A Categorical Basis for Robust Program Analysis

ZACHARY KINCAID, Princeton University, USA
SHAOWEI ZHU, Princeton University, USA

Users of program analyses expect that results change predictably in response to changes in their programs, but many analyses do not ensure such robustness. This paper introduces a theoretical framework that provides a unified language to articulate robustness properties. We adopt a categorical view in which programs and their properties form a category, and robust analyses are characterized as structure-preserving functors. A diverse range of robustness properties—e.g., invariance under variable renaming and monotonicity—arise from instantiating the category's arrows accordingly.

Beyond formulating the meaning of robustness, this paper provides methods for achieving it. The first is a general recipe for designing robust analyses, by lifting a sound and robust analysis from a restricted (sub-Turing) model of computation to a sound and robust analysis for general programs. This recipe demystifies the design of several existing loop summarization and termination analyses by showing they are instantiations of this general recipe, and furthermore elucidates their robustness properties. The second is a characterization of a sense in which an algebraic program analysis is robust, provided that it is comprised of robust operators. In particular, we show that such analyses behave predictably under common refactoring patterns, such as variable renaming and loop unrolling.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Program analysis, Program transformation, Robustness, Category theory

ACM Reference Format:
Zachary Kincaid and Shaowei Zhu. 2026. A Categorical Basis for Robust Program Analysis. *Proc. ACM Program. Lang.* 10, PLDI, Article 229 (June 2026), 24 pages. https://doi.org/10.1145/3808307

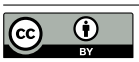
1 Introduction

The goal of program analysis is to predict how a program's state evolves over time. However, programs are not static artifacts—they too change. This paper is motivated by the question of how we can predict how a *program analysis* will behave in response to such changes.

Abstractly, we might think of a program analysis as a function mapping a program P to some approximation $A(P)$ of P's behavior (for instance, $A(P)$ might be a mapping from the loops of P to associated invariants, or a yes/no decision about whether P terminates). Suppose that a given program P is transformed in some particular way to yield a program P'. What can be said about the relationship between $A(P)$ and $A(P')$? Reasonable expectations might be:

- *(Invariance under renaming)*: If P and P' are equivalent modulo renaming of variables, $A(P)$ and $A(P')$ are also equivalent modulo renaming.
- *(Locality)*: $A(P)$ and $A(P')$ differ only in the places that P and P' do (for instance, one might expect that changing one function does not affect the information computed for another, unrelated function).

Authors' Contact Information: Zachary Kincaid, Princeton University, Princeton, NJ, USA, zkincaid@cs.princeton.edu; Shaowei Zhu, Princeton University, Princeton, NJ, USA, shaoweiz@cs.princeton.edu.



ACM 2475-1421/2026/6-ART229
https://doi.org/10.1145/3808307

- *(Monotonicity)*: If P' is a refinement of P (e.g., P' is obtained from P by inserting assumed invariants) then $A(P')$ should be at least as precise as $A(P)$.

Unfortunately, many program analysis tools do not provide guarantees on "robustness", which can make them difficult to use and pose challenges in their deployments in production environments. For instance, an experience report on the commercial static analyzer Coverity notes that users expect consistent results when their code changes, even across tool versions, and discusses how Coverity eschews techniques like SMT solving, randomization, and use of time-outs to avoid unstable bug reports [Bessey et al. 2010]. Several reports lament failure of locality, or *butterfly effects*—"a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program" [Leino and Pit-Claudel 2016]. As Leino and Moskal [2010] put it succinctly: "tools can understand our programs, but we cannot understand our tools."

The main contributions of this paper are:

- (Section 4) We develop a theoretical framework for understanding robustness. The high-level idea is that a program analysis A should be considered to be robust with respect to a class of relationships R (e.g., equivalence modulo renaming variables) exactly when, should some relationship $r \in R$ hold between two programs P and P', then r also holds between $A(P)$ and $A(P')$. We formalize this intuitive notion using the language of category theory: we think of both the space of programs and analysis results as forming categories (in which the arrows represent the class of relationships of interest), and define an analysis to be robust exactly when it preserves this categorical structure.
- (Section 5) We present a general recipe for designing robust program analyses. The idea is to leverage weak (that is, not Turing-complete) models of computation that have tractable dynamics. To analyze a program we (1) extract a *model* of the program of some restricted form, (2) analyze the behavior of the model, and then (3) translate the model analysis to an *approximate* analysis for the original program. We show that, supposing certain properties of the model extraction and result translation process, this recipe preserves robustness properties of the model analysis. Furthermore, we identify instances of this recipe in the literature, and thereby show that they are robust.
- (Section 6) We investigate robustness in the context of algebraic program analysis, driven by the question of how robustness guarantees of the individual operators of an algebra translate to high-level guarantees of the whole analysis. In particular, we show that program analyses following the recipe of Section 5 are robust with respect to *loop-preserving stuttering simulations*. This class of relationships accounts for both non-structural transformations (e.g., renaming a variable) and structural transformations (e.g., unrolling a loop).

Proofs of all results are provided in the extended version of this paper [Kincaid and Zhu 2026].

1.1 Related Work

Program analysis frameworks. Abstract interpretation provides a uniform framework for *sound* program analysis [Cousot and Cousot 1977]. It defines what it means for a program analysis to be sound in terms of an approximation relationship between a "concrete" semantics defining the program behavior and an "abstract" semantics defining the program analysis. It also provides a general recipe for designing sound program analyses—the job of the analysis designer is to develop a lattice of program properties, transfer functions that "lift" program commands to act on these properties, and a widening operator to extrapolate the limit of a sequence of properties.

The goal of our paper is to do for *robustness* what abstract interpretation does for *soundness*. While partial orders are sufficient to relate possible invariants for a single program ($X \sqsubseteq Y$ means that Y

Table 1. Correspondence between order-theoretic terminology in abstract interpretation and category-theoretic terminology in this paper.

Abstract interpretation	Robust program analysis
Lattice	Concrete category
Monotone function	Concrete functor
Galois connection	Weak adjoint

approximates X), we require richer structure to relate possible invariants for different programs. In our framework, the "abstract domain" forms a category. The arrows of a category can be thought of as a generalization of the approximation order in abstract interpretation: an arrow $f : X \to Y$ means that Y approximates X *modulo the transformation* f. For instance, if f is a transformation that replaces a variable x with a and y with b, we might write $f : (0 \leq x \leq y \leq 10) \to (0 \leq a \wedge b \leq 100)$, meaning that the property on the left of the arrow under the transformation f (that is, $0 \leq a \leq b \leq 10$) is stronger than the property on the right. In this sense, one can think of an order-theoretic abstract domain as a category in which the only transformations of interest are identity transformations. Table 1 provides a correspondence between some category-theoretic terminology used in this paper and their order-theoretic counterparts to provide intuition for how our work connects with classical abstract interpretation.

Prior categorical treatments of abstract interpretation [Katsumata et al. 2023; Steffen et al. 1992] use categories in a different way than ours: they use enriched categories in which objects are types, arrows are programs, and homsets carry a poset structure; analyses are lax functors, encoding the idea that the analysis of a composite program is more precise than the separate analysis of its components. In contrast, we use categories in which objects are programs and arrows are *relationships between programs*. Analyses are functors from programs to results, encoding the idea that robust analyses preserve relationships between programs.

Algebraic program analysis is a framework for compositional program analysis (that is, satisfying the *locality* principle) [Kincaid et al. 2021]. Kincaid [2018] presents a strategy for designing algebraic program analyses that are also *monotone*, by "lifting" a reflexive transitive closure operator for a sub-Turing model of computation to an over-approximate loop summarization routine for general loops. Monotonicity is a special case of robustness, in which the relationship between programs one wishes to preserve is refinement. There are several analyses that follow the spirit of this strategy, but formally are not instances because they "lift" an operation other than reflexive transitive closure (see Table 3). Section 5 presents a generalization of this design strategy, such that all examples in Table 3 are formally instances of this strategy; monotonicity and soundness of these analyses are thus corollaries of Theorems 1 and 2. Our strategy is parametric in the notion of robustness that it preserves (allowing us to conclude that the aforementioned analyses satisfy stronger properties than monotonicity), and furthermore is shown to preserve certain algebraic laws (Section 5.4).

Program transformation and analysis. Control-flow refinement is a family of techniques that aim to improve analysis precision by performing loop transformations (e.g., unrolling a loop, or splitting a loop into two or more phases). While these techniques have been shown to improve analysis precision experimentally, there are also cases where these transformations lead to results that are incomparable or even less precise. An exception is [Cyphert et al. 2019], which gives a transformation that is guaranteed to improve precision, subject to certain conditions of the analysis. In this paper, we give a recipe for designing analyses that meet these conditions.

Amato et al. [2010] and Ranzato and Zanella [2018] observe that numerical program analyses are often not robust under invertible linear transformations—i.e., if one applies a linear change of variables f to a program P (substituting each variable x with a linear combination of fresh variables)

to obtain an isomorphic program $f[P]$, the computed invariants are generally not isomorphic. Amato et al. [2010] take advantage of this to enhance analysis precision, by using a statistical analysis of execution data to identify beneficial change-of-basis transformation. Monniaux and Le Guen [2012] make a similar observation about non-robustness with respect to projecting out variables, and propose an analogous technique to exploit it. In this paper, we show that some analyses that have previously been shown to be *monotone* are in fact *robust under linear transformations* (a generalization of monotonicity that includes not only logical entailment of the results but also invariance under renaming of variables). As a result, these robust analyses do not benefit from the aforementioned techniques, but the analysis designer need not be concerned with *which* linear transformation or projection to use in the first place.

Logozzo and Fähndrich [2008] observe that translating a program from a high-level language to a low-level language (e.g., bytecode) may negatively impact program analysis precision, and introduced the notion of *relative completeness* to describe the situation in which precision loss does not occur. Lesbre and Lemerre [2024] develop a strategy for combining non-relational analyses with online SSA conversion to achieve analyses that are relatively complete in this sense. Our work provides a general framework in which to understand program analyses preserving relationships between programs; relative completeness is an example of a robustness property in which the relationship is a transformation that introduces new variables to represent intermediate computations.

