val}$ is also derived for expressions from $\mathcal{C}'[\![e]\!] : (2^{Var \to Val}) \to 2^{Val}$, such that it is an over-approximation of $\mathcal{C}'[\![e]\!] \circ \gamma_{\mathrm{BC}}$. Note that we do not specify an abstraction function α in this case, since it represents an identity function on the domain 2^{Val}. We derive $\mathcal{B}'[\![e]\!]$ by structural induction on expressions e.

The resulting over-approximated abstracted collecting semantics $\mathcal{B}[\![s]\!]$ and $\mathcal{B}'[\![e]\!]$ are shown in Fig. 6. The following result shows how the approximate semantics $\mathcal{B}$ and $\mathcal{B}'$ are related to the concrete semantics $\mathcal{C}$ and $\mathcal{C}'$, and it holds by construction, i.e., by definitions of $\mathcal{B}[\![s]\!]$ and $\mathcal{B}'[\![e]\!]$.

Theorem 2 *(Soundness of approximate collecting semantics).*

(i) $\forall e \in Exp, b \in \mathbb{B} :\ (\mathcal{C}'[\![e]\!] \circ \gamma_{\mathrm{BC}})(b) \subseteq \mathcal{B}'[\![e]\!]b$
(ii) $\forall s \in Stm, b \in \mathbb{B} :\ (\alpha_{\mathrm{CB}} \circ \mathcal{C}[\![s]\!] \circ \gamma_{\mathrm{BC}})(b) \,\dot\subseteq\, \mathcal{B}[\![s]\!]\, b$

Example 4. By using the rules in Fig. 6, we can calculate the approximate collecting semantics of program S from Example 1 for different input multi-valued stores. Thus, for a zero initialized (Java-like) input store, we have:

$$\mathcal{B}[\![S]\!]([\texttt{x} \mapsto \{0\}, \texttt{y} \mapsto \{0\}, \texttt{z} \mapsto \{0\}]) = [\texttt{x} \mapsto \{1,2,3,\ldots\}, \texttt{y} \mapsto \{0,7\}, \texttt{z} \mapsto \{3\}]$$

and for an uninitialized (C-like) input store, we have:

$$\mathcal{B}[\![S]\!]([\texttt{x} \mapsto \mathit{Val}, \texttt{y} \mapsto \mathit{Val}, \texttt{z} \mapsto \mathit{Val}]) = [\texttt{x} \mapsto \{1,2,3,\ldots\}, \texttt{y} \mapsto \mathit{Val}, \texttt{z} \mapsto \{3\}]$$

Notice how the final stores computed by $\mathcal{B}[\![S]\!]$ over-approximate the corresponding final store computed by $\mathcal{C}[\![S]\!]$ in Example 3.

This is now starting to look like a conventional dataflow analysis [17]. However, it is still intractable. For example, consider the following program: $\texttt{x} := 1; \texttt{while(1) do x} := \texttt{x} + 1$. It will give rise to an *infinite* multi-valued abstract store $b = [x \mapsto \{1,2,3,\ldots\}]$. Our next Galois connection will remedy this in abstracting our abstract domain, $Var \to 2^{Val}$, even further into a domain with *finite* height, thereby guaranteeing an analysis computable with a Kleene fixed point iteration.

3.4. Deriving constant propagation analysis via another Galois connection

We aim to derive a *constant propagation analysis*, which establishes whether a variable has a constant value whenever the execution reaches a program point. Fig. 7 presents a Galois connection between $\mathbb{B} = \mathrm{Var} \to 2^{Val}$ and $\mathbb{A} = \mathrm{Var} \to \mathit{Const}$,

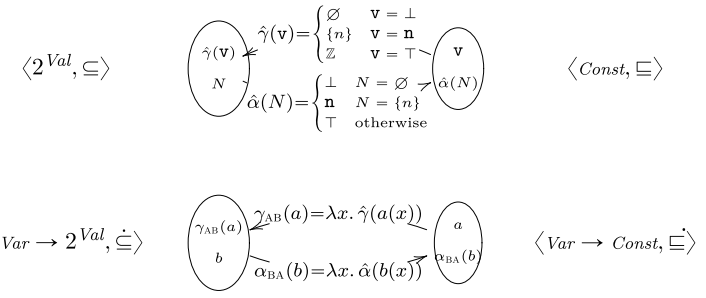

Fig. 7. Galois connection: $\langle 2^{Val}, \subseteq \rangle \underset{\hat{\alpha}}{\overset{\hat{\gamma}}{\leftrightarrows}} \langle Const, \sqsubseteq \rangle$ (top diagram) along with its pointwise lifting (bottom diagram): $\langle \mathbb{B} = Var \to 2^{Val}, \dot{\subseteq} \rangle \underset{\alpha_{BA}}{\overset{\gamma_{AB}}{\leftrightarrows}} \langle \mathbb{A} = Var \to Const, \dot{\sqsubseteq} \rangle$.

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathtt{skip}]\!] & = & \lambda a.\, a \\
\mathcal{A}[\![\mathtt{x} := e]\!] & = & \lambda a.\, a[\mathtt{x} \mapsto \mathcal{A}'[\![e]\!]a] \\
\mathcal{A}[\![s_0\ ;\ s_1]\!] & = & \mathcal{A}[\![s_1]\!] \circ \mathcal{A}[\![s_0]\!] \\
\mathcal{A}[\![\mathtt{if}\ e\ \mathtt{then}\ s_0\ \mathtt{else}\ s_1]\!] & = & \lambda a.\, \mathcal{A}[\![s_0]\!]a \mathbin{\dot{\sqcup}} \mathcal{A}[\![s_1]\!]a \\
\mathcal{A}[\![\mathtt{while}\ e\ \mathtt{do}\ s]\!] & = & \mathrm{lfp}\, \lambda \Phi.\, \lambda a.\, a \mathbin{\dot{\sqcup}} \Phi(\mathcal{A}[\![s]\!]a)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{A}'[\![n]\!] & = & \lambda a.\, \mathtt{n} \\
\mathcal{A}'[\![\mathtt{x}]\!] & = & \lambda a.\, a(\mathtt{x}) \\
\mathcal{A}'[\![e_0 \oplus e_1]\!] & = & \lambda a.\, \mathcal{A}'[\![e_0]\!]a \mathbin{\widehat{\oplus}} \mathcal{A}'[\![e_1]\!]a, \text{ where} \\
v_0 \mathbin{\widehat{\oplus}} v_1 & = & \begin{cases} \bot & \text{if } v_0 = \bot \vee v_1 = \bot \\ \mathtt{n}_0 \oplus \mathtt{n}_1 & \text{if } v_0 = \mathtt{n}_0 \wedge v_1 = \mathtt{n}_1 \\ \top & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 8. Constant propagation analysis $\mathcal{A}[\![s]\!] : (Var \to Const) \to (Var \to Const)$ and $\mathcal{A}'[\![e]\!] : (Var \to Const) \to Const$.

$\langle \mathbb{B}, \dot{\subseteq} \rangle \underset{\alpha_{BA}}{\overset{\gamma_{AB}}{\leftrightarrows}} \langle \mathbb{A}, \dot{\sqsubseteq} \rangle$, for abstracting the multi-valued store domain even further. It does so by approximating the value set of each individual variable with a constant propagation lattice $\langle Const, \sqsubseteq \rangle$, where $Const = Val \cup \{\bot, \top\}$ is partially ordered as follows: $\forall v \in Val.\, \bot \sqsubseteq v \sqsubseteq \top$, and $\forall v_1, v_2 \in Val.\, v_1 \sqsubseteq v_2$ iff $v_1 = v_2$. We use $\top$ to indicate that a variable is not constant, and $\bot$ to indicate that no information is available. All other elements show that a variable is constant with that particular value. The partial ordering $\sqsubseteq$ induces a least upper bound (*join*) operator, $\sqcup$, on the lattice elements, which is used to combine information during the analysis. For example, we have $\bot \sqcup 1 = 1$, $0 \sqcup 1 = \top$, etc. If we follow the systematic derivation steps for inferring $\mathcal{B}[\![s]\!]$ and $\mathcal{B}'[\![s]\!]$, we can finally derive a *computable* constant propagation analysis as an over-approximation of $\alpha_{BA} \circ \mathcal{B}[\![s]\!] \circ \gamma_{AB}$ and $\hat{\alpha} \circ \mathcal{B}'[\![s]\!] \circ \gamma_{AB}$, which we call $\mathcal{A}[\![s]\!]$ and $\mathcal{A}'[\![e]\!]$ respectively. See Fig. 8 for definitions of $\mathcal{A}[\![s]\!]$ and $\mathcal{A}'[\![e]\!]$, which are derived by structural induction on s and e respectively. We now show the derivation steps for the conditional statement:

$$
\begin{array}{l}
(\alpha_{BA} \circ \mathcal{B}[\![\mathtt{if}\ e\ \mathtt{then}\ s_0\ \mathtt{else}\ s_1]\!] \circ \gamma_{AB})(a) \\
\quad = \alpha_{BA}(\mathcal{B}[\![s_0]\!]\gamma_{AB}(a) \mathbin{\dot{\cup}} \mathcal{B}[\![s_1]\!]\gamma_{AB}(a)) \quad \text{(by def. of } \mathcal{B} \text{ in Fig. 6 and } \beta\text{-red.)} \\
\quad = \alpha_{BA}(\mathcal{B}[\![s_0]\!]\gamma_{AB}(a)) \mathbin{\dot{\sqcup}} \alpha_{BA}(\mathcal{B}[\![s_1]\!]\gamma_{AB}(a)) \quad (\alpha_{BA} \text{ is a CJM, and (GC5)}) \\
\quad \dot{\sqsubseteq} \mathcal{A}[\![s_0]\!]a \mathbin{\dot{\sqcup}} \mathcal{A}[\![s_1]\!]a \quad \text{(by IH, twice)} \\
\quad = \mathcal{A}[\![\mathtt{if}\ e\ \mathtt{then}\ s_0\ \mathtt{else}\ s_1]\!]a
\end{array}
$$

Since operators are functions, they too get abstracted by our Galois connection. Recall that our example uses $\dot{\oplus}$, the pointwise extension of the binary operator $\oplus$, defined as $V_0 \mathbin{\dot{\oplus}} V_1 = \{v_0 \oplus v_1 \mid v_0 \in V_0 \wedge v_1 \in V_1\}$. The abstract counterpart, $\widehat{\oplus}$, can be calculated by following the same recipe: $\hat{\alpha}(\hat{\gamma}(V) \mathbin{\dot{\oplus}} \hat{\gamma}(V')) \sqsubseteq V \mathbin{\widehat{\oplus}} V'$; i.e., by concretizing its arguments, performing the corresponding concrete operation, and finally abstracting the outcome. The resulting abstract operator, $\widehat{\oplus}$, can be computed effectively (in constant time) for all concrete binary operators (see Fig. 8). Finally, we write $\dot{\sqcup}$ to denote the pointwise join in the $Var \to Const$ lattice: $a_0 \mathbin{\dot{\sqcup}} a_1 = \lambda \mathtt{x}.\, a_0(\mathtt{x}) \sqcup a_1(\mathtt{x})$.

Since our domain now has a finite height, we have a tractable analysis. Indeed now the program: $\mathtt{x} := 1; \mathtt{while}\ (1)\ \mathtt{do}\ \mathtt{x} := \mathtt{x} + 1$ gives rise to a *finite* abstract store $a = [x \mapsto \top]$. Moreover, the soundness (correctness) of the constant propagation analysis follows by construction, i.e. by definition of $\mathcal{A}[\![s]\!]$ and $\mathcal{A}'[\![e]\!]$.

Theorem 3 *(Soundness of constant propagation analysis).*

(i) $\forall e \in Exp, a \in \mathbb{A}:\ (\hat{\alpha} \circ \mathcal{B}'[\![e]\!] \circ \gamma_{AB})(a) \sqsubseteq \mathcal{A}'[\![e]\!]a$
(ii) $\forall s \in Stm, a \in \mathbb{A}:\ (\alpha_{BA} \circ \mathcal{B}[\![s]\!] \circ \gamma_{AB})(a) \dot{\sqsubseteq} \mathcal{A}[\![s]\!]a$

Notice how Theorem 3 composes with the result of Theorem 2 yielding soundness of the analysis $\mathcal{A}$ not only with respect to $\mathcal{B}$, but also with respect to the collecting semantics $\mathcal{C}$.

Example 5. Returning to our running example program S, by using the rules in Fig. 8 we can calculate $\mathcal{A}[\![S]\!]$ for two different abstract input stores:

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

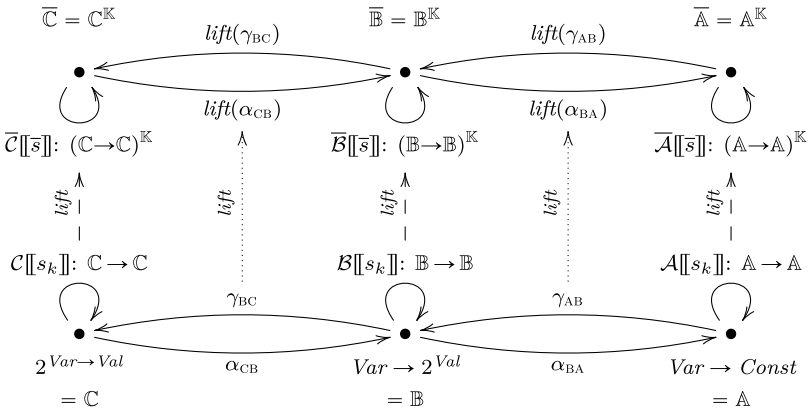

Fig. 9. Abstract interpretation of programs (bottom line) along with *lifted* "variational abstract interpretation" of SPLs (top line).

$$\mathcal{A}[\![S]\!]([x \mapsto 0, y \mapsto 0, z \mapsto 0]) = [x \mapsto \top, y \mapsto \top, z \mapsto 3]$$
$$\mathcal{A}[\![S]\!]([x \mapsto 0, y \mapsto 7, z \mapsto 0]) = [x \mapsto \top, y \mapsto 7, z \mapsto 3]$$

We may choose to implement the analysis in Fig. 8 directly. We may use Kleene's Fixed-Point Theorem to calculate local fixed point computations for loops iteratively. A more common approach is to implement a dataflow analysis as a set of dataflow equations. In Section 6, we will show how the dataflow equations can be *derived* from the analysis in Fig. 8.