2 Overview

Consider the two programs P_1 and P_2 pictured in Figure 1 (derived from [Monniaux and Le Guen 2012]). P_1 can be obtained from P_2 by (1) renaming the x variable to be i, and (2) projecting out the y variable. Thus, one might expect that a loop invariant generation algorithm would produce an invariant for P_2 that is at least as precise as P_1 (modulo renaming x to be i). However, Monniaux and Le Guen [2012] observed that polyhedral analysis violates this expectation—the invariant computed for P_1 is $1 \leq i \leq 5$, while the invariant for P_2 is $x \leq 5$. In the vocabulary of this paper, this example illustrates that polyhedral analysis is not robust with respect to linear transformations. The essential reason is that polyhedral analysis uses a widening operator to extrapolate the limit of a sequence of successively weaker candidate invariants—the extrapolation step is essentially an educated guess, and is sensitive to the particular way that a polyhedron is represented.

Not only is robustness a desirable property of an analysis—it is also *achievable*, even by analyses capable of inferring a rich class of invariants. We now consider an alternative method to analyze this loop, which makes use of polyhedra in a different way, and which *is* robust with respect to linear transformations. This method relies on a *convex hull* primitive: for any linear integer/rational arithmetic formula F, $conv(F)$ denotes the closed convex hull of F; i.e., a formula of the form $\bigwedge_{i=0}^{n} t_i \geq b_i$ where each t_i is a linear term (over the free variables of F) and b_i is a rational constant, such that F entails $conv(F)$ and $conv(F)$ entails any linear inequality that is entailed by F.

Instead of approximating the reachable states of a machine, our goal is to summarize its dynamics. This summary can be represented by a *transition formula*—a logical formula over a set of program variables and their "primed" copies, representing the program state before and after a computation. For instance, the instruction x++ in the program P_2 can be represented by the transition formula $x' = x + 1 \wedge y' = y$. *Algebraic program analysis* computes such summaries in "bottom-up" fashion: it begins by summarizing individual statements, then builds up summaries for larger and larger fragments of the program until it obtains a summary for the whole program. A central problem of interest in algebraic program analysis is the design of *iteration operators*, which given a summary for the body of a loop, computes a summary that represents any number of iterations of that loop body formula.

```
i = 1;
while(i < 5) {
  i++;
}
```

(a) P_1

```
x = 1; y = 0;
while(x < 5) {
  x++;
  y += x;
}
```

(b) P_2

Fig. 1. Two programs P_1 and P_2, where P_1 approximates P_2 modulo renaming the x variable to be i.

Suppose that F is a transition formula over the variables X and X'; our objective is to compute a formula $F^\star$ that over-approximates the reflexive transitive closure of F. We consider two particular ways to use the domain of convex polyhedra to define an iteration operator, inspired by [Ancourt et al. 2010; Farzan and Kincaid 2015]. Descriptions of these iteration operators appear below; Table 2 illustrates the corresponding computations for the programs P_1 and P_2.

- *(Polyhedral guard analysis)* Define $Pre(F) \triangleq conv(\exists X'.F)$ and $Post(F) \triangleq conv(\exists X.F)$, which represent the pre-condition and post-condition of the transition formula F, respectively. As long as at least one iteration of F is taken, both $Pre(F)$ and $Post(F)$ must hold, and so
$$F^{\star_{\text{PGA}}} \triangleq X' = X \vee (Pre(F) \wedge Post(F))$$
over-approximates the reflexive transitive closure of F.
- *(Linear recurrence analysis)* A *linear recurrence* is a linear inequality of the form $t' \leq t + b$ where t is a linear term over the pre-state variables X, t' is t under the substitution that replaces each unprimed variable with its primed counter-part, and b is a constant. If a loop body summary F entails a linear recurrence $t' \leq t + b$, then t may increase by at most b in one iteration of the loop (and thus may increase by at most kb after k iterations of the loop). We can compute linear recurrences from a loop body formula F as follows. First, we introduce a set of variables $\Delta_X \triangleq \{\delta_x : x \in X\}$, where each δ_x represents the change in x, and define
$$\Delta(F) \triangleq \exists X, X'.F \wedge \bigwedge_{x \in X} \delta_x = x' - x\ .$$
Then $conv(\Delta(F))$ takes the form $\bigwedge_{i=1}^{n} t_i \leq b_i$, where each t_i is a linear term over Δ_X. Since $\Delta(F)$ entails each $t_i \leq b_i$, we have that F entails $t_i[\Delta_X \mapsto X'] \leq t_i[\Delta_X \mapsto X] + b_i$ (where $t[\Delta_X \mapsto X']$ denotes the parallel substitution that replaces each δ_x with x' in the linear term t)—i.e., each constraint in the constraint representation of the convex hull of $\Delta(F)$ corresponds to a linear recurrence inequality entailed by F. Thus,
$$F^{\star_{LRA}} \triangleq \exists k \in \mathbb{N}.k \geq 0 \wedge \bigwedge_{i=1}^{n} t_i[\Delta_X \mapsto X'] \leq t_i[\Delta_X \mapsto X] + kb_i$$
over-approximates the reflexive transitive closure of F.

Naturally, these two ideas can be combined into one iteration operator, which we will refer to as the *polyhedral iteration operator*:

$$F^{\circledast} \triangleq \exists k \in \mathbb{N}. \left(\begin{array}{c} \bigwedge_{i=1}^{n} t_i[\Delta_X \mapsto X'] \leq t_i[\Delta_X \mapsto X] + kb_i \\ \wedge((k = 0 \wedge X' = X) \vee (k \geq 1 \wedge Pre(F) \wedge Post(F))) \end{array} \right)$$

The key point is that the relationship between P_1 and P_2 also holds between the summary for the loops in P_1 and P_2 (see Table 2): if we take the loop summary $F_2^{\circledast}$ for P_2, and rename x and x' to i and i', we obtain a formula that entails the loop summary $F_1^{\circledast}$ for P_1. In fact, this holds for any pair

Table 2. Loop summary computation for the programs P_1 and P_2 from Figure 1. Observe that the same linear transformation that relates P_1 and P_2 also relates the summary for P_1 and P_2.

	P_1	P_2
Loop body F	$F_1 \triangleq i < 5 \wedge i' = i + 1$	$F_2 \triangleq x < 5 \wedge x' = x + 1 \wedge y' = y + x + 1$
$\Delta(F)$	$\exists i, i'.\, F_1 \wedge \delta_i = i' - i$	$\exists x, x', y, y'.\, F_2 \wedge \delta_x = x' - x \wedge \delta_y = y' - y$
$conv(\Delta(F))$	$-\delta_i \leq -1 \wedge \delta_i \leq 1$	$-\delta_x \leq -1 \wedge \delta_x \leq 1 \wedge \delta_y \leq 5$
Recurrences	$i' = i + 1$	$x' = x + 1 \wedge y' \leq y + 5$
Precondition	$i < 5$	$x < 5$
Postcondition	$i' \leq 5$	$x' \leq 5$
Loop summary $F^{\circledast}$	$\exists k.\ \begin{array}{l} k \geq 0 \wedge i' = i + k \\ \wedge (k \geq 1 \Rightarrow (i < 5 \wedge i' \leq 5)) \end{array}$	$\exists k.\ \begin{array}{l} k \geq 0 \wedge x' = x + k \\ \wedge y' \leq y + 5k \\ \wedge (k = 0 \Rightarrow y' = y) \\ \wedge (k \geq 1 \Rightarrow (x < 5 \wedge x' \leq 5)) \end{array}$

of programs that are related by a linear transformation. Thus, we say that this analysis is *robust with respect to linear simulations*.

In view of this example, the contributions of this paper can be understood as follows. Section 4 presents a framework for understanding analyses that preserve relationships between programs, of which robustness with respect to linear simulations is an example. Robustness of $(-)^{\circledast}$ operator with respect to linear simulations is a consequence of the fact that it is an instance of the design strategy of Section 5. Section 6 shows that robustness of the $(-)^{\circledast}$ operator extends to the entire analysis based upon it—the relationship between P_1 and P_2 can be seen as a loop-preserving stuttering simulation, and thus the summary computed for the whole program P_2 entails that of P_1 (modulo renaming).

3 Background and Notation

3.1 Transition Systems and Transition Formulas

A **transition system** T consists of a set of states S_T and a transition relation $\rightarrow_T \subseteq S_T \times S_T$. We use the notation $x \rightarrow_T x'$ to denote that the pair $\langle x, x' \rangle$ belongs to the transition relation of T. Let T and U be transition systems. A **simulation from** T **to** U is a function $s : S_T \rightarrow S_U$ such that for all $x, x' \in S_T$ such that $x \rightarrow_T x'$, we have $s(x) \rightarrow_U s(x')$.

For transition systems T and U over the same state space, define their sequential composition

$$T \cdot U \triangleq \{\langle s, s'' \rangle : \exists s' \in S_T . s \rightarrow_T s' \wedge s' \rightarrow_U s''\}.$$

We use T^n to denote the n-fold composition of T with itself, and T^* to denote the reflexive transitive closure of T. We use T^ω to denote the non-terminating states of T; i.e., the set of all states $s \in S_T$ such that there exists an infinite sequence $s_0, s_1, \cdots \in S_T$ such that $s_0 = s$ and $s_i \rightarrow_T s_{i+1}$ for all $i \in \mathbb{N}$.

Let X be a set of variable symbols. A **transition formula** over X is a first-order formula whose free variables range over X and a set of "primed copies" $X' \triangleq \{x' : x \in X\}$; these variables designate the state of a program before and after a transition, respectively. For the sake of concreteness, we suppose that transition formulas are expressed in the language of linear integer/rational arithmetic. We let $\mathbf{TF}(X)$ denote the set of all transition formulas over X. Similarly, $\mathbf{SF}(X)$ denotes the state formulas over X—formulas whose free variables range over X.