4. Deriving a variability-aware program analysis

We are now ready to discuss how the analysis obtained in Section 3 can be effectively lifted to work on the level of program families (SPLs). We call this framework for systematic derivation of analyses for SPLs as *variational abstract interpretation*. This is illustrated by the commutative diagram in Fig. 9, which presents and relates the abstract interpretation of single programs and program families. The bottom part of the figure shows the derivation process for single programs presented in Section 3. The top part shows the same derivation process only lifted to work on SPLs. This top line of the diagram starts by defining the collecting semantics for the language with variability (in our case $\overline{\text{IMP}}$), and then it repeats the same abstraction steps as before but now at the level of program families. However, if we did this, we would almost completely ignore the artifacts accumulated during creation of the single-program analysis! The core idea of the *variational abstract interpretation* is that the analyses at the single-program level can be systematically lifted to work on the family level without rerunning the entire derivation process: you arrive at the same, provably sound lifted analysis by commutation of the diagram.

The final constant propagation $\mathcal{A}$ can be lifted to family-based constant propagation $\overline{\mathcal{A}}$ by applying a lifting combinator (*lift*) to $\mathcal{A}$ and performing simplifying calculations. In the following, we discuss how this is done in detail and obtain a correctness result. We show how the domains of analyses, the analyses themselves (the transfer functions), and the Galois connections are lifted to the family level. Two kinds of upward arrows (dashed and dotted) lift us from the single program world to the program family world in Fig. 9. There is a dashed upward arrow for lifting *analyses*, e.g. $\mathcal{A}[\![s]\!] : \mathbb{A}\to\mathbb{A}$ is lifted to $\overline{\mathcal{A}}[\![\overline{s}]\!] : (\mathbb{A}\to\mathbb{A})^{\mathbb{K}}$; and a dotted upward arrow for lifting *Galois connections*: $\mathbb{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{B}$ is *lifted* to $\overline{\mathbb{C}} \underset{lift(\alpha)}{\overset{lift(\gamma)}{\leftrightarrows}} \overline{\mathbb{B}}$.

4.1. Lifting domains

We first lift the semantic domains. Recall that $\mathbb{K}$ denotes a finite set of valid configurations. A domain, $(\mathbb{C}, \sqsubseteq)$, is lifted to a *variability domain*, $(\overline{\mathbb{C}}, \dot{\sqsubseteq})$, by taking $\overline{\mathbb{C}}$ to be $\mathbb{C}^{\mathbb{K}}$ (i.e., a tuple of $|\mathbb{K}|$ copies of $\mathbb{C}$, one for each valid configuration), and lifting the ordering $\dot{\sqsubseteq}$ configuration-wise; i.e., $\bar{c} \dot{\sqsubseteq} \bar{c}' \equiv_{def}$ for all $k \in \mathbb{K} : \pi_k(\bar{c}) \sqsubseteq \pi_k(\bar{c}')$, where π_k selects the kth component of a tuple.

4.2. Lifting analyses

The lifted domain representation, $\overline{\mathbb{A}} = \mathbb{A}^{\mathbb{K}}$, and Fig. 9 suggest that the lifted analysis, $\overline{\mathcal{A}}$, should be one complex function from $\mathbb{A}^{\mathbb{K}}$ to $\mathbb{A}^{\mathbb{K}}$. However, it turns out that using a tuple of $|\mathbb{K}|$ independent simple functions, $(\mathbb{A}\to\mathbb{A})^{\mathbb{K}}$, is a much better alternative. This models our intuition that lifting corresponds to running $|\mathbb{K}|$ analyses in parallel. Functions of type $(\mathbb{A}\to\mathbb{A})^{\mathbb{K}}$ are essentially a well behaved subset of functions from $\mathbb{A}^{\mathbb{K}}$ to $\mathbb{A}^{\mathbb{K}}$—namely those, for which the kth component of the function value only depends on the kth component of the argument. This causes no problems with interference between configurations, which is critical for correctness of lifting.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

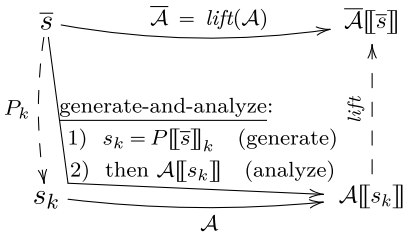

Fig. 10. Generate-and-analyze vs. lifted analysis.

To help readability, we introduce notational conventions that allow using tuples of functions, as if they were functions on tuples. We admit direct application of tuples of functions to tuples of arguments: if $\overline{f} : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ is a tuple of functions indexed by elements of $\mathbb{K}$, we write $\overline{f}(\bar{a})$ to mean the tuple of $|\mathbb{K}|$ values created by applying each function to the corresponding argument in the tuple of arguments: $\prod_{k\in K} \pi_k(\overline{f})(\pi_k(\bar{a}))$. Similarly, we overload the λ-abstraction notation, so creating a tuple of functions looks like creating a function on tuples: we write $\lambda\bar{a}.\prod_{k\in\mathbb{K}} f(\pi_k(\bar{a}))$ to mean $\prod_{k\in\mathbb{K}} \lambda a_k.f(a_k)$.

The straightforward way of analyzing a configuration, k, of an SPL, $\bar{s}$, using a conventional single-program analysis, $\mathcal{A}$, is to first *generate* a product, $s_k = P[\![\bar{s}]\!]_k$, using the preprocessor; then, *analyze* the generated product, s_k, using the conventional analysis: $\mathcal{A}[\![s_k]\!]$. This two stage process is depicted in Fig. 10 (cf. arrow labelled *generate-and-analyze*). However, it only analyzes *one* configuration of the SPL (the arrow ends up at the bottom part of Fig. 10). Thus, to define the analysis on the family level, we need to execute $\mathcal{A}$ for all valid configurations $k \in \mathbb{K}$. If $\mathcal{A}[\![s]\!] = \mathbb{A} \to \mathbb{A}$ is a single analysis function, then we require that its lifted version $\overline{\mathcal{A}}[\![\bar{s}]\!] : (\mathbb{A} \to \mathbb{A})^{\mathbb{K}}$ satisfies the following:

$$\overline{\mathcal{A}}[\![\bar{s}]\!] = \lambda\bar{a}.\prod_{k\in\mathbb{K}} \mathcal{A}[\![P[\![\bar{s}]\!]_k]\!](\pi_k(\bar{a})) \tag{3}$$

The equation stipulates that running the aggregate analysis $\overline{\mathcal{A}}$ must be equivalent to running the original analysis $\mathcal{A}$ for each variant separately, after deriving it using the preprocessor P. An analysis $\overline{\mathcal{A}}$ satisfying (3) transforms a lifted store, $\bar{a} \in \overline{\mathbb{A}} = \mathbb{A}^{\mathbb{K}}$, into another lifted store, $\bar{a}' = \prod_{k\in\mathbb{K}} \mathcal{A}[\![P[\![\bar{s}]\!]_k]\!]\pi_k(\bar{a})$, of the same type. In other words, $\overline{\mathcal{A}}$ is a transformer between aggregated state of all configurations on entry to a given program point to an aggregated state of all configurations on the exit from that program point.

This specification of lifting works for any single-program analysis, not just for constant propagation. We formulate it as a general analysis-independent and language-independent combinator.

Definition 4. The generic lifting of analysis, $\mathcal{X} : \mathbb{X} \to \mathbb{X}$, is:

$$lift(\mathcal{X})[\![\bar{s}]\!] \;=\; \lambda\bar{x}.\prod_{k\in\mathbb{K}} \mathcal{X}[\![P[\![\bar{s}]\!]_k]\!](\pi_k(\bar{x}))$$

In Fig. 9 the dashed upward arrows represent applications of the above lifting combinator *lift*. They transform an analysis function (solid loop arrows at the bottom), to a *family-based* analysis (solid loop arrows at the top).

Unfortunately, Definition 4 cannot be used as a direct definition of analysis $\overline{\mathcal{A}}$ as it still depends on the single-program analysis. Implementing $\overline{\mathcal{A}}$ naively, directly following (3), would merely apply the conventional analysis $|\mathbb{K}|$ times (one for each $k \in \mathbb{K}$). While this would give the correct results, it is not what we wanted! This analysis will generate and analyze all individual configurations one by one. We seek a family-based analysis that will analyze all configurations simultaneously. The question is how to obtain a definition of $\overline{\mathcal{A}}$ that is independent of $\mathcal{A}$, yet satisfies Eq. (3). To achieve this we simplify Eq. (3), similarly to how we simplified the composition of analysis functions with Galois connections. As such, our lifting is calculational in nature, following the natural steps in abstract interpretation. If we perform the composition and simplify the resulting expression systematically, we can eliminate the intermediate product generation step and obtain a direct expression for the lifted analysis as shown in Fig. 11 corresponding to the top arrow in Fig. 10 (cf. Contribution (C1)).

We now illustrate how the calculation of $\overline{\mathcal{A}}[\![\bar{s}]\!]$ given in Fig. 11 is done for conditional, iteration, and compile-time conditional statements. Again the derivation is performed by structural induction on statements $\bar{s}$. The calculation looks similar for the other cases. Note that the pointwise join operator $\dot{\sqcup}$ defined on the lattice $\mathbb{A}$ in Section 3 is lifted to $\ddot{\sqcup}$. It is defined on the lattice $\mathbb{A}^{\mathbb{K}}$ as follows: $\bar{a}_0 \ddot{\sqcup} \bar{a}_1 = \prod_{k\in\mathbb{K}} \pi_k(\bar{a}_0) \dot{\sqcup} \pi_k(\bar{a}_1)$.

Consider the derivation for the '`if`' statement.

$$\begin{aligned}
&\lambda\bar{a}.\prod_{k\in\mathbb{K}} \mathcal{A}[\![P[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]_k]\!](\pi_k(\bar{a})) \\
&= \lambda\bar{a}.\prod_{k\in\mathbb{K}} \mathcal{A}[\![\texttt{if } e \texttt{ then } P[\![s_0]\!]_k \texttt{ else } P[\![s_1]\!]_k]\!](\pi_k(\bar{a})) \quad \text{(by def. of } P\text{, Fig. 2)}
\end{aligned}$$

$$
\begin{aligned}
\overline{\mathcal{A}}[\![\texttt{skip}]\!] &= \lambda\bar{a}.\,\bar{a}\\
\overline{\mathcal{A}}[\![\texttt{x} := e]\!] &= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}(\pi_k(\bar{a}))[\texttt{x}\mapsto\pi_k(\overline{\mathcal{A}'}[\![e]\!]\bar{a})]\\
\overline{\mathcal{A}}[\![s_0\ ;\ s_1]\!] &= \overline{\mathcal{A}}[\![s_1]\!]\circ\overline{\mathcal{A}}[\![s_0]\!]\\
\overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!] &= \lambda\bar{a}.\,\overline{\mathcal{A}}[\![s_0]\!]\bar{a}\ddot{\sqcup}\overline{\mathcal{A}}[\![s_1]\!]\bar{a}\\
\overline{\mathcal{A}}[\![\texttt{while } e \texttt{ do } s]\!] &= \mathrm{lfp}\,\lambda\overline{\Phi}.\lambda\bar{a}.\,\bar{a}\ddot{\sqcup}\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\bar{a})\\
\\
\overline{\mathcal{A}}[\![\#\texttt{if } \varphi\ s]\!] &= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}\begin{cases}\pi_k(\overline{\mathcal{A}}[\![s]\!]\bar{a}) & k\vDash\varphi\\ \pi_k(\bar{a}) & k\nvDash\varphi\end{cases}\\
\overline{\mathcal{A}'}[\![n]\!] &= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}\mathtt{n}\\
\overline{\mathcal{A}'}[\![\texttt{x}]\!] &= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}\pi_k(\bar{a})(\texttt{x})\\
\overline{\mathcal{A}'}[\![e_0\oplus e_1]\!] &= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}\pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\bar{a})\ \widehat{\oplus}\ \pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\bar{a})
\end{aligned}
$$

Fig. 11. Lifted constant propagation analysis $\overline{\mathcal{A}}[\![\bar{s}]\!] : ((Var \to Const) \to (Var \to Const))^{\mathbb{K}}$ and $\overline{\mathcal{A}'}[\![e]\!] : ((Var \to Const) \to Const)^{\mathbb{K}}$.

$$
\begin{aligned}
&= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}(\mathcal{A}[\![P[\![s_0]\!]_k]\!](\pi_k(\bar{a})))\ \dot{\sqcup}\ (\mathcal{A}[\![P[\![s_1]\!]_k]\!](\pi_k(\bar{a}))) \quad \text{(by def. of } \mathcal{A} \text{ in Fig. 8, and } \beta\text{-red.)}\\
&= \lambda\bar{a}.\,\prod_{k\in\mathbb{K}}(\pi_k(\overline{\mathcal{A}}[\![s_0]\!]\bar{a}))\ \dot{\sqcup}\ (\pi_k(\overline{\mathcal{A}}[\![s_1]\!]\bar{a})) \quad \text{(by IH, twice)}\\
&= \lambda\bar{a}.\,\overline{\mathcal{A}}[\![s_0]\!]\bar{a}\ddot{\sqcup}\overline{\mathcal{A}}[\![s_1]\!]\bar{a} \quad \text{(by def. of } \ddot{\sqcup})\\
&= \overline{\mathcal{A}}[\![\texttt{if } e \texttt{ then } s_0 \texttt{ else } s_1]\!]
\end{aligned}
$$

For the derivation of '`while`', we first define an abstraction α_k which projects a tuple to a particular configuration entry k, along with the corresponding concretization function γ_k:

$$
\begin{aligned}
&\alpha_k : \mathbb{A}^{\mathbb{K}} \to \mathbb{A}, \text{where } \alpha_k(\bar{a}) = \pi_k(\bar{a})\\
&\gamma_k : \mathbb{A} \to \mathbb{A}^{\mathbb{K}}, \text{where } \gamma_k(a) = \prod_{k'\in\mathbb{K}}\begin{cases}a & k = k'\\ \dot{\top} & k \neq k'\end{cases}
\end{aligned}
$$

where $\dot{\top} = \lambda x.\,\top \in \mathbb{A}$, and $\ddot{\top} = \prod_{k\in\mathbb{K}}\dot{\top} \in \mathbb{A}^{\mathbb{K}}$. Such projections are well known to form a Galois connection $\langle\mathbb{A}^{\mathbb{K}}, \ddot{\sqsubseteq}\rangle \underset{\alpha_k}{\overset{\gamma_k}{\leftrightarrows}} \langle\mathbb{A}, \dot{\sqsubseteq}\rangle$. In particular it is a Galois insertion since $(\alpha_k \circ \gamma_k)(a) = a$. We then lift this Galois connection to a higher-order Galois connection given by two monotone transfer functions [25]:

$$
\begin{aligned}
&\alpha_{\to}(\overline{\Phi}) = \alpha_k \circ \overline{\Phi} \circ \gamma_k, \text{ for } \alpha_{\to} : (\mathbb{A}^{\mathbb{K}} \xrightarrow{m} \mathbb{A}^{\mathbb{K}}) \to \mathbb{A} \xrightarrow{m} \mathbb{A}\\
&\gamma_{\to}(\Phi) = \gamma_k \circ \Phi \circ \alpha_k, \text{ for } \gamma_{\to} : (\mathbb{A} \xrightarrow{m} \mathbb{A}) \to \mathbb{A}^{\mathbb{K}} \xrightarrow{m} \mathbb{A}^{\mathbb{K}}
\end{aligned}
$$

where we write $X \xrightarrow{m} Y$ for the domain of monotone functions from X to Y. In order to use the stronger version of the fixed-point theorem [9]:

$$\alpha_{\to}(\mathrm{lfp}\, f) = \mathrm{lfp}\, F \tag{4}$$

where $f : \mathbb{A}^{\mathbb{K}} \xrightarrow{m} \mathbb{A}^{\mathbb{K}}$ and $F : \mathbb{A} \xrightarrow{m} \mathbb{A}$, we need to show that the assumption: $\alpha_{\to} \circ f = F \circ \alpha_{\to}$ holds. In our case f is the fixed-point functional in the definition of $\overline{\mathcal{A}}[\![\texttt{while } e \texttt{ do } s]\!]$ in Fig. 11 and F is the fixed-point functional in the definition of $\mathcal{A}[\![\texttt{while } e \texttt{ do } s]\!]$ in Fig. 8. First we show the assumption:

$$
\begin{aligned}
&\alpha_{\to} \circ (\lambda\overline{\Phi}.\lambda\bar{a}.\,\bar{a}\ddot{\sqcup}\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\bar{a}))\\
&\quad= \lambda\overline{\Phi}.\,\alpha_k \circ (\lambda\bar{a}.\,\bar{a}\ddot{\sqcup}\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\bar{a})) \circ (\lambda a.\,\gamma_k(a)) \quad \text{(by def. of } \circ, \alpha_{\to}, \eta\text{-expan.)}\\
&\quad= \lambda\overline{\Phi}.\,\lambda a.\,\alpha_k(\gamma_k(a)\ddot{\sqcup}\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!](\gamma_k(a)))) \quad \text{(by def. of } \circ, \text{ and } \beta\text{-red.)}\\
&\quad= \lambda\overline{\Phi}.\,\lambda a.\,a\,\dot{\sqcup}\,\alpha_k(\overline{\Phi}(\overline{\mathcal{A}}[\![s]\!](\gamma_k(a)))) \quad \text{(Galois insertion, } \alpha_k \text{ is a CJM)}\\
&\quad= \lambda\overline{\Phi}.\,\lambda a.\,a\,\dot{\sqcup}\,\alpha_k(\overline{\Phi}(\prod_{k'\in\mathbb{K}}\mathcal{A}[\![P[\![s]\!]_{k'}]\!](\pi_{k'}(\gamma_k(a))))) \quad \text{(by IH)}\\
&\quad= \lambda\overline{\Phi}.\,\lambda a.\,a\,\dot{\sqcup}\,\pi_k(\overline{\Phi})(\mathcal{A}[\![P[\![s]\!]_k]\!](\pi_k(\gamma_k(a)))) \quad \text{(by def. of } \alpha_k)\\
&\quad= \lambda\overline{\Phi}.\,\lambda a.\,a\,\dot{\sqcup}\,\pi_k(\overline{\Phi})(\mathcal{A}[\![P[\![s]\!]_k]\!]a) \quad \text{(by def. of } \alpha_k, \text{ Galois insertion)}
\end{aligned}
$$

By using definitions of π_k, γ_k and $\circ$, we further obtain:

$$
\begin{aligned}
&= \lambda\overline{\Phi}.\, \lambda a.\, a \dot{\sqcup} (\pi_k \circ \overline{\Phi} \circ \gamma_k)(\mathcal{A}[\![P[\![s]\!]_k]\!]a) \\
&= \lambda\overline{\Phi}.\, (\lambda\Phi.\, \lambda a.\, a \dot{\sqcup} \Phi(\mathcal{A}[\![P[\![s]\!]_k]\!]a))(\alpha_k \circ \overline{\Phi} \circ \gamma_k) \quad \text{(by def. of } \alpha_k, \beta\text{-exp.)} \\
&= (\lambda\Phi.\, \lambda a.\, a \dot{\sqcup} \Phi(\mathcal{A}[\![P[\![s]\!]_k]\!]a)) \circ (\lambda\overline{\Phi}.\, \alpha_k \circ \overline{\Phi} \circ \gamma_k) \quad \text{(by def. of } \circ) \\
&= (\lambda\Phi.\, \lambda a.\, a \dot{\sqcup} \Phi(\mathcal{A}[\![P[\![s]\!]_k]\!]a)) \circ \alpha_{\to} \quad \text{(by def. of } \alpha_{\to})
\end{aligned}
$$

As a consequence, we can now apply the stronger fixed-point theorem given in (4) to show:

$$
\begin{aligned}
&\alpha_{\to}(\overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!]) \\
&\quad = \alpha_{\to}(\mathrm{lfp}\, \lambda\overline{\Phi}.\, \lambda\bar{a}.\, \bar{a} \ddot{\sqcup} \overline{\Phi}(\overline{\mathcal{A}}[\![s]\!]\bar{a})) \quad \text{(by def. of } \overline{\mathcal{A}}) \\
&\quad = \mathrm{lfp}\, \lambda\Phi.\, \lambda a.\, a \dot{\sqcup} \Phi(\mathcal{A}[\![P[\![s]\!]_k]\!]a) \quad \text{(by the fixed-point transfer theorem (4))} \\
&\quad = \mathcal{A}[\![\texttt{while}\ e\ \texttt{do}\ P[\![s]\!]_k]\!] \quad \text{(by def. of } \mathcal{A}) \\
&\quad = \mathcal{A}[\![P[\![\texttt{while}\ e\ \texttt{do}\ s]\!]_k]\!] \quad \text{(by def. of } P)
\end{aligned}
$$

Finally we can use this equality to obtain:

$$
\begin{aligned}
&\lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \mathcal{A}[\![P[\![\texttt{while}\ e\ \texttt{do}\ s]\!]_k]\!](\pi_k(\bar{a})) \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \alpha_{\to}(\overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!])(\pi_k(\bar{a})) \quad \text{(by above equality)} \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} (\alpha_k \circ \overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!] \circ \gamma_k)(\pi_k(\bar{a})) \quad \text{(by def. of } \alpha_{\to}) \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \alpha_k \left(\overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!] \left(\prod_{k'\in\mathbb{K}} \begin{cases} \pi_k(\bar{a}) & k = k' \\ \dot{\top} & k \neq k' \end{cases} \right) \right) \quad \text{(by def. of } \gamma_k \text{ and } \circ) \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \alpha_k \left(\prod_{k'\in\mathbb{K}} (\pi_{k'}(\overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!])) \left(\begin{cases} \pi_k(\bar{a}) & k = k' \\ \dot{\top} & k \neq k' \end{cases} \right) \right) \quad \text{(by def. of fun. tuple appl.)} \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} (\pi_k(\overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!]))(\pi_k(\bar{a})) \quad \text{(by def. of } \alpha_k) \\
&\quad = \lambda\bar{a}.\, \overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!]\bar{a} \quad \text{(by def. of appl.)} \\
&\quad = \overline{\mathcal{A}}[\![\texttt{while}\ e\ \texttt{do}\ s]\!] \quad \text{(by } \eta\text{-reduce)}
\end{aligned}
$$

Consider the case of '`#if`' statement.

$$
\begin{aligned}
&\lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \mathcal{A}[\![P[\![\#\texttt{if}\ \varphi\ s]\!]_k]\!](\pi_k(\bar{a})) \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \begin{cases} \mathcal{A}[\![P[\![s]\!]_k]\!](\pi_k(\bar{a})) & k \vDash \varphi \\ \mathcal{A}[\![\texttt{skip}]\!](\pi_k(\bar{a})) & k \nvDash \varphi \end{cases} \quad \text{(by def. of } P \text{ in Fig. 2)} \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \begin{cases} \mathcal{A}[\![P[\![s]\!]_k]\!](\pi_k(\bar{a})) & k \vDash \varphi \\ \pi_k(\bar{a}) & k \nvDash \varphi \end{cases} \quad \text{(by def. of } \mathcal{A} \text{ in Fig. 8)} \\
&\quad = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{\mathcal{A}}[\![s]\!]\bar{a}) & k \vDash \varphi \\ \pi_k(\bar{a}) & k \nvDash \varphi \end{cases} \quad \text{(by IH)} \\
&\quad = \overline{\mathcal{A}}[\![\#\texttt{if}\ \varphi\ s]\!]
\end{aligned}
$$

A lifted analysis $\overline{\mathcal{A}'}[\![e]\!]$ is also derived for expressions, such that $\overline{\mathcal{A}'}[\![e]\!] = \lambda\bar{a}.\, \prod_{k\in\mathbb{K}} \mathcal{A}'[\![e]\!](\pi_k(\bar{a}))$. The derivation is by structural induction on e. We only consider the case of binary operations.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

$$
\begin{aligned}
&\lambda\bar{a}.\prod_{k\in\mathbb{K}}\mathcal{A}'[\![e_0\oplus e_1]\!](\pi_k(\bar{a}))\\
&=\lambda\bar{a}.\prod_{k\in\mathbb{K}}(\lambda a.\,\mathcal{A}'[\![e_0]\!]a\mathbin{\widehat{\oplus}}\mathcal{A}'[\![e_1]\!]a)(\pi_k(\bar{a}))\quad\text{(by def. of }\mathcal{A}'\text{in Fig. 8)}\\
&=\lambda\bar{a}.\prod_{k\in\mathbb{K}}\mathcal{A}'[\![e_0]\!](\pi_k(\bar{a}))\mathbin{\widehat{\oplus}}\mathcal{A}'[\![e_1]\!](\pi_k(\bar{a}))\quad(\beta\text{-reduction})\\
&=\lambda\bar{a}.\prod_{k\in\mathbb{K}}\pi_k(\overline{\mathcal{A}'}[\![e_0]\!]\bar{a})\mathbin{\widehat{\oplus}}\pi_k(\overline{\mathcal{A}'}[\![e_1]\!]\bar{a})\quad\text{(by IH, twice)}\\
&=\overline{\mathcal{A}'}[\![e_0\oplus e_1]\!]
\end{aligned}
$$

The correctness of the lifted analysis $\overline{\mathcal{A}}$ and $\overline{\mathcal{A}'}$ shown in Fig. 11 holds by construction (cf. Contribution (C2)).

Theorem 5 *(Correctness of lifting the constant propagation analysis).*

(i) $\forall e\in Exp.\ \overline{\mathcal{A}'}[\![e]\!]=\lambda\bar{a}.\prod_{k\in\mathbb{K}}\mathcal{A}'[\![e]\!](\pi_k(\bar{a}))$
(ii) $\forall\bar{s}\in\overline{Stm}.\ \overline{\mathcal{A}}[\![\bar{s}]\!]=\lambda\bar{a}.\prod_{k\in\mathbb{K}}\mathcal{A}[\![P[\![\bar{s}]\!]_k]\!](\pi_k(\bar{a}))$

The equality signs in this theorem furthermore capture that lifting has introduced no approximation: the family-based analyses obtained this way are as precise as running the original analysis for each configuration individually. We prove that $\overline{\mathcal{A}}$ and $\overline{\mathcal{A}'}$ are monotone in [24].

Example 6. By using the rules in Fig. 11, we can calculate $\overline{\mathcal{A}}$ for the lifted program $\bar{S}$ from Example 2 at different lifted input stores, where $\mathbb{K}=\{\{A\},\{B\},\{A,B\}\}$. We assume a convention here that the first component of the lifted store corresponds to configuration $\{A\}$, the second to $\{B\}$, and the third to $\{A,B\}$. For example, for a $\mathbb{K}$-tuple where all variables have non-constant value, $\top$, we have:

$$
\begin{aligned}
&\overline{\mathcal{A}}[\![\bar{S}]\!]\big([\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto\top],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto\top],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto\top]\big)\\
&=\big([\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3]\big)
\end{aligned}
$$

So the result of analyzing $\bar{S}$ using $\overline{\mathcal{A}}$ is that for configuration $\{A\}$ only $\mathtt{z}$ has the constant value 3, while for configurations $\{B\}$ and $\{A,B\}$ both $\mathtt{y}$ and $\mathtt{z}$ have constant values 7 and 3, respectively.

5. Variational abstract interpretation

We now look again at the variational abstract interpretation framework illustrated in Fig. 9, and present an alternative way to derive a lifted analysis. Then, we prove that for any analysis the diagram in Fig. 9 is commutative (Theorem 6). We end the section by summarizing the basic steps in the proposed framework.