A transition formula $F \in \mathbf{TF}(X)$ defines a transition system where $S_F = \mathbb{Q}^X$ (i.e., its states are maps from X to $\mathbb{Q}$) and $s \rightarrow_F s'$ iff the formula F is satisfied by a model in which the variables in X are interpreted according to s and the variables in X' are interpreted according to s'. Note that transition formulas are closed under sequential composition, in the sense that for any transition

formulas F and G, the transition system $F \cdot G$ is represented by the transition formula $\exists X''.F[X' \mapsto X''] \wedge G[X \mapsto X'']$. However, it is not generally the case that F^* or F^ω are representable by a formula.

If $F \in \mathbf{TF}(X)$ and $G \in \mathbf{TF}(Y)$ are transition formulas, a **linear simulation** from F to G is a linear map $f : \mathbb{Q}^X \to \mathbb{Q}^Y$ that is also a simulation (if $s \to_F s'$, then $f(s) \to_G f(s')$). Since formulas are *constraints* on transitions, it is convenient to think of a linear simulation in *transposed form*—that is, a substitution σ that maps the variables in Y to linear terms over the variables in X. Here the substitution σ can be understood as the linear map that sends $s \in \mathbb{Q}^X$ to $\lambda y. [\![\sigma(y)]\!](s)$, where $[\![t]\!](s)$ denotes the evaluation of the term t over X in the state s. We also introduce a convenient notation for representing simulations using term substitutions as follows. For any substitution σ mapping a set of variables Y to terms over a set of variables X, we let σ' denote "primed" variant of σ, such that $y' \in Y'$ is mapped to $\sigma(y)[X \mapsto X']$. Then we have that σ is a simulation from F to G iff $F \models G[\sigma, \sigma']$, where $G[\sigma, \sigma']$ denotes the result of applying both substitutions σ and σ' to G. That is, we may think of $G[\sigma, \sigma']$ as the *weakest* among the transition formulas H such that σ is a simulation from H to G.

Example 3.1. Consider the programs P_1 and P_2 from Figure 1. The loop body of P_1 is the transition formula $F_1 \triangleq i < 5 \wedge i' = i+1$ over $\{i\}$, and the loop body of P_2 is $F_2 \triangleq x < 5 \wedge x' = x+1 \wedge y' = y+x+1$ over $\{x, y\}$. The projection that maps $(x, y) \mapsto x$, which corresponds to the substitution $\sigma = [i \mapsto x]$, is a linear simulation from F_2 to F_1 (one may verify that $F_2 \models F_1[\sigma, \sigma'] \equiv (x < 5 \wedge x' = x + 1)$). Additional examples of transition formulas and linear simulations appear in Figure 3. ⌟

3.2 Algebraic Program Analysis

An algebraic program analysis [Kincaid et al. 2021] can be thought of as a two-step process: (1) compute a regular expression recognizing a set of paths of interest through a program, and then (2) calculate a summary that over-approximates this set of paths by interpreting that regular expression within an appropriate algebraic structure (that is, re-interpret each regular expression operator as a corresponding operation on summaries).

In the context of this paper, we assume summaries form Pre-Kleene algebra. A **Pre-Kleene algebra**[1] (PKA) is an algebraic structure $\langle A, 0, 1, +, \cdot, (-)^\star \rangle$ equipped with two distinguished elements 0 and 1, two binary operations + and ·, and a unary operation $(-)^\star$ that satisfies the laws of idempotent semirings as well as the following laws governing the iteration operator:

- *(Reflexivity)* $1 \leq a^\star$
- *(Extensivity)* $a \leq a^\star$
- *(Transitivity)* $a^\star \cdot a^\star = a^\star$
- *(Monotonicity)* if $a \leq b$ then $a^\star \leq b^\star$
- *(Unrolling)* $(a^n)^\star \leq a^\star$ for all $n \in \mathbb{N}$

where $\leq$ is the order defined by the + operator ($a \leq b$ iff $a + b = b$). An example of a PKA is the algebra $\mathbf{TF}(X)$ in which the universe is the set of transition formulas over X, $0^{\mathbf{TF}}$ is *false*, 1 is the identity relation $\bigwedge_{x \in X} x' = x$, + is disjunction, · is relational composition ·, and $(-)^\star$ is the over-approximate transitive closure operator $(-)^{\circledast}$.

Let A be a PKA. A **flow graph** over A consists of a finite set of vertices V_G, a finite set of directed edges $E_G \subseteq V_G \times V_G$, a distinguished root vertex $r_G \in G$, and a weight function $w_G : E \to A$ assigning each edge a non-zero weight. We extend w_G to a function $E^* \to A$ as $w_G(e_1 e_2 \ldots e_n) \triangleq w_G(e_1) \cdot w_G(e_2) \cdot \cdots \cdot w_G(e_n)$. In this paper, we use flow graphs over $\mathbf{TF}(X)$ as a model for programs.

[1]This definition of Pre-Kleene algebras first appeared in [Cyphert et al. 2019], which observes that the laws of Kleene algebra are often not satisfied by algebraic program analyses. For instance, the algebra of transition formulas cannot be extended with a $\star$ operator to form a Kleene algebra, owing to the fact that least fixed points are not first-order definable.

3.3 Category Theory

We recall some of the basic definitions of category theory. A **category** $\mathbf{C}$ consists of

- A collection of objects $Ob(\mathbf{C})$
- For each pair of objects X, Y, a collection $\mathbf{C}(X, Y)$ of arrows
- For each triple of objects X, Y, Z, a composition function $\circ : \mathbf{C}(Y, Z) \times \mathbf{C}(X, Y) \to \mathbf{C}(X, Z)$
- For each object X, an arrow $1_X \in \mathbf{C}(X, X)$

such that composition is associative, and for each X, Y and each $f \in \mathbf{C}(X, Y)$ we have $f \circ 1_X = f = 1_Y \circ f$. Examples of categories include **Set**, in which the objects are sets and arrows are functions; $\mathbf{Vect}_{\mathbb{Q}}$, in which the objects are rational vector spaces and arrows are linear transformations; and **TS**, in which the objects are transition systems and arrows are simulations.

Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A **functor** is a map F sending each object X of $\mathbf{C}$ to an object $F(X)$ of $\mathbf{D}$ and each arrow $f \in \mathbf{C}(X, Y)$ to an arrow $F(f) \in \mathbf{D}(F(X), F(Y))$ such that:

- For any $f \in \mathbf{C}(Y, Z)$ and $g \in \mathbf{C}(X, Y)$, we have $F(f \circ g) = F(f) \circ F(g)$
- For each object X of $\mathbf{C}$, we have $F(1_X) = 1_{F(X)}$

A functor F is **faithful** if it is injective on arrows; i.e., for any objects X and Y of $\mathbf{C}$ and any arrows $f, g \in \mathbf{C}(X, Y)$, if $F(f) = F(g)$, then $f = g$.

Suppose that F and G are functors from $\mathbf{C}$ to $\mathbf{D}$. A **natural transformation** $\eta : F \Rightarrow G$ associates each object $X \in Ob(\mathbf{C})$ with an arrow $\eta_X \in \mathbf{D}(F(X), G(X))$ such that for any $f \in \mathbf{C}(X, Y)$, we have $\eta_Y \circ F(f) = G(f) \circ \eta_X$.

A **subcategory** $\mathbf{D}$ of a category $\mathbf{C}$ is obtained by selecting a sub-collection of objects and arrows that includes all identity arrows of the selected objects and is closed under composition. There is an obvious faithful inclusion functor from $\mathbf{D}$ to $\mathbf{C}$ that maps each object and arrow in $\mathbf{D}$ to themselves.

4 A Category-Theoretic Foundation of Robust and Sound Analysis

Informally, we say that a program analysis $A : \mathit{Program} \to \mathit{Property}$ is robust with respect to a class of relationships R if for all programs P and P', if P and P' are related by some $r \in R$, then $A(P)$ and $A(P')$ are also related by r. We formalize this by considering *Program* and *Property* to be *concrete categories* over a common base category which defines the class of relationships that we are interested in preserving, and defining an operation to be robust when it is a *concrete functor*.

Let $\mathbf{B}$ be a category (the "base" category). A **concrete category over $\mathbf{B}$** consists of a category $\mathbf{C}$ along with a faithful functor $\phi_{\mathbf{C}} : \mathbf{C} \to \mathbf{B}$ (see [Adámek et al. 2009] for an introduction to concrete category theory). For any objects $A, B \in Ob(\mathbf{C})$ and any arrow $f \in \mathbf{B}(\phi_{\mathbf{C}}(A), \phi_{\mathbf{C}}(B))$, we use $A \overset{f}{\Mapsto} B$ to denote that there exists some arrow $g \in \mathbf{C}(A, B)$ such that $\phi_{\mathbf{C}}(g) = f$ (note that the choice of g is unique, since $\phi_{\mathbf{C}}$ is faithful). Thus, the functor $\phi_{\mathbf{C}}$ allows us to understand the arrows of $\mathbf{C}$ as being drawn from $\mathbf{B}$, in the sense that for any objects $A, B \in Ob(\mathbf{C})$, $\mathbf{B}(\phi_{\mathbf{C}}(A), \phi_{\mathbf{C}}(B))$ defines a space of "possible" relationships that may hold between A and B, and $A \overset{r}{\Mapsto} B$ is the proposition that the relationship r does hold between A and B. The fact that $\mathbf{C}$ is a category means that these relationships compose:

- *(Identity)*: for any object A we have $A \overset{1_{\phi_{\mathbf{C}}(A)}}{\Mapsto} A$. (Since $\phi_{\mathbf{C}}$ is a functor, we must have $\phi_{\mathbf{C}}(1_A) = 1_{\phi_{\mathbf{C}}(A)}$)
- *(Composition)*: if $A \overset{f}{\Mapsto} B \overset{g}{\Mapsto} C$, then $A \overset{g \circ f}{\Mapsto} C$. (Letting $f' \in \mathbf{C}(A, B)$ and $g' \in \mathbf{C}(B, C)$ be such that $\phi_{\mathbf{C}}(f') = f$ and $\phi_{\mathbf{C}}(g') = g$, we have $g' \circ f' \in \mathbf{C}(A, C)$ and $\phi_{\mathbf{C}}(g' \circ f') = \phi_{\mathbf{C}}(g') \circ \phi_{\mathbf{C}}(f') = g \circ f$).

Example 4.1. The category of transition systems **TS** can be understood as a concrete category over **Set**, where $\phi_{\mathbf{TS}}$ sends any transition system T to its state space S_T, and sends any simulation $f : T \to U$ to itself (essentially "forgetting" that f is a simulation, and just looking at it as a function

from S_T to S_U). For a function $f : S_T \to S_U$, we write $T \overset{f}{\Vdash\!\!\to} U$ exactly when f is a simulation; i.e., $T \overset{f}{\Vdash\!\!\to} U$ is a proposition meaning that for all s and s' such that $s \to_T s'$, we have $f(s) \to_U f(s')$.

Observe that the transitive closure operation is robust in the sense that the state spaces of T and T^* coincide, and if there is a simulation $f : T \to U$ between two transition systems, f is also a simulation $T^* \to U^*$. The $(-)^\omega$ operation is similarly robust. Define the category **SubSet** where objects are pairs $\langle U, S\rangle$ where U is a set and S is a subset of U, and where an arrow $f : \langle U, S\rangle \to \langle U', S'\rangle$ is a function $U \to U'$ such that for all $x \in S$, we have $f(x) \in S'$. Observe that **SubSet** is a concrete category over **Set**, where $\phi_{\mathbf{SubSet}}$ maps each object $\langle U, S\rangle$ to U, and each arrow $f : \langle U, S\rangle \to \langle U', S'\rangle$ to itself. Then $(-)^\omega : \mathbf{TS} \to \mathbf{SubSet}$ satisfies the property that if $T \overset{f}{\Vdash\!\!\to} U$, then $T^\omega \overset{f}{\Vdash\!\!\to} U^\omega$. ⌟

The above example motivates the formal definition of robustness:

Definition 1 (Robustness)**.** Let **B** be a category, and let **C** and **D** be concrete categories over **B**. A function $F : Ob(\mathbf{C}) \to Ob(\mathbf{D})$ is **robust** (with respect to **B**, or arrows in **B**) if for every object $X \in Ob(\mathbf{C})$ we have $\phi_{\mathbf{C}}(X) = \phi_{\mathbf{D}}(F(X))$ and for every pair of objects $X, Y \in \mathbf{C}$ and every arrow $f \in \mathbf{B}(\phi_{\mathbf{C}}(X), \phi_{\mathbf{C}}(Y))$ such that $X \overset{f}{\Vdash\!\!\to} Y$, we have $F(X) \overset{f}{\Vdash\!\!\to} F(Y)$.

Robust functions coincide with an existing notion in category theory called *concrete functors*. For concrete categories **C** and **D** over some common base category **B**, a functor $F : \mathbf{C} \to \mathbf{D}$ is **concrete** if $\phi_{\mathbf{C}} = \phi_{\mathbf{D}} \circ F$.

Lemma 1 (Robust operations are concrete functors)**.** Let **B** be a category, and let **C** and **D** be concrete categories over **B**. Suppose that $F : Ob(\mathbf{C}) \to Ob(\mathbf{D})$ is robust. For any $f \in \mathbf{C}(X, Y)$, define $F(f)$ to be the arrow $g \in \mathbf{D}(F(X), F(Y))$ such that $\phi_{\mathbf{C}}(f) = \phi_{\mathbf{D}}(g)$—the choice of g is unique since $\phi_{\mathbf{D}}$ is faithful by assumption. Then F is a concrete functor.

Example 4.1 illustrates that semantic operations like reflexive transitive closure are robust; however, they are typically not computable. Our goal in program analysis is to develop approximate analyses that are computable, while retaining *some* robustness properties.

Example 4.2 (Robustness under linear transformations)**.** Suppose that we are interested in an over-approximating transitive closure operation for transition formulas that is robust with respect to linear transformations. We endow the set of transition formulas with the structure of a concrete category over $\mathbf{Vect}_{\mathbb{Q}}$ as follows. Let **TF** be a category where the objects are transition formulas over any set of variables (note that we use **TF** for the category and $\mathbf{TF}(X)$ for the set of transition formulas over a specific variable set X). If $F \in \mathbf{TF}(X)$ and $G \in \mathbf{TF}(Y)$ are transition formulas over X and Y respectively, then the set of arrows between F and G is the set of linear maps $\mathbb{Q}^X \to \mathbb{Q}^Y$ that are simulations between the transition systems defined by F and G (i.e., $x \to_F x'$ implies $f(x) \to_G f(x')$). Observe that **TF** forms a concrete category over $\mathbf{Vect}_{\mathbb{Q}}$, where the concrete functor $\phi_{\mathbf{TF}}$ sends each transition formula $F \in \mathbf{TF}(X)$ to its corresponding state space $\mathbb{Q}^X$, and each simulation $f : F \to G$ to itself. **TF** can be understood as a subcategory of **TS**, which contains only transition systems that are representable using transition formulas, and only linear simulations as arrows. As we will see in Section 5, the iteration operator $(-)^{\circledast}$ (Section 2) is a concrete functor. ⌟

Example 4.3 (Invariance under renaming)**.** This example formalizes the intuitive idea that constant propagation is insensitive to the names of variables. Formally, define a category of flow graphs $\mathbf{FG}_\alpha$: objects are flow graphs weighted with transition formulas, and an arrow from G (over variables X) to a structurally isomorphic H (over Y) is a bijection $\sigma : X \to Y$ such that $w_G(e)[\sigma] \equiv w_H(e)$ for each edge e. Given a flow graph G with weights in $\mathbf{TF}(X)$, the goal of constant propagation is to compute a function $k \in Const(V_G, X) \triangleq V_G \to (\{\bot\} \cup (X \to (\mathbb{Z} \cup \{\top\})))$, which assigns to each

```
if (y > 0) {
  y = 0;
  x = y + 1;
} else { x = 1; }
Post: {x ↦ 1, y ↦ ⊤}
```

(a) A program P

```
if (a-b > 0) {
  a = (a+b)/2; b = a;
  a = a-b+1/2; b = 1/2;
} else { a = (a-b+1)/2; b = 1-a; }
Post: {a ↦ ⊤, b ↦ ⊤}
```

(b) A program P'

Fig. 2. An example illustrating that constant propagation is not robust under linear transformations: The programs P and P' are isomorphic under the linear substitution $[x \mapsto a+b, y \mapsto a-b]$; however the result of constant propagation analysis on P is not isomorphic to that of P'.

control location $v \in V_G$ either the value $\bot$ (indicating that v is statically unreachable) or a *constant environment* that assigns each variable either an integer (the variable must always be equal to that value at v) or $\top$ (the variable may take on more than one value). Let **Bij** denote the category whose objects are finite sets (of variables) and whose arrows are bijections between them. Let **Const** be the category whose objects are constant assignments (belonging to $Const(V, X)$ for some set of vertices V and set of variables X), and where an arrow from $k \in Const(V, X)$ to $k' \in Const(V, Y)$ is a bijection $\sigma : X \to Y$ such that $k(v)(x) = k'(v)(\sigma(x))$ for all $v \in V$ and $x \in X$. Both $\mathbf{FG}_\alpha$ and **Const** are concrete categories over **Bij**, and constant propagation is a concrete functor from $\mathbf{FG}_\alpha$ to **Const**. If we are interested in robustness with respect to linear transformations rather than variable renaming, then we may analogously consider a base category of $\mathbf{Vect}_\mathbb{Q}$. However, constant propagation is *not* robust with respect to linear transformations. For instance, the programs P and P' in Figure 2 are isomorphic under the linear map $(x, y) \mapsto (a+b, a-b)$; constant propagation correctly determines $x = 1$ in P, but cannot recover the corresponding fact $a + b = 1$ in P'.[2] ⌟

Example 4.4 (Monotonicity)**.** Interval analysis computes an interval bounding the possible values of each variable at each control location. Let **VarSet** be the discrete category whose objects are finite sets of variables and whose only arrows are identities. Define a category $\mathbf{FG}_{mon}$ whose objects are flow graphs weighted with transition formulas; for flow graphs G and H with the same vertex set and the same edge set, there is a unique arrow $G \to H$ iff, for every edge e, $w_G(e)$ and $w_H(e)$ are over the same variable set and $w_G(e) \models w_H(e)$. Similarly to Example 4.3, define a category **Intv** of interval assignments: an object is a function I that maps each vertex v of some set V and each variable x of some set X to an interval $I(v)(x)$, and an arrow from I (over vertices V and variables X) to J (over the same V and X) exists iff $I(v)(x) \subseteq J(v)(x)$ for all $v \in V$ and $x \in X$—i.e., I is at least as precise as J. Both $\mathbf{FG}_{mon}$ and **Intv** are concrete over **VarSet**, where the projection sends each object to its underlying variable set. Robustness with respect to **VarSet** then requires that whenever G's formulas are stronger than H's, the analysis of G is at least as precise as that of H—it is *monotonicity*. The iterative interval analysis [Cousot and Cousot 1977] is *not* robust in this sense since it uses widening, which is inherently non-monotone [Cousot 2015]; on the other hand, policy iteration [Costan et al. 2005] (which computes strongest interval invariants) *is* robust. ⌟

Using the Framework. We envision two modes for using this framework. In the first mode, an analysis designer has a class of transformations of interest, and uses the framework to inform the design of a robust analysis (perhaps facilitated by the recipe of Section 5). We expect that for the most part, analysis designers will use a small number of choices for the base category **B**: e.g., $\mathbf{Vect}_\mathbb{Q}$ captures robustness under linear transformations (Example 4.2), **Bij** captures invariance

[2] Affine relations analysis, on the other hand, *is* robust; one might think of affine relations analysis as the minimal generalization of constant propagation that is robust with respect to linear transformations.

under variable renaming (Example 4.3), and the discrete category **VarSet** captures monotonicity (Example 4.4). In the second mode, an analysis designer has an analysis of interest, and uses the framework to characterize the class of transformations under which it is robust. For instance, the analyses in Table 3 were designed with monotonicity in mind, but in the next section we show that in fact they are robust with respect to the richer class of linear simulations.