5.1. Many routes to family-based analysis

If we want to prove soundness of the lifted analysis $\overline{\mathcal{A}}$ using the classical abstract interpretation approach, we should devise a collecting semantics $\overline{\mathcal{C}}$ and a Galois connection relating them. Then, we need to follow the same incremental process as in Section 3, and define a chain of Galois connections:

$$\langle\overline{\mathbb{C}},\dot{\sqsubseteq}\rangle\ \underset{\overline{\alpha_{\mathrm{CB}}}}{\overset{\overline{\gamma_{\mathrm{BC}}}}{\leftrightarrows}}\ \langle\overline{\mathbb{B}},\ddot{\sqsubseteq}\rangle\ \underset{\overline{\alpha_{\mathrm{BA}}}}{\overset{\overline{\gamma_{\mathrm{AB}}}}{\leftrightarrows}}\ \langle\overline{\mathbb{A}},\dddot{\sqsubseteq}\rangle$$

Subsequently, we need to compute $\overline{\mathcal{A}}$ by composing these Galois connections with $\overline{\mathcal{C}}$ and prove that the resulting analysis is identical to the lifting of $\mathcal{A}$, so that the diagram in Fig. 9 commutes. A detailed development taking this route is available in the technical report [24]. But there is an easier route! Instead of devising the collecting semantics at family level, $\overline{\mathcal{C}}$, and then a sequence of Galois connections, we can obtain them all by lifting the corresponding operations from the single-program level.

The transfer functions $\mathcal{C}$ and $\mathcal{B}$ can be lifted to $\overline{\mathcal{C}}$ and $\overline{\mathcal{B}}$ like $\mathcal{A}$ was lifted to a family-based analysis $\overline{\mathcal{A}}$ in Section 5. We can pointwise lift a Galois connection α,γ using the combinator *lift* defined as:

$$lift(\alpha)=\lambda\bar{c}.\prod_{k\in\mathbb{K}}\alpha(\pi_k(\bar{c})),\qquad lift(\gamma)=\lambda\bar{a}.\prod_{k\in\mathbb{K}}\gamma(\pi_k(\bar{a}))$$

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

This way, no invention of new analyses for the family level is needed. Instead, all analyses can be uniformly lifted and composed. This is by no means automatic, but it is systematic (calculational); it does not require any design effort, as the original analysis is a sufficient source of information for obtaining the family-based analysis.

The following theorem states that the result of lifting the final single-program analysis is equivalent to lifting and recalculating all intermediate steps. This result does not depend on any particular analysis. It states that if a static single-program analysis $\mathcal{X}[\![s]\!]$ is obtained from a more concrete analysis $\mathcal{Y}[\![s]\!]$ by applying a Galois connection and simplifying, then the lifting of this analysis can be soundly obtained by applying lifted Galois connections to the lifting of $\mathcal{Y}$. Effectively, we prove that the diagram of Fig. 9 commutes (cf. Contribution (C3)). It is sound to develop the single program analysis and lift it as in Section 4.2, instead of lifting the collecting semantics and developing the entire analysis anew at the family level.

Theorem 6. *If for all single programs s we have that* $\alpha \circ \mathcal{Y}[\![s]\!] \circ \gamma \mathrel{\dot{\sqsubseteq}} \mathcal{X}[\![s]\!]$ *then also for each program* $\bar{s}$ *with variability, we have:*

$$lift(\alpha) \circ lift(\mathcal{Y})[\![\bar{s}]\!] \circ lift(\gamma) \mathrel{\ddot{\sqsubseteq}} lift(\mathcal{X})[\![\bar{s}]\!]$$

Proof. Assume that for all $s \in Stm$: $\alpha \circ \mathcal{Y}[\![s]\!] \circ \gamma \mathrel{\dot{\sqsubseteq}} \mathcal{X}[\![s]\!]$. Let $\bar{s} \in \overline{Stm}$ be given.

$$\begin{aligned}
&lift(\alpha) \circ lift(\mathcal{Y})[\![\bar{s}]\!] \circ lift(\gamma) \\
&= \lambda \bar{y}.\, (lift(\alpha) \circ lift(\mathcal{Y})[\![\bar{s}]\!] \circ lift(\gamma))(\bar{y}) \quad (\eta\text{-expansion}) \\
&= \lambda \bar{y}.\, lift(\alpha)\Big(\prod_{k\in\mathbb{K}} \mathcal{Y}[\![P[\![\bar{s}]\!]_k]\!](\pi_k(lift(\gamma)(\bar{y})))\Big) \quad (\text{by def. of } \circ \text{ and } lift(\mathcal{Y})) \\
&= \lambda \bar{y}.\, lift(\alpha)\Big(\prod_{k\in\mathbb{K}} \mathcal{Y}[\![P[\![\bar{s}]\!]_k]\!](\pi_k(\prod_{k'\in\mathbb{K}} \gamma(\pi_{k'}(\bar{y}))))\Big) \quad (\text{by def. of } lift(\gamma)) \\
&= \lambda \bar{y}.\, lift(\alpha)\Big(\prod_{k\in\mathbb{K}} \mathcal{Y}[\![P[\![\bar{s}]\!]_k]\!](\gamma(\pi_k(\bar{y})))\Big) \quad (\text{by def. of } \pi_k) \\
&= \lambda \bar{y}.\, \prod_{k\in\mathbb{K}} \alpha(\mathcal{Y}[\![P[\![\bar{s}]\!]_k]\!](\gamma(\pi_k(\bar{y})))) \quad (\text{by def. of } lift(\alpha)) \\
&\mathrel{\ddot{\sqsubseteq}} \lambda \bar{y}.\, \prod_{k\in\mathbb{K}} \mathcal{X}[\![P[\![\bar{s}]\!]_k]\!](\pi_k(\bar{y})) \quad (\text{by assumption}) \\
&= lift(\mathcal{X})[\![\bar{s}]\!] \quad (\text{by def. of } lift) \qquad \square
\end{aligned}$$

Moreover, if no approximation is introduced during the derivation of a single-program analysis $\mathcal{X}$ (so that $\alpha \circ \mathcal{Y}[\![s]\!] = \mathcal{X}[\![s]\!] \circ \alpha$) then lifting introduces no additional abstraction at the family level: $lift(\alpha) \circ lift(\mathcal{Y})[\![\bar{s}]\!] = lift(\mathcal{X})[\![\bar{s}]\!] \circ lift(\alpha)$. With this general theorem, the soundness for $\overline{\text{IMP}}$ now follows as a corollary from Theorems 2, 3, 5 and 6 (cf. Contribution (C4)):

Corollary 7 *(Soundness). For all* $\bar{s} \in \overline{Stm}$ *and* $\bar{a} \in \mathbb{A}^{\mathbb{K}}$*:*

$$lift(\alpha_{\text{BA}} \circ \alpha_{\text{CB}}) \circ lift(\mathcal{C})[\![\bar{s}]\!] \circ lift(\gamma_{\text{BC}} \circ \gamma_{\text{AB}})(\bar{a}) \mathrel{\ddot{\sqsubseteq}} lift(\mathcal{A})[\![\bar{s}]\!](\bar{a}) = \overline{\mathcal{A}}[\![\bar{s}]\!](\bar{a})$$

5.2. Summary of the steps in the variational abstract interpretation

Let us summarize the methodology of developing analyses of program families. We want to highlight the abstract steps and results of our method independently of the IMP language. The first three steps are the traditional steps of calculational abstract interpretation:

1. Develop formal operational semantics for your language.
2. Design collecting semantics for your language. Show equivalence of the operational and collecting semantics. Steps 1–2 are often given for existing established languages.
3. Specify a series of abstractions applied to the semantics in the form of Galois connections and compose them with the collecting semantics to obtain a single-program analysis. The calculation of compositions includes developing an inductive proof that the resulting analysis is sound.

Once the single-program analysis is established we set off to develop the family-based analysis:

4. Extend the syntax of the language with a preprocessor, and give semantics to the preprocessor P mapping syntactic constructs with variability to syntactic constructs without variability.
5. Apply the lifting combinator *lift* to the analysis calculated in step 3 above.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

$$
\begin{aligned}
[\![\texttt{skip}^{\ell}]\!]_{\text{out}} &= [\![\texttt{skip}^{\ell}]\!]_{\text{in}} \\
[\![\texttt{x} :=^{\ell} e]\!]_{\text{out}} &= [\![\texttt{x} :=^{\ell} e]\!]_{\text{in}}[\texttt{x} \mapsto \mathcal{A}'[\![e]\!][\![\texttt{x} :=^{\ell} e]\!]_{\text{in}}] \\
[\![s_0^{\ell_0} ;^{\ell} s_1^{\ell_1}]\!]_{\text{out}} &= [\![s_1^{\ell_1}]\!]_{\text{out}} \\
[\![s_1^{\ell_1}]\!]_{\text{in}} &= [\![s_0^{\ell_0}]\!]_{\text{out}} \\
[\![s_0^{\ell_0}]\!]_{\text{in}} &= [\![s_0^{\ell_0} ;^{\ell} s_1^{\ell_1}]\!]_{\text{in}} \\
[\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]_{\text{out}} &= [\![s_0^{\ell_0}]\!]_{\text{out}} \dot{\sqcup} [\![s_1^{\ell_1}]\!]_{\text{out}} \\
[\![s_0^{\ell_0}]\!]_{\text{in}} &= [\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]_{\text{in}} \\
[\![s_1^{\ell_1}]\!]_{\text{in}} &= [\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]_{\text{in}} \\
[\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\text{out}} &= [\![s^{\ell_0}]\!]_{\text{in}} \\
[\![s^{\ell_0}]\!]_{\text{in}} &= [\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\text{in}} \dot{\sqcup} [\![s^{\ell_0}]\!]_{\text{out}}
\end{aligned}
$$

Fig. 12. Dataflow equations for constant propagation of Fig. 8.

Remark. In the paper we only applied *lift* to transfer functions, which were endofunctions. This is not a requirement. For example, when lifting expression semantics, we had to lift functions that given a store argument produce a simple value as a result (see [24]). So variational abstract interpretation can be applied not only to languages expressing computations (state transfers), but also to others, for example constraint languages.

6. Simplify the resulting function to obtain a lifted analysis that is formulated independently of the original single-program analysis. Soundness of the lifted analysis at the family level now follows from combining the calculations in steps 3 and 6 with Theorem 6.

In Section 6.3, we will add five optional extra steps to this process in case one would like to incorporate *variational abstraction* in the analysis (in order to essentially trade precision at the variability level for extra speed).

6. Implementing efficient lifted analyses

We will now show how to *derive* dataflow equations from the constant propagation analysis (Fig. 8 from Section 3). Then, we show how to *derive* the lifted dataflow equations from the lifted analysis (Fig. 11 from Section 4). Next, we show how to insert variability-aware abstractions in the lifted analysis. Hereafter, we consider optimization; in particular, how to exploit equivalent analysis information in configurations which can speed up a lifted analysis dramatically. Finally, we provide an evidence that lifted analyses (with or without variability abstractions) may be significantly faster than the naive brute-force implementation (analyzing configurations one at a time).

6.1. Deriving dataflow equations

To implement the analysis defined in Fig. 8, we can extract the corresponding dataflow equations and then use an iterative algorithm to obtain the fixed-point solution to the generated equation system [17]. Dataflow equations are used to specify and relate information that is true on entry and exit of a statement (program point) to information present in statements from which control can flow to the statement of interest. We assume that individual statements have been uniquely labelled with labels, ℓ, to distinguish the individual flow to and from them and adapt $\mathcal{A}$ to work over such labelled statements. The corresponding dataflow equations are shown in Fig. 12. The transformation from Fig. 8 to Fig. 12 is essentially mechanical. For each statement s^{ℓ} we generate two flow variables $[\![s^{\ell}]\!]_{\text{in}}$ and $[\![s^{\ell}]\!]_{\text{out}}$ for the input and output abstract stores, respectively. Then for each statement we simply write down that the input and output variable are related by an expression of the right-hand-side of the corresponding domain transformer in Fig. 8, where the input variable is substituted for the parameter, and the output variable for the value of the function (the same could be done for all expressions, but for brevity we refer directly to the semantics of expressions $\mathcal{A}'[\![e]\!]$ in Fig. 12). Observe that in the while equations the fixed point operator is stripped, and the value of the output variable is used for the recursive reference. The iterative algorithm for computing the analysis result using these equations will handle the fixed point in the while rule at the meta-level. The iteration starts from the bottom value of the semantic domain assigned to all flow variables (if we disregard input), and stops when a fixed point is reached.

Example 7. Let us consider a labelled version of the program S:

$$
\begin{aligned}
&\texttt{z} :=^{1} 3;^{2} \\
&\texttt{x} :=^{3} 1;^{4} \\
&\texttt{while}^{5}\,(\texttt{x} < 5)\,\texttt{do}\,\{ \\
&\quad \texttt{if}^{6}\,(\texttt{x} = 1)\,\texttt{then}\,\texttt{y} :=^{7} 7\,\texttt{else}\,\texttt{y} :=^{8} \texttt{z} + 4;^{9} \\
&\quad \texttt{x} :=^{10} \texttt{x} + 1\}
\end{aligned}
$$

$$
\begin{array}{lcllcllcl}
[\![s^1]\!]_{\text{out}} & = & [\![s^1]\!]_{\text{in}}[\mathtt{z} \mapsto \mathcal{A}'[\![3]\!]] & [\![s^3]\!]_{\text{in}} & = & [\![s^1]\!]_{\text{out}} & [\![s^3]\!]_{\text{out}} & = & [\![s^3]\!]_{\text{in}}[\mathtt{x} \mapsto \mathcal{A}'[\![1]\!]] \\
[\![s^5]\!]_{\text{in}} & = & [\![s^3]\!]_{\text{out}} & [\![s^5]\!]_{\text{out}} & = & [\![s^9]\!]_{\text{in}} = [\![s^6]\!]_{\text{in}} & [\![s^9]\!]_{\text{in}} & = & [\![s^5]\!]_{\text{in}} \sqcup [\![s^9]\!]_{\text{out}} \\
[\![s^6]\!]_{\text{out}} & = & [\![s^7]\!]_{\text{out}} \sqcup [\![s^8]\!]_{\text{out}} & [\![s^{10}]\!]_{\text{in}} & = & [\![s^6]\!]_{\text{out}} & [\![s^9]\!]_{\text{out}} & = & [\![s^{10}]\!]_{\text{out}} \\
[\![s^7]\!]_{\text{in}} & = & [\![s^8]\!]_{\text{in}} = [\![s^6]\!]_{\text{in}} & [\![s^7]\!]_{\text{out}} & = & [\![s^7]\!]_{\text{in}}[\mathtt{y} \mapsto \mathcal{A}'[\![7]\!]] & [\![s^8]\!]_{\text{out}} & = & [\![s^8]\!]_{\text{in}}[\mathtt{y} \mapsto \mathcal{A}'[\![\mathtt{z}+4]\!][\![s^8]\!]_{\text{in}}]
\end{array}
$$

Fig. 13. Dataflow equations for Example 7.

l	$[\![s^l]\!]_{\text{in}}$	$[\![s^l]\!]_{\text{out}}$
1	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, z \mapsto \top]$	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3]$
3	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, z \mapsto 3]$	$[\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3]$
5	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, z \mapsto 3]$	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3]$
6	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, z \mapsto 3]$	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]$
7	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, z \mapsto 3]$	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]$
8	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, z \mapsto 3]$	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]$
10	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, z \mapsto 3]$	$[\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]$

Fig. 14. A solution to the dataflow equations for Example 7 (some labels omitted).