4.1 Soundness of Robust Analysis

The main novelty of this section is to define what it means for a program analysis to be robust. To understand what it means to be a program analysis, we must also define the meaning of *soundness*, and it is convenient to do so in the same category-theoretic vocabulary that we use for robustness. This section demonstrates how any concrete category can be equipped with an "approximation pre-order" to formalize soundness.

Example 4.2 is suggestive of a general mechanism by which we arrive at the structure of a concrete category. While we might think of an algebraic analysis as a function mapping a program into the algebra of transition formulas, there is in fact a family of such algebras, parameterized by the set of variables over which they are defined.[3] We obtain **TF** as the disjoint union of transition formulas over all parameters, and the functor $\phi_{\mathbf{TF}}$ sends each transition formula back to its parameter. Thus, we can recover the algebra $\mathbf{TF}(X)$ as the *fiber* of $\mathbb{Q}^X$, $\phi_{\mathbf{TF}}^{-1}(\mathbb{Q}^X) \triangleq \left\{F : \phi_{\mathbf{TF}}(F) = \mathbb{Q}^X\right\} = \mathbf{TF}(X)$. We recover the entailment relation on transition formulas as $F \models G \iff$ there is some arrow f in **TF** between F and G such that $\phi_{\mathbf{TF}}(f) = 1_{\phi_{\mathbf{TF}}(X)}$.

More generally, for any concrete category $\mathbf{C}$ over $\mathbf{B}$, the fibers of $\mathbf{C}$ can be equipped with an "approximation pre-order." That is, for any object $B \in Ob(\mathbf{B})$, and any $C, C' \in \phi_{\mathbf{C}}^{-1}(B)$ write $C \preceq_B C'$ if and only if $C \Vdash^{1_{\phi_{\mathbf{C}}(C)}} C'$ (i.e., there is an arrow $f \in \mathbf{C}(C, C')$ such that $\phi_{\mathbf{C}}(f) = 1_{\phi_{\mathbf{C}}(C)}$). We use $C \preceq C'$ to denote that $\phi_{\mathbf{C}}(C) = \phi_{\mathbf{C}}(C')$ and $C \preceq_{\phi_{\mathbf{C}}(C)} C'$. Observe that $\preceq$ is a pre-order (where reflexivity and transitivity follow from the *Identity* and *Composition* properties of the $\Vdash^{1_{\phi_{\mathbf{C}}(C)}}$ relation, respectively).

The basic principle espoused by the theory of abstract interpretation [Cousot and Cousot 1977] is that soundness is a connection between a "concrete semantics" (the fundamental matter of interest, which is typically not computable) and an "abstract semantics" (a computable *approximation* of the concrete semantics). In category-theoretic vocabulary, we may think of a concrete semantics as an object of some concrete category $\mathbf{R}^\natural$, and an abstract semantics as an object of another concrete category $\mathbf{R}^\sharp$. A concrete functor $\gamma : \mathbf{R}^\sharp \to \mathbf{R}^\natural$ (the *concretization* functor) maps each abstract object to its concrete meaning (e.g., mapping a polyhedron to the set of states it contains); being a concrete functor means that this mapping is compatible with arrows. We say that an abstract semantics $a \in Ob(\mathbf{R}^\sharp)$ approximates a concrete semantics $c \in Ob(\mathbf{R}^\natural)$ if $c \preceq \gamma(a)$ (where $\preceq$ is the approximation order on $\mathbf{R}^\natural$). Finally, if we let $\mathbf{C}$ be a category of programs, $F^\natural : \mathbf{C} \to \mathbf{R}^\natural$ be a functor interpreting programs as their concrete semantics and $F^\sharp : \mathbf{C} \to \mathbf{R}^\sharp$ be a functor interpreting programs as their abstract semantics, then $F^\sharp$ is sound with respect to $F^\natural$ when $F^\sharp(P)$ approximates $F^\natural(P)$ for all programs P. Formally,

Definition 2 (Soundness)**.** Let $\mathbf{C}$, $\mathbf{R}^\sharp$, and $\mathbf{R}^\natural$ be concrete categories over a common base category $\mathbf{B}$, and let $\gamma : \mathbf{R}^\sharp \to \mathbf{R}^\natural$, $F^\sharp : \mathbf{C} \to \mathbf{R}^\sharp$, and $F^\natural : \mathbf{C} \to \mathbf{R}^\natural$ be concrete functors. We say that $F^\sharp$ is **sound** with respect to $F^\natural$ iff for all $P \in Ob(\mathbf{C})$, we have $F^\natural(P) \preceq \gamma(F^\sharp(P))$.

[3]The same phenomenon appears in iterative abstract interpretation. For example, although we speak of "the" lattice of polyhedra, it is really a family of lattices, one for each dimension.

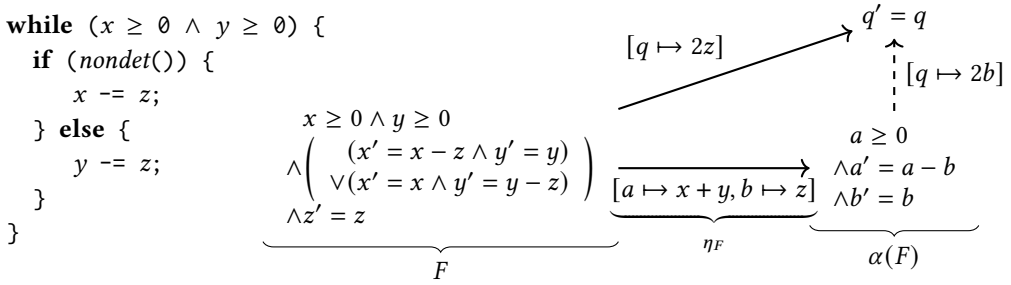


Fig. 3. Over-approximation of a loop by a deterministic affine transition system.

Again, this notion of soundness coincides with an existing notion in category theory, called *concrete natural transformations*. For concrete categories $\mathbf{C}$ and $\mathbf{D}$ and concrete functors $F, G : \mathbf{C} \to \mathbf{D}$, a natural transformation $\epsilon : F \Rightarrow G$ is called concrete if for each object A of $\mathbf{C}$, $\phi_{\mathbf{D}}(\epsilon_A) = 1_{\phi_{\mathbf{D}}(F(A))}$.

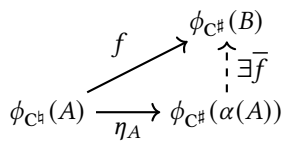
Lemma 2. $F^\sharp$ is sound w.r.t. $F^\natural$ iff there is a concrete natural transformation $\epsilon : F^\natural \Rightarrow \gamma \circ F^\sharp$.

As we will see in Section 5.4, concrete natural transformations are also a convenient mechanism with which to express algebraic laws.

4.2 Best Abstractions

Another key element of abstract interpretation is the notion of "best abstractions," which are formalized using Galois connections. Intuitively, the "best" abstraction is the universal object among all abstractions, such that all other abstractions "factor through" it. In this section, we formulate a notion of best abstractions for the purpose of program analysis in the language of concrete categories using *weak left adjoints* [Kainen 1971]. We give a definition that is specialized towards our setting below, and provide a proof of equivalence to the standard definition in [Kincaid and Zhu 2026].

Let $\mathbf{C}^\natural$ and $\mathbf{C}^\sharp$ be concrete categories over some common base category $\mathbf{B}$, and let $\gamma : \mathbf{C}^\sharp \to \mathbf{C}^\natural$ be a concrete functor (one may think of $\mathbf{C}^\sharp$ as an abstraction of $\mathbf{C}^\natural$, and γ as mapping each abstraction back to a representative in $\mathbf{C}^\natural$). A **weak left adjoint** of γ is a pair $\langle \alpha, \eta \rangle$ where α associates every object $A \in Ob(\mathbf{C}^\natural)$ with an object $\alpha(A) \in Ob(\mathbf{C}^\sharp)$, and η associates every object $A \in Ob(\mathbf{C}^\natural)$ with an arrow $\eta_A \in \mathbf{B}(\phi_{\mathbf{C}^\natural}(A), \phi_{\mathbf{C}^\sharp}(\alpha(A)))$ such that the following properties hold. First, we must have $A \overset{\eta_A}{\Longmapsto} \gamma(\alpha(A))$—i.e., $\gamma(\alpha(A))$ approximates A modulo the transformation η_A. Second, $\alpha(A)$ is the closest approximation to A within $\mathbf{C}^\sharp$, in the sense of the following universal property: for any object $B \in \mathbf{C}^\sharp$ that approximates A modulo any transformation f, there is a way to factor the transformation f as $\overline{f} \circ \eta_A$ such that B approximates $\alpha(A)$ modulo $\overline{f}$. More precisely:



for any object $B \in \mathbf{C}^\sharp$ and any arrow $f \in \mathbf{B}(\phi_{\mathbf{C}^\natural}(A), \phi_{\mathbf{C}^\sharp}(B))$ such that $A \overset{f}{\Longmapsto} \gamma(B)$, there is an arrow $\overline{f}$ such that $\alpha(A) \overset{\overline{f}}{\Longmapsto} B$ and the diagram to the left commutes. Note that α is not necessarily a functor, and $\overline{f}$ is not necessarily unique (if both hold, we recover the usual definition of an adjunction).