The analysis is defined by abstract stores $[\![s^l]\!]_{\text{in}}, [\![s^l]\!]_{\text{out}} : \mathbb{A} = (\mathit{Var} \to \mathit{Const})$ for all statements s^l in the program. Since every statement is uniquely determined by a label l, for simplicity we will denote them as s^l. For example, the statement $\mathtt{x} :=^3 1$ will be denoted as s^3. Let us suppose that at the start of the program an initial abstract store is available, where any variable can have an arbitrary value, i.e., we have $[\![s^2]\!]_{\text{in}} = [\![s^1]\!]_{\text{in}} = [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top]$. Fig. 13 lists some of the dataflow equations we obtain for this program. After the first two assignments, by using the equations in Fig. 13 we obtain:

$$
\begin{aligned}
[\![s^1]\!]_{\text{out}} = [\![s^3]\!]_{\text{in}} &= [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3] \\
[\![s^3]\!]_{\text{out}} &= [\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3]
\end{aligned}
$$

In Fig. 14, we show input and output abstract stores for some of the statements in S that satisfy our dataflow equations for constant propagation.

Note that after the assignment statement with label 10 the values of `y` and `z` are constants 7 and 3, respectively. But after the `while` loop with label 5, only `z` is constant.

Formally, a solution to the dataflow equations is sound with respect to the derived analysis.

Theorem 8 *(Soundness of dataflow analysis). For all s^ℓ, such that $[\![s^\ell]\!]_{\text{in}}$, $[\![s^\ell]\!]_{\text{out}}$ satisfies the dataflow equations in Fig. 12:*

$$\mathcal{A}[\![s^\ell]\!]([\![s^\ell]\!]_{\text{in}}) \mathrel{\dot\sqsubseteq} [\![s^\ell]\!]_{\text{out}}$$

Proof. The proof is by structural induction on s^ℓ. We consider the most involved cases.

Case $\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}$:

$$
\begin{aligned}
&\mathcal{A}[\![\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}]\!]([\![\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}]\!]_{\text{in}}) \\
&= \mathcal{A}[\![s_0^{\ell_0}]\!]([\![\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}]\!]_{\text{in}}) \mathbin{\dot\sqcup} \\
&\qquad \mathcal{A}[\![s_1^{\ell_1}]\!]([\![\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}]\!]_{\text{in}}) \quad \text{(by def. of } \mathcal{A} \text{ in Fig. 8)} \\
&= \mathcal{A}[\![s_0^{\ell_0}]\!]([\![s_0^{\ell_0}]\!]_{\text{in}}) \mathbin{\dot\sqcup} \mathcal{A}[\![s_1^{\ell_1}]\!]([\![s_1^{\ell_1}]\!]_{\text{in}}) \quad \text{(by def. of } [\![s_0^{\ell_0}]\!]_{\text{in}}, [\![s_1^{\ell_1}]\!]_{\text{in}} \text{ in Fig. 12)} \\
&\mathrel{\dot\sqsubseteq} [\![s_0^{\ell_0}]\!]_{\text{out}} \mathbin{\dot\sqcup} [\![s_1^{\ell_1}]\!]_{\text{out}} \quad \text{(by IH, twice)} \\
&= [\![\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}]\!]_{\text{out}} \quad \text{(by def. of } [\![\mathtt{if}^\ell\ e\ \mathtt{then}\ s_0^{\ell_0}\ \mathtt{else}\ s_1^{\ell_1}]\!]_{\text{out}} \text{ in Fig. 12)}
\end{aligned}
$$

Case $\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}$: First by (inner) induction on n, we can prove that

$$\mathfrak{F}^n(\ddot\bot)([\![\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}]\!]_{\text{in}} \mathbin{\dot\sqcup} [\![s^{\ell_0}]\!]_{\text{out}}) \mathrel{\dot\sqsubseteq} [\![\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}]\!]_{\text{out}} \tag{5}$$

for all $n \geqslant 0$, where $\mathfrak{F} = \lambda\Phi.\lambda a.\, a \mathbin{\ddot\sqcup} \Phi(\mathcal{A}[\![s^{\ell_0}]\!]a)$, and $\ddot\bot = \lambda a.\, \dot\bot$. Here $\mathfrak{F}$ represents the fixed-point functional from the definition of $\mathcal{A}$ for `while` e `do` s in Fig. 8. Now we have:

$$
\begin{aligned}
&\mathcal{A}[\![\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}]\!]([\![\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}]\!]_{\text{in}}) \\
&= (\mathrm{lfp}\,\mathfrak{F})([\![\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}]\!]_{\text{in}}) \quad \text{(by def. of } \mathcal{A} \text{ in Fig. 8)} \\
&= (\ddot\bigsqcup_i \mathfrak{F}^i(\ddot\bot))([\![\mathtt{while}^\ell\ e\ \mathtt{do}\ s^{\ell_0}]\!]_{\text{in}}) \quad \text{(by Kleene's fixed-point theorem)}
\end{aligned}
$$

$$
\begin{array}{rcl}
[\![\texttt{skip}^{\ell}]\!]_{\overline{\text{out}}} & = & [\![\texttt{skip}^{\ell}]\!]_{\overline{\text{in}}} \\
\underline{\forall k \in \mathbb{K}}\colon \pi_k([\![\texttt{x} :=^{\ell} e^{\ell_0}]\!]_{\overline{\text{out}}}) & = & \pi_k([\![\texttt{x} :=^{\ell} e^{\ell_0}]\!]_{\overline{\text{in}}})[\texttt{x} \mapsto \pi_k(\overline{\mathcal{A}'}[\![e^{\ell_0}]\!][\![\texttt{x} :=^{\ell} e^{\ell_0}]\!]_{\overline{\text{in}}})] \\
[\![s_0^{\ell_0} \;;^{\ell}\; s_1^{\ell_1}]\!]_{\overline{\text{out}}} & = & [\![s_1^{\ell_1}]\!]_{\overline{\text{out}}} \\
[\![s_1^{\ell_1}]\!]_{\overline{\text{in}}} & = & [\![s_0^{\ell_0}]\!]_{\overline{\text{out}}} \\
[\![s_0^{\ell_0}]\!]_{\overline{\text{in}}} & = & [\![s_0^{\ell_0} \;;^{\ell}\; s_1^{\ell_1}]\!]_{\overline{\text{in}}} \\
[\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]_{\overline{\text{out}}} & = & [\![s_0^{\ell_0}]\!]_{\overline{\text{out}}} \ddot{\cup} [\![s_1^{\ell_1}]\!]_{\overline{\text{out}}} \\
[\![s_0^{\ell_0}]\!]_{\overline{\text{in}}} & = & [\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]_{\overline{\text{in}}} \\
[\![s_1^{\ell_1}]\!]_{\overline{\text{in}}} & = & [\![\texttt{if}^{\ell}\ e\ \texttt{then}\ s_0^{\ell_0}\ \texttt{else}\ s_1^{\ell_1}]\!]_{\overline{\text{in}}} \\
[\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\overline{\text{out}}} & = & [\![s^{\ell_0}]\!]_{\overline{\text{in}}} \\
[\![s^{\ell_0}]\!]_{\overline{\text{in}}} & = & [\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\overline{\text{in}}} \ddot{\cup} [\![s^{\ell_0}]\!]_{\overline{\text{out}}} \\
\underline{\forall k \in \mathbb{K}}\colon \pi_k([\![\#\texttt{if}^{\ell}\ \varphi\ s^{\ell_0}]\!]_{\overline{\text{out}}}) & = & \pi_k([\![s^{\ell_0}]\!]_{\overline{\text{out}}}) \quad \text{if} \quad \underline{k \vDash \varphi} \\
\underline{\forall k \in \mathbb{K}}\colon \pi_k([\![\#\texttt{if}^{\ell}\ \varphi\ s^{\ell_0}]\!]_{\overline{\text{out}}}) & = & \pi_k([\![\#\texttt{if}^{\ell}\ \varphi\ s^{\ell_0}]\!]_{\overline{\text{in}}}) \quad \text{if} \quad \underline{k \nvDash \varphi} \\
\underline{\forall k \in \mathbb{K}}\colon \pi_k([\![s^{\ell_0}]\!]_{\overline{\text{in}}}) & = & \pi_k([\![\#\texttt{if}^{\ell}\ \varphi\ s^{\ell_0}]\!]_{\overline{\text{in}}}) \quad \text{if} \quad \underline{k \vDash \varphi}
\end{array}
$$

Fig. 15. Dataflow equations for lifted constant propagation of Fig. 11.

$$
\begin{array}{ll}
k \in \mathbb{K}\colon \pi_k([\![s^1]\!]_{\overline{\text{out}}}) = \pi_k([\![s^1]\!]_{\overline{\text{in}}})[\texttt{x} \mapsto \overline{\mathcal{A}'}[\![3]\!][\![s^1]\!]_{\overline{\text{in}}}] & [\![s^3]\!]_{\overline{\text{in}}} = [\![s^1]\!]_{\overline{\text{out}}} \\
k \in \{\{A\},\{A,B\}\}\colon \pi_k([\![s^3]\!]_{\overline{\text{out}}}) = \pi_k([\![s^4]\!]_{\overline{\text{out}}}) & \pi_{\{B\}}([\![s^3]\!]_{\overline{\text{out}}}) = [\![s^3]\!]_{\overline{\text{in}}} \\
k \in \{\{A\},\{A,B\}\}\colon \pi_k([\![s^4]\!]_{\overline{\text{in}}}) = \pi_k([\![s^3]\!]_{\overline{\text{in}}}) & \\
k \in \{\{A\},\{A,B\}\}\colon \pi_k([\![s^4]\!]_{\overline{\text{out}}}) = \pi_k([\![s^4]\!]_{\overline{\text{in}}})[\texttt{x} \mapsto \overline{\mathcal{A}'}[\![1]\!][\![s^4]\!]_{\overline{\text{in}}}] & [\![s^6]\!]_{\overline{\text{in}}} = [\![s^3]\!]_{\overline{\text{out}}} \\
k \in \{\{B\},\{A,B\}\}\colon \pi_k([\![s^6]\!]_{\overline{\text{out}}}) = \pi_k([\![s^7]\!]_{\overline{\text{out}}}) & \pi_{\{A\}}([\![s^6]\!]_{\overline{\text{out}}}) = [\![s^6]\!]_{\overline{\text{in}}} \\
k \in \{\{B\},\{A,B\}\}\colon \pi_k([\![s^7]\!]_{\overline{\text{in}}}) = \pi_k([\![s^6]\!]_{\overline{\text{in}}}) & \\
k \in \{\{B\},\{A,B\}\}\colon \pi_k([\![s^7]\!]_{\overline{\text{out}}}) = \pi_k([\![s^7]\!]_{\overline{\text{in}}})[\texttt{y} \mapsto \overline{\mathcal{A}'}[\![7]\!][\![s^7]\!]_{\overline{\text{in}}}] & [\![s^9]\!]_{\overline{\text{out}}} = [\![s^{13}]\!]_{\overline{\text{in}}} \\
[\![s^{13}]\!]_{\overline{\text{in}}} = [\![s^9]\!]_{\overline{\text{in}}} \ddot{\cup} [\![s^{13}]\!]_{\overline{\text{out}}} & [\![s^{10}]\!]_{\overline{\text{in}}} = [\![s^{13}]\!]_{\overline{\text{in}}} = [\![s^{11}]\!]_{\overline{\text{in}}} = [\![s^{12}]\!]_{\overline{\text{in}}} \\
k \in \mathbb{K}\colon \pi_k([\![s^{14}]\!]_{\overline{\text{out}}}) = \pi_k([\![s^{14}]\!]_{\overline{\text{in}}})[\texttt{x} \mapsto \pi_k(\overline{\mathcal{A}'}[\![\texttt{x}+1]\!][\![s^{14}]\!]_{\overline{\text{in}}})] & \\
k \in \mathbb{K}\colon \pi_k([\![s^{11}]\!]_{\overline{\text{out}}}) = \pi_k([\![s^{11}]\!]_{\overline{\text{in}}})[\texttt{y} \mapsto \overline{\mathcal{A}'}[\![7]\!][\![s^{11}]\!]_{\overline{\text{in}}}] &
\end{array}
$$

Fig. 16. Lifted dataflow equations for Example 8.

$$
\begin{aligned}
&= \dot{\sqcup}_i \mathfrak{F}^i(\ddot{\bot})[\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\text{in}} \quad \text{(by def. of } \ddot{\cup}\text{, and } \beta\text{-reduction)} \\
&\dot{\sqsubseteq} \dot{\sqcup}_i \mathfrak{F}^i(\ddot{\bot})([\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\text{in}} \ddot{\cup} [\![s^{\ell_0}]\!]_{\text{out}}) \quad \text{(by mono. of } \mathfrak{F}^i(\ddot{\bot})\text{)} \\
&\dot{\sqsubseteq} [\![\texttt{while}^{\ell}\ e\ \texttt{do}\ s^{\ell_0}]\!]_{\text{out}} \quad \text{(by Eq. (5))} \qquad \square
\end{aligned}
$$

The resulting constant propagation analysis is the same as the dataflow analysis presented in, e.g., [17], but with one crucial difference; the one presented here has been *systematically derived* using the abstract interpretation framework, resulting in an analysis whose soundness (correctness) follows by construction.

6.2. Deriving lifted dataflow equations

Just like in Section 6.1, we can use the definition of $\overline{\mathcal{A}}$ in Fig. 11 to derive lifted dataflow equations. This is a fairly mechanical process that results in the equations of Fig. 15 (compare to Fig. 11 and Fig. 12). Now, for each statement s^{ℓ} we generate two flow variables $[\![s^{\ell}]\!]_{\overline{\text{in}}}$ and $[\![s^{\ell}]\!]_{\overline{\text{out}}}$ for the input and output lifted stores, respectively.

Example 8. Let us consider a labelled version of the program $\overline{S}$, which we use as a running example in the lifted analysis (cf. Contribution (C6)).

```
z :=1 3;2
#if3 (A) x :=4 1;5
#if6 (B) y :=7 7;8
while9 (x<5) do{
    if10 (x=1) then y :=11 7 else y :=12 z + 4;13
    x :=14 x+1}
```

where the set of all possible valid configurations $\mathbb{K}$ is $\{\{A\},\{B\},\{A,B\}\}$. The analysis is defined by $[\![s^l]\!]_{\overline{\text{in}}}, [\![s^l]\!]_{\overline{\text{out}}} : \mathbb{A}^{\mathbb{K}} = (\mathit{Var} \to \mathit{Const})^{\mathbb{K}}$, which represent abstract stores for all possible configurations before and after the statement s^l. We assume that in the initial abstract store all variables can have arbitrary values, i.e., $[\![s^1]\!]_{\overline{\text{in}}} = \big([\texttt{x} \mapsto \top, \texttt{y} \mapsto \top, \texttt{z} \mapsto \top], [\texttt{x} \mapsto \top, \texttt{y} \mapsto \top, \texttt{z} \mapsto \top], [\texttt{x} \mapsto \top, \texttt{y} \mapsto \top, \texttt{z} \mapsto \top]\big)$. Again, we assume that the first component of the lifted store corresponds to the configuration $\{A\}$, the second to $\{B\}$, and the third to $\{A,B\}$. Fig. 16 lists some of the dataflow equations obtained for this

l	$[\![s^l]\!]_{\overline{\mathsf{out}}}$
1	$([\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3])$
3	$([\mathtt{x}\mapsto 1,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto 1,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3])$
6	$([\mathtt{x}\mapsto 1,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3])$
9	$([\mathtt{x}\mapsto\top,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3])$
10	$([\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3])$
14	$([\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3])$

Fig. 17. A solution to the lifted dataflow equations for Example 8 (some labels omitted).

program. As in Example 7, we denote by s^l the statement with label l. By using the equations in Fig. 16, after the first two lines we have:

$$[\![s^3]\!]_{\overline{\mathsf{out}}} = \big([\mathtt{x}\mapsto 1,\mathtt{y}\mapsto\top,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto\top,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3],[\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 7,\mathtt{z}\mapsto 3]\big)$$

By using lifted dataflow equations, we can also calculate input and output abstract stores for all statements in the above program. Some of the output stores are highlighted in Fig. 17. We can see that after the termination of the program, $[\![s^9]\!]_{\overline{\mathsf{out}}}$, the values of $\mathtt{y}$ and $\mathtt{z}$ are constants 7 and 3 respectively for configurations $\{B\}$ and $\{A,B\}$, whereas for configuration $\{A\}$ the value of $\mathtt{y}$ is non-constant.

The obtained dataflow equations are variability aware and provably sound:

Theorem 9 *(Soundness of lifted dataflow analysis). For all $\bar{s}\in\overline{Stm}$ such that $[\![\bar{s}^\ell]\!]_{\overline{\mathsf{in}}}$, $[\![\bar{s}^\ell]\!]_{\overline{\mathsf{out}}}$ satisfies the dataflow equations in Fig. 15:*

$$\overline{\mathcal{A}}[\![\bar{s}^\ell]\!]([\![\bar{s}^\ell]\!]_{\overline{\mathsf{in}}}) \;\ddot{\sqsubseteq}\; [\![\bar{s}^\ell]\!]_{\overline{\mathsf{out}}}$$

Proof. The proof is by structural induction on s^ℓ. We consider the case for $\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}$.

$$\begin{aligned}
&\overline{\mathcal{A}}[\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]([\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{in}}})\\
&= \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{\mathcal{A}}[\![s^{\ell_0}]\!][\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{in}}}) & k\vDash\varphi \\ \pi_k([\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{in}}}) & k\nvDash\varphi \end{cases} && \text{(by def. of } \overline{\mathcal{A}} \text{ in Fig. 11)}\\
&= \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{\mathcal{A}}[\![s^{\ell_0}]\!][\![s^{\ell_0}]\!]_{\overline{\mathsf{in}}}) & k\vDash\varphi \\ \pi_k([\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{in}}}) & k\nvDash\varphi \end{cases} && \text{(by guarded def. of } [\![s^{\ell_0}]\!]_{\overline{\mathsf{in}}})\\
&\ddot{\sqsubseteq} \prod_{k\in\mathbb{K}} \begin{cases} \pi_k([\![s^{\ell_0}]\!]_{\overline{\mathsf{out}}}) & k\vDash\varphi \\ \pi_k([\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{in}}}) & k\nvDash\varphi \end{cases} && \text{(by IH)}\\
&= \prod_{k\in\mathbb{K}} \begin{cases} \pi_k([\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{out}}}) & k\vDash\varphi \\ \pi_k([\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{out}}}) & k\nvDash\varphi \end{cases} && \text{(by def. of } [\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{out}}})\\
&= [\![\mathtt{\#if}^\ell\ \varphi\ s^{\ell_0}]\!]_{\overline{\mathsf{out}}} \quad \text{(simplify)} \qquad \square
\end{aligned}$$

6.3. Trading precision for speed with variability abstraction

So far, we have argued that it is most practical to develop analyses for single programs, and then apply our lifting combinator to lift their definition to program families via a formal calculation. This process appears most straightforward, but it has one disadvantage: all the abstractions applied in the derivation of a single-program analysis are unaware of variability. This way it is impossible to abstract over variability, which could sometimes be beneficial. For example, when the configuration space is too large, it may be difficult or impossible to represent lifted stores symbolically, so that they take little space in memory. Abstraction is the standard response of static analysis to such challenges, but for this particular problem one needs to abstract the configuration space. Variability abstractions can only be applied at the family level: one needs an analysis formulated at the family level and then apply the variability aware abstraction to it, in the very same way as we applied usual abstractions on the single program level in Section 3. In the end, we obtain a computationally cheaper but less precise analysis, since an additional over-approximation is introduced in it.

Variability-aware abstractions can be plentiful. In this section we show one example: an abstraction that ignores a certain subset of features, presumably meant to have insignificant impact on the analysis results. Let $F\subset\mathbb{F}$ be a set of features that we deem relevant for the analysis. Then if $k\in\mathbb{K}$ is a valid configuration, $k\cap F$ is a simplification of this configuration to relevant features only. Let $\mathbb{K}_F$ be the set of valid configurations over relevant features, so we have $\mathbb{K}_F=\{k\cap F\mid k\in\mathbb{K}\}$. Let $\langle\mathbb{X},\sqsubseteq\rangle$ stand for *any* complete lattice domain, which is lifted as usual, so $\overline{\mathbb{X}}=\mathbb{X}^{\mathbb{K}}$. We write $\overline{\mathbb{X}}_F$ for lifting $\mathbb{X}$ to the set of

valid configurations over only the relevant features, so $\overline{\mathbb{X}}_F = \mathbb{X}^{\mathbb{K}_F}$. Both $\langle \overline{\mathbb{X}}, \dot{\sqsubseteq} \rangle$ and $\langle \overline{\mathbb{X}}_F, \dot{\sqsubseteq} \rangle$ are complete lattices. Clearly since the latter tracks the analysis values for a smaller set of configurations, it is a more abstract domain, thereby collapsing more information. Indeed, one can formulate abstraction and concretization functions between the two lifted domains:

$$\alpha_F(\bar{x}) = \prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_k(\bar{x}) \tag{6}$$

$$\gamma_F(\bar{x}_F) = \prod_{k \in \mathbb{K}} \pi_{(k \cap F)}(\bar{x}_F) \tag{7}$$

Theorem 10. $\langle \overline{\mathbb{X}}, \dot{\sqsubseteq} \rangle \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \langle \overline{\mathbb{X}}_F, \dot{\sqsubseteq} \rangle$ *is a Galois connection.*

Proof. We first show that α_F and γ_F are monotone. Assume that $\bar{x} \dot{\sqsubseteq} \bar{x}'$ and $\bar{x}_F \dot{\sqsubseteq} \bar{x}'_F$, we have:

$$\alpha_F(\bar{x}) = \textstyle\prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_k(\bar{x}) \dot{\sqsubseteq} \prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_k(\bar{x}') = \alpha_F(\bar{x}')$$

$$\gamma_F(\bar{x}_F) = \textstyle\prod_{k \in \mathbb{K}} \pi_{(k \cap F)}(\bar{x}_F) \dot{\sqsubseteq} \prod_{k \in \mathbb{K}} \pi_{(k \cap F)}(\bar{x}'_F) = \gamma_F(\bar{x}'_F)$$

$\gamma_F \circ \alpha_F$ extensive:

$$\begin{aligned}
\gamma_F(\alpha_F(\bar{x})) &= \prod_{k \in \mathbb{K}} \pi_{(k \cap F)}(\alpha_F(\bar{x})) \quad \text{(by def. of } \gamma_F\text{)} \\
&= \prod_{k \in \mathbb{K}} \pi_{(k \cap F)}\Big(\prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k' \in \mathbb{K} \mid k_F = k' \cap F\}} \pi_{k'}(\bar{x})\Big) \quad \text{(by def. of } \alpha_F\text{)} \\
&= \prod_{k \in \mathbb{K}} \bigsqcup_{\{k' \in \mathbb{K} \mid k \cap F = k' \cap F\}} \pi_{k'}(\bar{x}) \quad \text{(by def. of } \pi_{(k \cap F)}\text{)} \\
&\dot{\sqsupseteq} \prod_{k \in \mathbb{K}} \pi_k(\bar{x}) \;=\; \bar{x} \quad \text{(since } k \cap F = k \cap F\text{)}
\end{aligned}$$

$\alpha_F \circ \gamma_F$ reductive:

$$\begin{aligned}
\alpha_F(\gamma_F(\bar{x}_F)) &= \prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_k(\gamma_F(\bar{x}_F)) \quad \text{(by def. of } \alpha_F\text{)} \\
&= \prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_k\Big(\prod_{k' \in \mathbb{K}} \pi_{(k' \cap F)}(\bar{x}_F)\Big) \quad \text{(by def. of } \gamma_F\text{)} \\
&= \prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_{(k \cap F)}(\bar{x}_F) \quad \text{(by def. of } \pi_k\text{)} \\
&= \prod_{k_F \in \mathbb{K}_F} \bigsqcup_{\{k \in \mathbb{K} \mid k_F = k \cap F\}} \pi_{k_F}(\bar{x}_F) \quad \text{(since } k_F = k \cap F\text{)} \\
&= \prod_{k_F \in \mathbb{K}_F} \pi_{k_F}(\bar{x}_F) \;=\; \bar{x}_F \quad \text{(simplify)} \qquad \square
\end{aligned}$$

This Galois connection can be composed with any family-based analysis transfer function to produce a version of the analysis that is less precise regarding the set of valid configurations (cf. Contribution (C5)). In particular, it could be composed with our constant propagation analysis $\overline{\mathcal{A}}$. In the extreme case, if we ask for an analysis that is insensitive to all features (so $F = \emptyset$), we obtain an abstracted analysis, which conservatively detects which values are constant (same) in all configurations.

Example 9. Let us reconsider Example 6, where we calculated that:

$$\begin{aligned}
&\overline{\mathcal{A}}[\![\overline{S}]\!]\big([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \top]\big) \\
&\quad = \big([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]\big)
\end{aligned}$$

for $\mathbb{K} = \{\{A\}, \{B\}, \{A, B\}\}$. We denote the final output $\mathbb{K}$-tuple as $\bar{a}_0$.

Let the set of relevant features F_1 be $\{B\}$. Then $\mathbb{K}_{F_1} = \{\emptyset, \{B\}\}$ and $\alpha_{F_1}(\bar{a}_0) = \big([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3], [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 3]\big)$. So we have $\pi_\emptyset(\alpha_{F_1}(\bar{a}_0)) = \pi_{\{A\}}(\bar{a}_0)$ and $\pi_{\{B\}}(\alpha_{F_1}(\bar{a}_0)) = \pi_{\{B\}}(\bar{a}_0) \dot{\sqcup} \pi_{\{A,B\}}(\bar{a}_0)$.

If the set of relevant features is $F_2 = \emptyset$, we have $\mathbb{K}_{F_2} = \{\emptyset\}$ and $\alpha_{F_2}(\bar{a}_0) = \pi_{\{A\}}(\bar{a}_0) \dot{\sqcup} \pi_{\{B\}}(\bar{a}_0) \dot{\sqcup} \pi_{\{A,B\}}(\bar{a}_0) = \big([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 3]\big)$, which indicates that the final value of $\mathtt{z}$ will be the constant 3 for all configurations in $\mathbb{K}$.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

In general, our process of developing an analysis, which should abstract variability, starts at a single program level (Fig. 9). We recommend developing the analysis for single programs first, and then applying lifting at a convenient intermediate step. After lifting the intermediate analysis function, one can apply a variability abstraction (for example $\langle \overline{\mathbb{X}}, \dot{\sqsubseteq} \rangle \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \langle \overline{\mathbb{X}}_F, \dot{\sqsubseteq} \rangle$ presented above) and then continue applying the liftings of the remaining abstractions (Galois connections) to develop the final analysis. In simple words: it is possible to switch the level in Fig. 9 at a convenient point, where abstracting over variability is beneficial for the design.