Example 4.5. Zhu and Kincaid [2021] present a termination analysis which is based on finding best abstractions of loops as deterministic affine transition systems (DATS). Define a DATS over a set of variables X to be a transition formula of the form $G \wedge T \in \mathbf{TF}(X)$ where G is a guard (expressed over the pre-state variables X) and T represents an affine transformation $T = \bigwedge_{x \in X} x' = t_x + b_x$ (where t_x is a linear term over X and $b_x \in \mathbb{Q}$). Let **DATS** denote the category of deterministic affine transition systems where arrows are linear simulations. **DATS** is a subcategory of **TF**, and so has a

natural inclusion functor $\gamma : \mathbf{DATS} \to \mathbf{TF}$, and γ has a weak left adjoint $\langle \alpha, \eta \rangle$. Figure 3 depicts an example of a transition formula F along with its best abstraction, consisting of a DATS $\alpha(F)$ and a linear simulation η_F. Observe that the DATS $\alpha(F)$ does not over-approximate F (in the sense that F does not entail $\alpha(F)$)—rather, $\alpha(F)$ over-approximates F *modulo the transformation* η_F (i.e., F entails $\alpha(F)[\eta_F, \eta'_F]$). Zhu and Kincaid [2021]'s termination analysis proves that the loop is terminating for all states satisfying the precondition $z > 0$ via asymptotic analysis of $\alpha(F)$.

The figure also depicts a second abstraction of F, consisting of a DATS $q' = q$ and simulation $[q \mapsto 2z]$. Intuitively, $(\alpha(F), \eta_F)$ is a closer approximation to F than this second abstraction, and this fact is witnessed by the simulation $[q \mapsto 2b]$ from $\alpha(F)$ to $q' = q$. One might thus think of weak left adjoints as Galois connections *modulo transformations*. ⌟

5 A Lifting Strategy for Robust Analyses

This section presents a general strategy for obtaining robust analyses. The high-level idea is that if we have two concrete categories **C** and **D**, and a concrete functor $\gamma : \mathbf{D} \to \mathbf{C}$ with a weak left adjoint, then a robust operation that is defined on **D** can be "lifted" to a robust operation that is defined on **C** (under certain technical assumptions). For instance, one might think of **C** as a category of programs and **D** as a subcategory of programs of a restricted form, for which a (robust) operation F is computable (for instance, reachability is decidable for programs in **D**). Our recipe "lifts" F to an *over-approximate* operation $\overline{F}$ that is defined for all programs (rather than just those in **D**). The analysis designer is responsible for the creative work of defining **D** and F and showing that the functor $\gamma : \mathbf{D} \to \mathbf{C}$ has a (computable) weak left adjoint, but needn't prove soundness or robustness (since they follow from Theorems 2 and 1).

We set the stage by illustrating examples of our strategy (Section 5.1). We then describe the general strategy (Section 5.2), formulate a soundness theorem (Section 5.3), and show that the lifting preserves algebraic laws (Section 5.4).

5.1 Examples of Lifted Analyses

We begin by showing how the analyses introduced in Section 2 arise as instances of a common "lifting" pattern. In the following, **TF** refers to the category in which the objects are transition formulas expressed in linear arithmetic and arrows are linear simulations (Example 4.2). We think of **TF** as the category that we wish to lift an operator to; the category that we lift from, and the operation we wish to lift, will vary.

Polyhedral Guard Analysis. Define **PolyCart** (polyhedral cartesian transition formulas) to be the subcategory of **TF** consisting of formulas of the form $(\bigwedge_{i=1}^{n} s_i \geq a_i) \wedge (\bigwedge_{j=1}^{m} t_j \geq b_j)$ where each s_i is a linear term over unprimed variables and each t_j is a linear term over primed variables. Observe that every polyhedral cartesian transition formula is transitively closed, so we can define a reflexive transitive closure operator $(-)^\star : \mathbf{PolyCart} \to \mathbf{TF}$ as $F^\star \triangleq X' = X \vee F$ for any F (over the variables X). Let $\gamma : \mathbf{PolyCart} \to \mathbf{TF}$ be the inclusion functor. Observe that $\langle \alpha, \eta \rangle$ is a weak left adjoint to γ (see [Kincaid and Zhu 2026, Corollary 4]), where $\alpha(G) \triangleq \mathit{Pre}(G) \wedge \mathit{Post}(G)$, and η_G is the identity. Polyhedral guard analysis can be understood as the operator $G^{\star_{\mathrm{PGA}}} = \alpha(G)^\star$, or equivalently $G^{\star_{\mathrm{PGA}}} = (\alpha(G)^\star)[\eta_G, \eta'_G]$ (noting that since η_G is the identity, the substitution $[\eta_G, \eta'_G]$ has no effect).

Linear Recurrence Analysis. Define $\mathbf{LT}_\bot$ (lossy translations) to be the subcategory of **TF** whose objects are formulas of the form $\bigwedge_{x \in X} x' \leq x + a_x$ or *false*. Computing the reflexive transitive closure of a formula in $\mathbf{LT}_\bot$ is straightforward, and gives rise to a concrete functor $(-)^\star : \mathbf{LT}_\bot \to \mathbf{TF}$,

where for a formula $F \in \mathbf{LT}_\perp$ over variables X, $F^\star$ is defined as:

$$\mathit{false}^\star \triangleq \bigwedge_{x \in X} x' = x \qquad \left(\bigwedge_{x \in X} x' \leq x + a_x\right)^\star \triangleq \exists k \in \mathbb{N}.k \geq 0 \wedge \bigwedge_{x \in X} x' \leq x + ka_x$$

Let $\gamma : \mathbf{LT}_\perp \to \mathbf{TF}$ be the inclusion functor. Observe that $\langle \alpha, \eta \rangle$ is a weak left adjoint of γ (see [Kincaid and Zhu 2026, Corollary 5]), where for any transition formula G with $\mathit{conv}(\Delta(G)) = \bigwedge_{i=1}^n t_i \leq b_i$, $\alpha(G)$ is $\bigwedge_{i=1}^n y_i' \leq y_i + b_i$ (over fresh variables $y_1, \ldots, y_n$) and η_G is the linear map corresponding to the substitution $[y_i \mapsto \hat{t}_i]_{i=1}^n$ where $\hat{t}_i \triangleq t_i[\Delta_X \mapsto X]$. Intuitively, $\alpha(G)$ introduces a fresh variable for each recurrence relation, and the η_G represents the correspondence between these fresh variables and the linear terms they represent. Linear recurrence analysis can be understood as the operator $G^{\star_{\mathrm{LRA}}} = (\alpha(G)^\star)[\eta_G, \eta_G']$. (Note the right-hand side of this equation is syntactically identical to that of $(-)^{\star_{\mathrm{PGA}}}$ above, but α, η, and $(-)^\star$ are defined with respect to the category $\mathbf{LT}_\perp$ instead of **PolyCart**).

Example 5.1. Table 2 illustrates the steps of the above analyses. Consider the loop body F_2 of P_2. For *LRA*, we compute $\Delta(F_2)$ and its convex hull, which gives $-\delta_x \leq -1 \wedge \delta_x \leq 1 \wedge \delta_y \leq 5$. Each inequality $t_i \leq b_i$ yields a fresh variable y_i, giving $\alpha(F_2) = (y_1' \leq y_1 - 1 \wedge y_2' \leq y_2 + 1 \wedge y_3' \leq y_3 + 5)$ with $\eta = [y_1 \mapsto -x,\ y_2 \mapsto x,\ y_3 \mapsto y]$. Applying the star and substituting via $[\eta_{F_2}, \eta'_{F_2}]$ yields the formula $\exists k \geq 0.\, x' = x + k \wedge y' \leq y + 5k$. Observe that the linear simulation $\sigma = [i \mapsto x]$ from F_2 to F_1 (Example 3.1) is preserved: σ is also a simulation from $F_2^{\star_{\mathrm{LRA}}}$ to $F_1^{\star_{\mathrm{LRA}}}$ (a consequence of Theorem 1 in the following). ⌟

Polyhedral Iteration Operator. The polyhedral iteration operator defined in Section 2 combines polyhedral guard analysis and linear recurrence analysis, resulting in an analysis that produces more precise invariants than both combined. This can be understood as an iteration operator induced by the product category $\mathbf{PolyCart} \times \mathbf{LT}_\perp$. More generally, lifted operations can be combined: if two subcategories $\mathbf{D}$ and $\mathbf{E}$ each admit best abstractions for $\mathbf{C}$, then so does the product $\mathbf{D} \times \mathbf{E}$ (see [Kincaid and Zhu 2026, Appendix A.1]), yielding a more precise analysis that inherits all robustness and soundness properties. For example, products of domains like $\mathbb{Q}$-VASR $\times \mathbf{LT}_\perp$ (both from Table 3) can yield new robust analyses for computing reflexive transitive closure.

Termination analysis via DATS Abstraction. Recall that a deterministic affine transition system over a set of variables X is a transition formula F of the form $G \wedge T \in \mathbf{TF}(X)$ where $G \in \mathbf{SF}(X)$ is a guard and T represents an affine transformation $T = \bigwedge_{x \in X} x' = t_x + b_x$. Zhu and Kincaid [2021] give a termination analysis for DATS, which analyzes the asymptotic behavior of the affine transformation T to compute a state formula F^ω representing a set of states from which F must eventually terminate. To lift this analysis to apply to all transition formulas, we define $F^{\overline{\omega}} \triangleq (\alpha(F))^\omega[\eta_F]$, where $\langle \alpha, \eta \rangle$ is a weak left adjoint to the inclusion functor $\gamma : \mathbf{DATS} \to \mathbf{TF}$ (Example 4.5). For instance, in Figure 3 the precondition for termination of the DATS $\alpha(F)$ is $\alpha(F)^\omega = b > 0$; applying the substitution η_F yields $z > 0$—a sufficient precondition for termination of the transition formula F.

5.2 Lifted Operations and Robustness

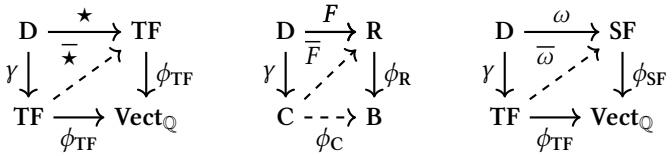
Table 3 lists more examples of "lifting" an analysis from a sub-Turing model of computation. This section formalizes this pattern, providing a unified understanding of these analyses and an account of why they are sound and robust.