The presented variability-aware abstraction performs a projection of the space of all valid configurations onto some subset, and can be used to implement a *sampling strategy* [26] in static analysis. That approach selects only a subset of possible configurations, usually based on some sampling criteria, in order to reduce the verification problem and to identify results faster. Another interesting abstraction to consider would be to join all analysis results corresponding to configurations that satisfy some condition. In this way, we can find whether some fact (result) holds for all such products. For example, we can conclude whether some variable has a constant value for all products that satisfy given property.

We now extend the variational abstract interpretation framework summarized in Section 5.2 with five additional steps to include variability-aware abstractions in the derivation process:

1. Decide at which point in the design of single-program analysis the variability abstraction should be inserted. Compose the collecting semantics and all Galois connections until this point to obtain a partially specified analysis for single programs.
2. Apply the lifting combinator to the obtained analysis, and simplify the result to obtain the partially specified lifted analysis. As before, correctness follows from combining the previous calculations and Theorem 6.
3. Lift the remaining Galois connections to program families by applying our lifting combinator for Galois connections. By property of the lifting combinator, the lifted functions form composable Galois connections between lifted domains.
4. Formulate the Galois connection abstracting configurations.

Remark. You may want to use the variability abstraction specified in Eqs. (6) and (7), which is independent of $\overline{\text{IMP}}$ and the analysis domains considered here.

5. Compose the lifted Galois connections with the lifted partial analysis, in order to obtain the final formulation of the lifted analysis that includes variability abstraction. Soundness of the result follows from the soundness of the partially lifted analysis and the soundness of composing Galois connections.

6.4. Efficient lifted analyses

Taking a step back, it appears as if we have merely traded a breadth-first for a depth-first iteration strategy. Instead of performing $|\mathbb{K}|$ analyses, we now perform *one* analysis on $|\mathbb{K}|$-sized tuples—afterall: $|\mathbb{K}| \times 1 = 1 \times |\mathbb{K}|$. Note that, even so, empirical evidence suggests that the latter might be faster than the former because of *caching effects* [27]. At the lifted analysis level, however, a lot of improvements are possible.

First, the brute force approach has to compile (preprocess and build control flow graph) and execute the fixed point iterative algorithm once for each possible valid product. On the other hand, the lifted approach will compile and execute the algorithm only once per SPL. Still, in the latter case the algorithm has to do as many iterations as are needed for the slowest converging product.

Second, all configuration satisfiability tests, $k \models \varphi$, can be memorized (see the last three dataflow equations of Fig. 15 ending with: "$\underline{k \models \varphi}$"). This will enable faster analysis for '`#if`' statements whose conditions have been previously tested.

Third, observe that many of the dataflow equations (transfer functions) act identically for all (resp., some) valid configurations. Thus they can be executed efficiently by running them once (resp., several times), instead of exponentially many times. For example, consider the statement `x := n`. It can be analyzed only once, and then its result will be propagated to all configurations. In fact, only the last three equations of Fig. 15 deal with variability directly.

Fourth, it is now possible to use a *shared representation* for representing sets of configurations with equivalent analysis information. Fig. 18 shows the results of applying a constant propagation analysis to a simple program, "`if (A) x:=x+1; if (B) x:=x+1`", with the four valid configurations, $\mathbb{K} = \{\emptyset, \{A\}, \{B\}, \{A, B\}\}$. We assume that the variable `x` is initially zero. Fig. 18 illustrates how sets of configurations (equivalent with respect to the analysis) are only slowly split by the variability `#if` statements. Initially, *all* configurations, $[\![true]\!] = \{\emptyset, \{A\}, \{B\}, \{A, B\}\}$, may be shared as they all have equivalent analysis information, $[x \mapsto 0]$, associated with them. Thus, the initial lifted store $([x \mapsto 0], [x \mapsto 0], [x \mapsto 0], [x \mapsto 0])$ is represented as $([\![true]\!] \mapsto [x \mapsto 0])$. After the first variability statement, "`if (A) x:=x+1`", the four configurations get split into those for which A is disabled, $[\![\neg A]\!] = \{\emptyset, \{B\}\}$ (which are still mapped to $[x \mapsto 0]$) and those where A is enabled, $[\![A]\!] = \{\{A\}, \{A, B\}\}$ (which are mapped to $[x \mapsto 1]$; i.e., where x has been increased by one). Now, the lifted store $([x \mapsto 0], [x \mapsto 1], [x \mapsto 0], [x \mapsto 1])$ is represented as $([\![\neg A]\!] \mapsto [x \mapsto 0], [\![A]\!] \mapsto [x \mapsto 1])$. After the second variability statement, "`if (B) x:=x+1`", we have $[\![\neg A \wedge \neg B]\!] = \{\emptyset\}$ mapped to $[x \mapsto 0]$ and $[\![A \wedge B]\!] = \{\{A, B\}\}$ mapped to $[x \mapsto 2]$. Also, we have an equivalence class with two configurations: $[\![(\neg A \wedge B) \vee (A \wedge \neg B)]\!] = \{\{A\}, \{B\}\}$ mapped to $[x \mapsto 1]$; i.e., we have some sharing which comes from merging two configurations that were in distinct equivalence classes before the statement.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

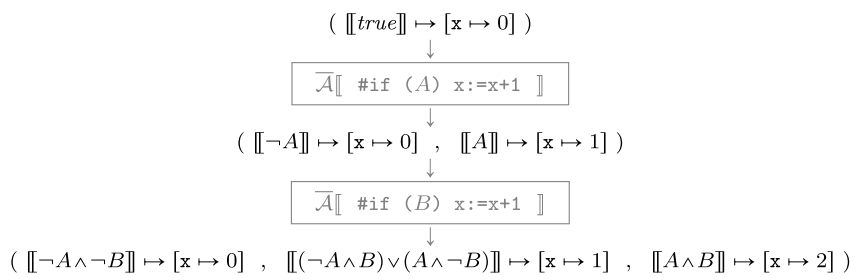

Fig. 18. The set of configurations is slowly split into equivalence classes by `#if` statements.

Bench. method	$\lvert\mathbb{K}\rvert$	Brute force	Lifted+sha.	Lifted+max.abs	Single prog
`GPL::V'.display()`	106	179.3	23.1	1.17	1.05
`GPL::G'.run()`	72	4.36	0.87	0.11	0.064
`BerkeleyDB::D'.main()`	40	197.83	83.23	5.05	4.94
`Prevayler::P'.publisher()`	8	0.35	0.18	0.065	0.051

Fig. 19. Performance comparison of the brute force vs. lifted with sharing vs. lifted with maximum abstraction ($\alpha_\emptyset$) vs. the average time of single program analysis for methods with the highest number of configurations $|\mathbb{K}|$. All times are in milliseconds (ms).

The triangular shape of Fig. 18 is a general phenomenon. Initially *all* configurations may be shared. Then, as the flow of control passes `#if` statements, the configuration space is slowly split up into more and more equivalence classes (sets of analysis-equivalent configurations). Sharing is initially optimal, and then produces diminishing results later in the program.

Fifth, sharing can be efficiently and compactly implemented either by representing sets of configurations as *bit vectors* [27] or formulae in *conjunctive/disjunctive normal form* (aka, CNF/DNF); or, even more effectively, as *binary decision diagrams* (aka, BDDs) [28]. Despite the optimistic effects of sharing, the analysis problem is inherently exponential, in the worst case. Given $|\mathbb{F}|$ *features*, there exists $|\mathbb{K}| = 2^{|\mathbb{F}|}$ distinct *configurations*, $k \subseteq \mathbb{F}$, and thereby $2^{2^{|\mathbb{F}|}}$ distinct formulae (`#if` constraints), $\varphi \subseteq \mathbb{K}$. Since each formula can be represented in logarithmic space, a formula will take up exponential space: $O(log(2^{2^{|\mathbb{F}|}})) = O(2^{|\mathbb{F}|})$, in the worst case.

Sixth, we can introduce variability abstractions as another way (along with sharing) to speed up lifted analyses. Variability abstractions, such as those presented in Section 6.3, may tame the combinatorial explosion of configurations and reduce it to something more tractable. Their aim is to replace a large configuration space with a smaller one and perform an approximate, but feasible lifted analysis on it.

6.5. Evaluation

We have evaluated the performance of different lifted analyses based on the implementation from [27]. That implementation uses SOOT's intra-procedural dataflow analysis framework [29] for analyzing Java programs. Sharing is implemented using high-performance bit vector library [30]. We have implemented a lifted analysis with the variability abstraction $\alpha_\emptyset$ ($F = \emptyset$) within the tool [27]. As we pointed out in Section 6.3, this is the maximum abstracted analysis that is insensitive to all features and operates on 1-sized tuples. In this way, it represents the fastest abstracted lifted analysis with the coarsest precision. We ran a lifted *reaching definitions* dataflow analysis on all methods of three Java SPL benchmarks: `Graph PL (GPL)`, `Prevayler`, and `BerkelyDB` [3]. Figs. 19 and 20 show a performance comparison between the brute force approach (which analyzes all configurations, one at a time), lifted analysis with sharing, lifted analysis with maximum variability abstraction $\alpha_\emptyset$, and the average time of single program analysis. The experiments are executed on a 64-bit Intel®Core™ i5 CPU running at 1.8 GHz frequency with 8 GB memory. Fig. 19 shows the analysis times to run four methods from our benchmarks with the highest number of configurations $|\mathbb{K}|$. Fig. 20 plots the average "speed up factor" with sharing and maximum abstraction as a function of the number of valid configurations in all methods of our benchmarks. We see that the brute force is fastest when we only have 1 configuration and thus nothing to share or abstract. We see that the effectiveness of sharing and maximum abstraction goes up as we get more configurations. For the method with the highest number of configurations `GPL::Vertex.display()` with $|\mathbb{K}| = 106$ configurations, we obtain that sharing is 8 times and maximum abstraction is 160 times faster than the brute force approach.

If the number of configurations $|\mathbb{K}|$ in an SPL is very large, then lifted analysis (even with sharing) may become very slow or even infeasible since the properties and transfer functions are $|\mathbb{K}|$-sized tuples. In that case, we can introduce variability abstractions into it in order to reduce the configuration space. Thus, we will obtain an approximate (less precise), but feasible (faster) lifted analysis. We can choose some appropriate variability abstraction in the spectrum from the finest one $\alpha_\mathbb{F}$ (which is identity) to the coarsest (maximum) one $\alpha_\emptyset$ (which works on 1-sized tuples). In fact, since lifted analysis with maximum abstraction $\alpha_\emptyset$ works on 1-size tuples, its analysis time is quite close to the single program analysis that runs on only one valid product from an SPL. To illustrate this, we observe in Figs. 19 and 20 the running time of lifted analysis with $\alpha_\emptyset$ and the average duration of all single program analyses that take one configuration from an SPL at a time.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

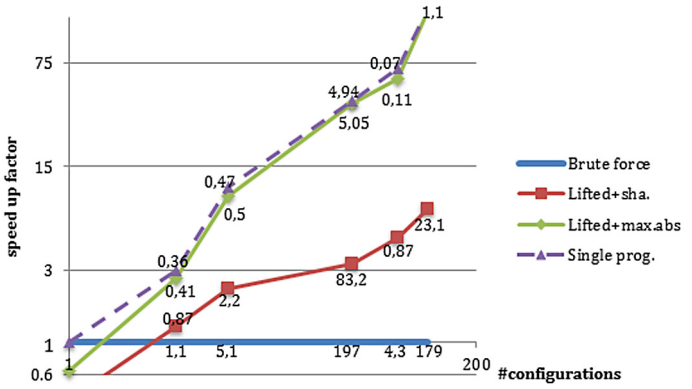

Fig. 20. The average "speed-up" effect of sharing and maximum abstraction as a function of the number of configurations N in all methods in our benchmarks. The unlifted *brute force* analysis of all N configurations (normalized with "speed factor" 1) vs. the *lifted analysis with sharing* vs. *lifted analysis with maximum abstraction* vs. the average time of *single program analysis*. For $N = 4, 8, 40, 72$, and 106, we show the actual running times in ms of one representative method with N configurations for the four considered analyses.

7. Related work

Recently, many different techniques have been proposed for analyzing software product lines (see [6] for a survey). The main distinctive feature of our approach is that we propose a generic framework that can be used for lifting different analyses (phrased as abstract interpretation). Thus, we start this section by comparing our approach with the closest related work that has the same aim to define: *lifting as a general framework*. Then we continue our comparison to other related work that lifts individual analyses, which we split into several categories: *lifting representations*, *lifting dataflow analyses*, *lifting other analyses*, and *lifting by simulation*. We end our discussion by looking at: *abstract interpretation* and *multi-staged program analysis*.

Lifting as a general framework: Some of the main approaches to program analysis as identified in [17] are: dataflow analysis, abstract interpretation, and type-based analysis. It is important to note that these are not competitive approaches, but are different techniques appropriate for different purposes, and to some extent for different languages. Type-based analyses are specified by extending the typing system of the given language in order to express the program properties of interest. Examples of such analyses are: control flow analysis, exception analysis, flow inference, overloading, etc. Very recently, Chen and Erwig have proposed [31] a generic framework for lifting any type-based analysis to program families. That approach shares the same objective as our method. Namely, we propose a generic framework for lifting any analysis specified using abstract interpretation to program families. Thus, both approaches have the abilities to reuse much of the artifacts generated during creation of single-program analysis for the new lifted analysis, which is provably correct by construction. However, abstract interpretation based framework can be applied equally well to typed and untyped programming languages, whereas the type based framework can be applied only to typed languages. Moreover, different formal techniques (such as variability abstractions) can be incorporated into our approach in order to additionally speed up lifted analyses.

Lifting representations: The preprocessor directives may be applied directly to abstract syntax trees (ASTs), not only to statements as here. In that case, some AST element, i.e., the corresponding code fragment, will be included or not in a product depending on the given configuration. For example, [32] presents an imperative language similar to our IMP with such fine-grained variability. However, variability in arbitrary language elements, even undisciplined directives, can always be converted accordingly using code duplication [22,23]. A similar coarse-grained variability as here is presented in Colored Featherweight Java [33], where variability can occur only in a restricted set of code elements, such as class declarations, fields, terms, etc.

Kästner et al. [34] show how languages with preprocessor syntax can be parsed and represented in syntax trees with variability, even if the preprocessor syntax is not properly nested in the main language syntax. Erwig and Walkingshaw [8] present the Choice Calculus, which can be seen as an elegant version of a preprocessor with a fixed and well defined semantics. It would be interesting to develop variational abstract interpretation further, to support preprocessors like choice calculus, and undisciplined preprocessor use. The former appears a rather straightforward extension, while the latter likely remains a challenge due to difficulty of defining semantics elegantly in a syntax-directed manner.

Lifting dataflow analysis: Previous work lifts dataflow analysis, resulting in *feature-sensitive* dataflow analysis [27], corresponding to our Fig. 15. Lifted dataflow analyses are much faster than ones based on $|\mathbb{K}|$ runs of the naive generate and analyze strategy [27]. Another efficient implementation of the lifted dataflow analysis formulated within the IFDS framework [35] is proposed in SPL$^{\text{LIFT}}$ [28]. It achieves several orders of magnitude speedups through the use of BDD-based sharing of configurations and encoding of lifted transfer functions and control-flow as graphs for which the fixed-point computation can be rephrased as graph reachability. In fact, it has been shown that the running time of analyzing all programs in a family is close to the analysis of a single program. However, this technique is limited to work only for analyses phrased within the IFDS framework [35], a subset of dataflow analyses with certain properties, such as distributivity of

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

transfer functions. Many dataflow analyses, including constant propagation, are not distributive and hence cannot be expressed in IFDS. Moreover, many static analyses that are not expressible as dataflow analyses, such as type checking, model checking, testing, cannot be handled by the techniques described in [27,28].

There are also lifted dataflow analyses that are not obtained from existing analyses for single programs, but are specific to SPLs. For example, [36] presents a configurability related analysis for large-scale product lines, such as the Linux kernel, which finds and removes dead source code and superfluous `#ifdef` statements.

Lifting other analyses: Recent work [6] has surveyed analysis strategies for SPLs and proposes a taxonomy of such which would classify our lifted analyses as *family-based analyses* (whereas the *generate and analyze* strategy yields a *product-based analysis*).

The approaches of *type checking*, *well-formedness checking*, and *model checking* are complementary to dataflow analysis and share the commendable goal of detecting errors at compile-time as opposed to run-time. There is work on lifting all of them in an attempt to find errors at SPL compile-time as opposed to post product-instantiation time, when a product happens to be compiled, possibly long after it has been developed: *lifted type checking* [37,38], *lifted well-formedness checking* [39], and *lifted model checking* [40,41].

Safe composition [37,38,42] is about verification and safe generation of properties for SPL assets and aims to provide guarantees that only products where certain properties are obeyed can be generated. Errors detected include type and definition-usage errors (e.g., undeclared variables, undeclared fields, and unimplemented abstract methods). We complement this with an approach based on abstract interpretation with which analyses intercepting those kinds of errors can be derived.

Lifting by simulation: Variability encoding [43] or configuration lifting [44] are based on generating a product-line *simulator* which simulates the behavior of all products in the product line. Compile-time variability (`#if` statements) is encoded using normal control flow mechanisms, such as `if` statements, and available features are encoded using non-deterministically initialized global (feature) variables. Then, an existing off-the-shelf model checker is used to verify the generated product-line simulator, so that an erroneous product can be reconstructed/decoded from an erroneous execution path. Similar to our work, the simulation-based approaches adopt a family-based analysis strategy, working on the level of program families. They are able to reuse existing off-the-shelf tools that work on the level of single programs, but there is a loss of precision during the lifting phase. On the other hand, in our approach we do not lose any precision in the lifting phase (see Theorem 6). It would be interesting to consider implementing a dataflow analysis by using the simulation technique, i.e., by converting compile-time into run-time variability. On the other hand, we could also use techniques from abstract interpretation and dataflow analysis to extract a finite-state model from program, which can be used for verifying temporal properties by model checking.

Abstract interpretation: Abstract interpretation is a general theory that unifies *dataflow analysis* [9], *model checking* [11, 12], *type systems* [13], and *testing* [45]. Our analyses have been developed using the classical Galois connection framework [9]. In particular, we follow the calculational and compositional approach advocated by Cousot [15]. With this approach, soundness follows from a systematic derivation. Indeed, this is the case for the dataflow analysis derived in Fig. 15. This approach has previously been used by the first author to derive, for instance, iterative graph algorithms [46] and modular control-flow analyses [47].

Multi-staged program analysis: Our work is related to *multi-staged program analysis*, analyzing "programs that generate programs", e.g., [48]. However, we are in a much simpler case where the first preprocessing stage considered here is significantly more restrictive than a Turing-complete programming language and can thus be dealt with without approximation. For SPLs, our approach is simpler and sufficient; and without loss of precision on the variability level.

8. Conclusion

We have shown how compositional and systematic derivation of static analyses based on abstract interpretation can be lifted to Software Product Lines. The result is variational abstract interpretation—a compositional and systematic approach for the derivation of variability-aware product line analyses, with the following distinctive components and properties:

- A scheme to lift domain types, and combinators for lifting analyses and Galois connections.
- A general soundness-by-construction result (Theorem 6), allowing to lift a formally developed analysis, without re-proving the entire abstract interpretation process. This crucially reuses all the effort invested in developing a single-program analysis, to obtain a provably sound family-based analysis.
- A possibility of incorporating abstractions that involve configuration space; including an example of one such abstraction.
- Precise control over precision of analyses (lifting does not loose any information *per se*).
- A scheme to obtain dataflow equations for family-based analyses from the abstract interpretation definition.

Variational abstract interpretation mixes language-independent and language-specific elements. The main language specific theorem (Theorem 5) needs to be proved for each new analysis. We have proved it for all the three semantics of our language and extracted a general proof methodology presented in this paper. On the other hand, the main language-independent soundness theorem (Theorem 6) holds in general and needs not be reproved.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

Abstract interpretation is a unifying theory that allows the derivation of dataflow analyses, control flow analyses, model checking, type systems, verification, and even testing. Hence, variational abstract interpretation tells us how to systematically obtain lifted versions of all such analyses. We believe that in this sense, variational abstract interpretation, contributes to the understanding of how variability affects analysis of programs in general.

Finally, since the lifting operator can be applied to a directly formulated analysis, we claim that the obtained insight into lifting extends beyond abstract interpretation. In particular, the *lift* combinator can be applied to any single-program analyses developed in an ad hoc process, without abstract interpretation, but represented as transfer functions (soundness of such lifting requires a separate argument though).

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison–Wesley, 2001.
[2] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison–Wesley, 2000.
[3] C. Kästner, Virtual separation of concerns: toward preprocessors 2.0, Ph.D. thesis, University of Magdeburg, Germany, May 2010.
[4] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, J. Sincero, Configuration coverage in the analysis of large-scale system software, Oper. Syst. Rev. 45 (3) (2011) 10–14.
[5] F. Medeiros, M. Ribeiro, R. Gheyi, Investigating preprocessor-based syntax errors, in: Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA – October 27–28, 2013, 2013, pp. 75–84.
[6] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (1) (2014) 6.
[7] IBM, Thales, F. FOKUS, TCS, Proposal for Common Variability Language (CVL) Revised Submission, 2012.
[8] M. Erwig, E. Walkingshaw, The choice calculus: a representation for software variation, ACM Trans. Softw. Eng. Methodol. 21 (1) (2011) 6:1–6:27.
[9] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: POPL'79, 1979, pp. 269–282.
[10] J. Midtgaard, T. Jensen, A calculational approach to control-flow analysis by abstract interpretation, in: SAS'08, in: Lecture Notes in Computer Science, vol. 5079, Springer-Verlag, Valencia, Spain, 2008, pp. 347–362.
[11] P. Cousot, R. Cousot, Refining model checking by abstract interpretation, Autom. Softw. Eng. 6 (1) (1999) 69–95.
[12] P. Cousot, R. Cousot, Temporal abstract interpretation, in: POPL'00, 2000, pp. 12–25.
[13] P. Cousot, Types as abstract interpretations, in: POPL'97, 1997, pp. 316–331.
[14] J. Midtgaard, C. Brabrand, A. Wasowski, Systematic derivation of static analyses for software product lines, in: 13th International Conference on Modularity, MODULARITY '14, 2014, 2014, pp. 181–192.
[15] P. Cousot, The calculational design of a generic abstract interpreter, in: M. Broy, R. Steinbrüggen (Eds.), Calculational System Design, in: NATO ASI Series F, IOS Press, Amsterdam, 1999.
[16] G. Winskel, The Formal Semantics of Programming Languages, Foundation of Computing Series, The MIT Press, 1993.
[17] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, Secaucus, USA, 1999.
[18] M. Sagiv, N. Rinetzky, Class notes on program analysis course, Tech. rep., Computer Science Department, Tel Aviv University, March 2007.
[19] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-Oriented Domain Analysis (FODA) feasibility study, Tech. rep., Carnegie–Mellon University Software Engineering Institute, November 1990.
[20] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, Cool features and tough decisions: a comparison of variability modeling approaches, in: VaMoS'12, 2012, pp. 173–182.
[21] D. Batory, Feature models, grammars, and propositional formulas, in: 9th International Software Product Lines Conference, in: LNCS, vol. 3714, Springer-Verlag, 2005, pp. 7–20.
[22] A. Garrido, R.E. Johnson, Analyzing multiple configurations of a C program, in: ICSM'05, IEEE Computer Society, 2005, pp. 379–388.
[23] A. Garrido, R.E. Johnson, Refactoring C with conditional compilation, in: ASE'03, IEEE Computer Society, 2003, pp. 323–326.
[24] J. Midtgaard, C. Brabrand, A. Wasowski, Systematic derivation of static analyses for software product lines, Tech. rep. TR-2014-170, IT University of Copenhagen, 2014.
[25] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Log. Program. 13 (2–3) (1992) 103–179.
[26] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, D. Beyer, Strategies for product-line verification: case studies and experiments, in: 35th International Conference on Software Engineering, ICSE '13, 2013, pp. 482–491.
[27] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, P. Borba, Intraprocedural dataflow analysis for software product lines, in: Transactions on Aspect-Oriented Software Development, vol. 10, 2013, pp. 73–108.
[28] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, M. Mezini, Spl^{lift}: statically analyzing software product lines in minutes instead of years, in: ACM SIGPLAN Conference on PLDI '13, 2013, pp. 355–364.
[29] R. Vallée-Rai, P. Co, E. Gagnon, L.J. Hendren, P. Lam, V. Sundaresan, Soot—a Java bytecode optimization framework, in: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, 1999, p. 13.
[30] The colt project: open source libraries for high performance scientific and technical computing in java, CERN: European Organization for Nuclear Research, http://acs.lbl.gov/software/colt/.
[31] S. Chen, M. Erwig, Type-based parametric analysis of program families, in: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, 2014, pp. 39–51.
[32] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, K. Ostermann, Toward variability-aware testing, in: FOSD '12, 2012, pp. 1–8.
[33] C. Kästner, S. Apel, T. Thüm, G. Saake, Type checking annotation-based product lines, ACM Trans. Softw. Eng. Methodol. 21 (3) (2012) 14.
[34] C. Kästner, P.G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, T. Berger, Variability-aware parsing in the presence of lexical macros and conditional compilation, in: OOPSLA'11, ACM, Portland, OR, USA, 2011, pp. 805–824.
[35] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: Proc. 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '95, 1995, pp. 49–61.
[36] R. Tartler, D. Lohmann, J. Sincero, W. Schröder-Preikschat, Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem, in: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 47–60.
[37] S. Apel, C. Kästner, A. Grösslinger, C. Lengauer, Type safety for feature-oriented product lines, Autom. Softw. Eng. 17 (2010) 251–300.
[38] C. Kästner, S. Apel, Type-checking software product lines—a formal approach, in: ASE'08, L'Aquila, Italy, 2008, pp. 258–267.
[39] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness OCL constraints, in: GPCE'06, ACM, New York, NY, USA, 2006, pp. 211–220.
[40] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, Symbolic model checking of software product lines, in: ICSE'11, 2011, pp. 321–330.
[41] A. Gruler, M. Leucker, K.D. Scheidemann, Modeling and model checking software product lines, in: FMOODS'08, 2008, pp. 113–131.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005

[42] S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe composition of product lines, in: GPCE'07, ACM, New York, NY, USA, 2007, pp. 95–104.
[43] S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer, Detection of feature interactions using feature-aware verification, in: ASE'11, IEEE Computer Society, Lawrence, USA, 2011.
[44] H. Post, C. Sinz, Configuration lifting: verification meets software configuration, in: ASE'08, IEEE Computer Society, L'Aquila, Italy, 2008, pp. 347–350.
[45] D. Guilbaud, E. Goubault, A. Pacalet, B.S.F. Védrine, A simple abstract interpreter for threat detection and test case generation, in: WAPATV'01, with ICSE'01, Toronto, 2001.
[46] I. Sergey, J. Midtgaard, D. Clarke, Calculating graph algorithms for dominance and shortest path, in: MPC'12, in: LNCS, vol. 7342, Springer, Madrid, Spain, 2012, pp. 132–156.
[47] J. Midtgaard, M.D. Adams, M. Might, A structural soundness proof for Shivers's escape technique: a case for Galois connections, in: SAS'12, in: LNCS, vol. 7460, Springer-Verlag, 2011, pp. 352–369.
[48] W. Choi, B. Aktemur, K. Yi, M. Tatsuta, Static analysis of multi-staged programs via unstaging translation, SIGPLAN Not. 46 (1) (2011) 81–92.

Please cite this article in press as: J. Midtgaard et al., Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.04.005