Suppose that $\mathbf{D}$ is a subcategory of the category of transition formulas $\mathbf{TF}$ (Example 4.2), such that the inclusion functor $\gamma : \mathbf{D} \to \mathbf{TF}$ has a weak left adjoint $\langle \alpha, \eta \rangle$. The previous section gave examples in which a loop summarization analysis was lifted from an operator $(-)^\star : \mathbf{D} \to \mathbf{TF}$ by defining

Table 3. Summary of robust analyses that can be understood as "lifted" from a sub-Turing model. As a consequence of Theorem 1, all listed analyses are robust with respect to linear simulations.

Source	Model	"Lifted" Analysis
	Lossy translation	Reflexive transitive closure
	Cartesian relations	Reflexive transitive closure
[Zhu and Kincaid 2021]	Deterministic affine transition systems	(Eventual) non-termination analysis
[Silverman and Kincaid 2019]	Rational Vector Addition System with Resets (ℚ-VASR)	Reflexive transitive closure
[Pimpalkhare and Kincaid 2024]	Labeled ℚ-VASR	Context-free reachability
[Cyphert and Kincaid 2024]	Solvable transition ideals	Polynomial invariant generation

$F^{\overline{\star}} \triangleq (\alpha(F)^\star)[\eta_F, \eta'_F]$, and termination analysis was lifted from an operator $(-)^\omega : \mathbf{D} \to \mathbf{SF}$ by defining $F^{\overline{\omega}} \triangleq (\alpha(F)^\omega)[\eta_F]$. Naturally, this pattern can be generalized to arbitrary operations and concrete categories. The picture below depicts the general pattern of "lifted" operations (center) along with these two particular cases of loop summarization (left) and termination (right).



For an object A of $\mathbf{C}$, we may think of $\alpha(A)$ as approximating A *modulo* η, and so applying F to $\alpha(A)$ gives us a *modulo* η approximation of the desired result for A. The final ingredient for formalizing "lifting" is a category-theoretic formalization of the translation step (corresponding to applying the substitution $[\eta, \eta']$ for loop summarization, or the substitution $[\eta]$ for termination analysis), which is responsible for calculating $\overline{F}(A)$ from its modulo-η approximation $F(\alpha(A))$. For this purpose, we require *cartesian morphisms* and *fibred categories* (due to Grothendieck [1960]; see Barr and Wells [1995] for an introduction—we specialize to the language of concrete categories below).

Let $\mathbf{C}$ be a concrete category over $\mathbf{B}$. Let B be an object of $\mathbf{C}$, and $f : X \to \phi_{\mathbf{C}}(B)$ be an arrow of $\mathbf{B}$. A $\mathbf{C}$-arrow $u : A \to B$ is **cartesian** (for f and B) if $\phi_{\mathbf{C}}(A) = X$, $\phi_{\mathbf{C}}(u) = f$, and for any object $Z \in \mathbf{C}$ and arrow $h : \phi_{\mathbf{C}}(Z) \to \phi_{\mathbf{C}}(B)$ such that $Z \overset{h \circ f}{\Longmapsto} B$, we have $Z \overset{h}{\Longmapsto} A$. We use $A \overset{f}{\Longmapsto\!\!\!\!\!\rightarrow} B$ to denote that there exists a cartesian arrow $u \in \mathbf{C}(A, B)$ for f and B. A particular case that will be used in this paper (obtained by specializing h to the identity morphism) is:

(†) If $A \overset{f}{\Longmapsto\!\!\!\!\!\rightarrow} B$ and $Z \overset{f}{\Longmapsto} B$, then $Z \preceq A$.

Say that a concrete category $\mathbf{C}$ is **fibred** (over its base category $\mathbf{B}$) if (1) every object B of $\mathbf{C}$ and arrow $f : A \to \phi_{\mathbf{C}}(B)$ of $\mathbf{B}$ has a cartesian arrow, and (2) the composition of cartesian arrows is cartesian. In the following, we suppose that there is a canonical choice of cartesian arrow for any f and B, and denote its domain by $f^{-1}(B)$. Then we observe that $f^{-1}(B) \overset{f}{\Longmapsto\!\!\!\!\!\rightarrow} B$, and that if $A \overset{f}{\Longmapsto\!\!\!\!\!\rightarrow} B$ and $B \overset{g}{\Longmapsto\!\!\!\!\!\rightarrow} C$, then $A \overset{g \circ f}{\Longmapsto\!\!\!\!\!\rightarrow} C$.

Example 5.2. $\mathbf{TF}$ is fibred over $\mathbf{Vect}_{\mathbb{Q}}$. For any transition formula $F \in \mathbf{TF}(X)$ and any linear map $\sigma : \mathbb{Q}^Y \to \mathbb{Q}^X$, we have that $\sigma \in \mathbf{TF}(F[\sigma, \sigma'], F)$ is cartesian.

Similarly, $\mathbf{SF}$ is fibred over $\mathbf{Vect}_{\mathbb{Q}}$. For any state formula $G \in \mathbf{SF}(X)$ and any linear map $\sigma : \mathbb{Q}^Y \to \mathbb{Q}^X$, we have that $\sigma \in \mathbf{SF}(G[\sigma], G)$ is cartesian. ⌟

Definition 3. Let $\mathbf{B}$ be a category, let $\mathbf{C}$, $\mathbf{D}$, $\mathbf{R}$ be concrete categories over $\mathbf{B}$, and let $\gamma : \mathbf{D} \to \mathbf{C}$ and $F : \mathbf{D} \to \mathbf{R}$ be concrete functors. Suppose that γ has a weak left adjoint $\langle \alpha, \eta \rangle$ and that $\mathbf{R}$ is fibred. Define the *lifting of* F to be $\overline{F}(A) \triangleq \eta_A^{-1}(F(\alpha(A)))$.

Theorem 1 (Robustness of lifted operations)**.** We have that $\overline{F}$ is a concrete functor from **C** to **R**.

Proof. We must show that for any $A \overset{f}{\Mapsto} B$ in **C**, we have $\overline{F}(A) \overset{f}{\Mapsto} \overline{F}(B)$. Since $A \overset{f}{\Mapsto} B$ and $B \overset{\eta_B}{\Mapsto} \gamma(\alpha(B))$, composing gives $A \overset{\eta_B \circ f}{\Mapsto} \gamma(\alpha(B))$. The universal property of weak left adjoints yields $\overline{f}$ with $\alpha(A) \overset{\overline{f}}{\Mapsto} \alpha(B)$ and $\overline{f} \circ \eta_A = \eta_B \circ f$.

$$\begin{array}{ccc} \phi_{\mathbf{C}}(A) & \xrightarrow{\eta_A} & \phi_{\mathbf{D}}(\alpha(A)) \\ f \downarrow & & \vdots\, \overline{f} \\ \phi_{\mathbf{C}}(B) & \xrightarrow[\eta_B]{} & \phi_{\mathbf{D}}(\alpha(B)) \end{array}$$

Since F is a concrete functor, we have $F(\alpha(A)) \overset{\overline{f}}{\Mapsto} F(\alpha(B))$. Composing with $\overline{F}(A) \overset{\eta_A}{\twoheadrightarrow} F(\alpha(A))$ and using $\overline{f} \circ \eta_A = \eta_B \circ f$, we get $\overline{F}(A) \overset{\eta_B \circ f}{\twoheadrightarrow} F(\alpha(B))$. Meanwhile, $\overline{F}(B) \overset{\eta_B}{\twoheadrightarrow} F(\alpha(B))$ and $f^{-1}(\overline{F}(B)) \overset{f}{\Mapsto} \overline{F}(B)$; composing cartesian arrows gives $f^{-1}(\overline{F}(B)) \overset{\eta_B \circ f}{\twoheadrightarrow} F(\alpha(B))$. We have two arrows to $F(\alpha(B))$ over $\eta_B \circ f$, the latter cartesian; by (†), we have $\overline{F}(A) \preceq f^{-1}(\overline{F}(B))$, and thus $\overline{F}(A) \overset{f}{\Mapsto} \overline{F}(B)$. □

5.3 Soundness of Lifted Operations

This section shows that lifting preserves soundness: a sound operation on **D** lifts to a sound operation on **C**.

Say that a functor F is **cartesian** if for every cartesian arrow f, $F(f)$ is also cartesian. If we think of a concretization functor $\gamma_R : \mathbf{R}^\sharp \to \mathbf{R}^\natural$, then γ_R being cartesian intuitively means that inverse images in $\mathbf{R}^\sharp$ are sound with respect to $\mathbf{R}^\natural$. In particular, the concretization functor from **TF** to **TS** is cartesian, since **TF** is closed under inverse images of linear maps. Similarly, $\gamma_R : \mathbf{SF} \to \mathbf{SubSet}$ is cartesian.

Theorem 2 (Soundness of lifted operations)**.** Suppose that we have the following (not commuting) diagram of concrete categories over **B**:

$$\begin{array}{ccc} \mathbf{D} & \xrightarrow{F^\sharp} & \mathbf{R}^\sharp \\ {\scriptstyle\gamma_D}\downarrow & \overset{\overline{F}^\sharp}{\nearrow} & \downarrow{\scriptstyle\gamma_R} \\ \mathbf{C} & \xrightarrow[F^\natural]{} & \mathbf{R}^\natural \end{array}$$

where:

- **C**, **D**, $\mathbf{R}^\natural$, and $\mathbf{R}^\sharp$ are concrete categories over **B**;
- $\gamma_D : \mathbf{D} \to \mathbf{C}$ is a concrete functor that has a weak left adjoint $\langle \alpha, \eta \rangle$;
- $F^\natural : \mathbf{C} \to \mathbf{R}^\natural$, $F^\sharp : \mathbf{D} \to \mathbf{R}^\sharp$; and $\gamma_R : \mathbf{R}^\sharp \to \mathbf{R}^\natural$ are concrete functors;
- $F^\sharp$ is sound with respect to $F^\natural \circ \gamma_D$.

Supposing that $\mathbf{R}^\sharp$ is fibred (so that the lifted operation $\overline{F}^\sharp : \mathbf{C} \to \mathbf{R}^\sharp$ is well-defined) and γ_R is cartesian, then $\overline{F}^\sharp$ is sound with respect to $F^\natural$.

Proof. For any object A of **C**, we must show $F^\natural(A) \preceq \gamma_R(\overline{F}^\sharp(A))$. Since $F^\natural$ is a concrete functor and $A \overset{\eta_A}{\Mapsto} \gamma_D(\alpha(A))$, we have $F^\natural(A) \overset{\eta_A}{\Mapsto} F^\natural(\gamma_D(\alpha(A)))$. Soundness of $F^\sharp$ on $\alpha(A)$ gives $F^\natural(\gamma_D(\alpha(A))) \preceq \gamma_R(F^\sharp(\alpha(A)))$. Composing the above we get $F^\natural(A) \overset{\eta_A}{\Mapsto} \gamma_R(F^\sharp(\alpha(A)))$. By definition, $\overline{F}^\sharp(A) = \eta_A^{-1}(F^\sharp(\alpha(A)))$, giving a cartesian arrow $\overline{F}^\sharp(A) \overset{\eta_A}{\twoheadrightarrow} F^\sharp(\alpha(A))$. Since γ_R is a cartesian functor, $\gamma_R(\overline{F}^\sharp(A)) \overset{\eta_A}{\twoheadrightarrow} \gamma_R(F^\sharp(\alpha(A)))$. We now have two arrows to $\gamma_R(F^\sharp(\alpha(A)))$ over η_A, the latter cartesian; by (†), we have $F^\natural(A) \preceq \gamma_R(\overline{F}^\sharp(A))$. □

5.4 Algebraic Properties of Lifted Operations

This section shows that (in a certain sense) if an operation $F : \mathbf{D} \to \mathbf{R}$ satisfies a certain kind of (in)equational law, then that law is also satisfied by its lifting $\overline{F} : \mathbf{C} \to \mathbf{R}$ (obtained by Definition 3). Such laws amount to another kind of robustness guarantee (further explored in Section 6).

As an example, suppose that **D** is a subcategory of **TF** equipped with a reflexive transitive closure operator $(-)^\star : \mathbf{D} \to \mathbf{TF}$, and let $(-)^{\overline{\star}} : \mathbf{TF} \to \mathbf{TF}$ denote its lifting. Since for any transition formula F in **D**, $F^\star$ is transitively closed, we have the law $\forall F \in Ob(\mathbf{D}).F^\star \cdot F^\star \leq F^\star$, or equivalently

$\forall F \in Ob(\mathbf{D}).(F^\star)^2 \leq F^\star$ (where $(-)^2$ is a functor mapping each transition formula G to $G \cdot G$). As a consequence of Theorem 3 below, we can raise this inequation to one for the lifted operator $(-)^{\overline{\star}}$, yielding the law $\forall F \in Ob(\mathbf{TF}).(F^{\overline{\star}})^2 \leq F^{\overline{\star}}$.

In the technical statement of the lifting theorem (below), $G : \mathbf{D} \to \mathbf{R}$ is an operation that is to be lifted to $\overline{G} : \mathbf{C} \to \mathbf{R}$. The left-hand side and right-hand side of an inequational law are represented as the composition of functors F_1GH and F_2G, respectively. To instantiate this theorem to lift the transitivity property above, we set F_1 to $(-)^2$, G to $(-)^\star$, and H and F_2 to identity functors. In [Kincaid and Zhu 2026, Appendix A], we show how several more laws can be obtained by instantiating these functors appropriately, including:

- *Transitivity:* $F^{\overline{\star}} \cdot F^{\overline{\star}} = F^{\overline{\star}}$ (Corollary 1)
- *Unrolling:* $(F^n)^{\overline{\star}} \leq F^{\overline{\star}}$ (Corollary 2)

Theorem 3 (Lifting)**.** Let $\mathbf{B}$ be a category, and let $\mathbf{C}$, $\mathbf{D}$, $\mathbf{R}$, and $\mathbf{R}'$ be concrete categories over $\mathbf{B}$. Let $\gamma : \mathbf{D} \to \mathbf{C}$ be a concrete functor that has a weak left adjoint $\langle \alpha, \eta \rangle$. Let $H : \mathbf{D} \to \mathbf{D}$ and $\widetilde{H} : \mathbf{C} \to \mathbf{C}$ be concrete functors such that $\widetilde{H}\gamma = \gamma H$. Let $G : \mathbf{D} \to \mathbf{R}$, $F_1 : \mathbf{R} \to \mathbf{R}'$, and $F_2 : \mathbf{R} \to \mathbf{R}'$ be concrete functors. Suppose that $\mathbf{R}$ is fibred, and that F_2 is a cartesian functor. Let $\overline{G}$ be the lifting of G. If $F_1(G(H(d))) \preceq F_2(G(d))$ for all objects d of $\mathbf{D}$, then $F_1(\overline{G}(\widetilde{H}(c))) \preceq F_2(\overline{G}(c))$ for all objects c of $\mathbf{C}$.

6 Robust Algebraic Program Analysis

The previous section demonstrated a recipe for designing robust program analysis operations, which generalizes a pattern seen in several algebraic program analyses (Table 3). In this section, we investigate the robustness properties of such analyses. That is, assuming that each individual operation of an algebraic program analysis is robust, in what sense is the overall analysis robust?

An algebraic program analysis tool can be viewed as consisting of a front-end and a back-end. The front-end translates the program into a flow graph G with weights drawn from some algebra A (as in Figure 4). The back-end computes a summary assignment from G—a function $s : V_G \to A$ that maps each vertex v to a summary $s(v)$ that over-approximates the semantics of all program paths from the entry r_G to v, via a *summarization* algorithm such as Tarjan [1981a]'s. To formalize robustness of an algebraic analysis, we restrict our attention to the back-end—that is, we consider transformations in flow graphs, and the sense in which (some) summarization algorithms are robust (assuming that each algebraic operation is robust).

To motivate our discussion, consider the flow graphs in Figure 4. G_1 simulates G_2 in the sense that there is a substitution $f = [i \mapsto x]$ mapping the variables of G_2 to linear terms over the variables of G_1, and a function $h : V_{G_1} \to V_{G_2}$ from the vertices of G_1 to the vertices of G_2 such that for each edge $\langle u, v \rangle$ in G_2, there is a corresponding path in G_1 from $h(u)$ to $h(v)$ that simulates it (modulo the transformation f).[4] When an algebraic program analysis computes a summary assignment $s_1 : V_{G_1} \to \mathbf{TF}(\{i\})$ for G_1 and $s_2 : V_{G_2} \to \mathbf{TF}(\{x, y\})$, they are similarly related: for each vertex v, $s_1(f(v))$ simulates $s_2(v)$ (modulo the transformation f). This is the case for all of the analyses derived using the pattern in Section 5.

The input to an algebraic program analysis is a flow graph G with weights drawn from some algebra A, and the output is a summary assignment $s : V_G \to A$. Following the framework introduced in Section 4, we wish to define concrete categories for flow graphs and summary assignments (over some common base category) such that summarization is robust (under the assumption that each algebraic operation is robust). The remainder of this section formalizes robust algebraic program

[4]For $\langle 1, 2 \rangle$, $\langle 3, 4 \rangle$, and $\langle 4, 5 \rangle$, the simulating path is simply the corresponding edge in G_1; edges $\langle 2, 3 \rangle$ and $\langle 5, 3 \rangle$ do not change the value of x, and the simulating path is the empty path from B to B in G_1.

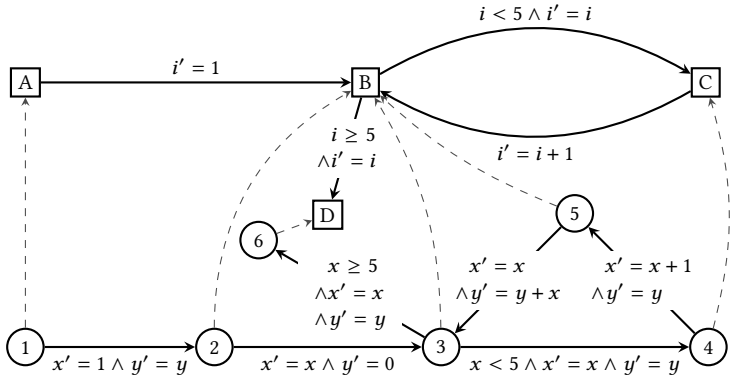


Fig. 4. Two flow graphs G_1 (above) and G_2 (below), corresponding to the programs P_1 from Figure 1a and P_2 from Figure 1b. The dashed lines indicate a stuttering simulation $\langle h, f \rangle$ (where h maps $1 \mapsto$ A, $\{2, 3, 5\} \mapsto$ B, $4 \mapsto$ C, $6 \mapsto$ D, and $f = [i \mapsto x]$) that preserves loops from the flow graph below to the one above. The summaries at the loop headers B and 3 are the loop summaries of Table 2.

analysis and summary assignments (Section 6.1), vertex elimination (Section 6.2), and a class of relationships under which it is robust (Section 6.3).

6.1 Robust Algebraic Program Analysis

This section defines the structure of a robust algebraic program analysis. Informally, an algebraic program analysis is robust (with respect to some base category **B**) if all of its operators are robust and satisfy the Pre-Kleene algebra laws. To formalize this, we must generalize Pre-Kleene algebras to a setting wherein the universe is a concrete category **C** over **B** rather than a set, and operators are concrete functors rather than functions. An n-ary operator on **C** corresponds to functor from the n-fold product $\mathbf{C} \times_{\mathbf{B}} \cdots \times_{\mathbf{B}} \mathbf{C}$ to **C**, where the product operator is defined as follows (and noting that "0 times" **C** is **B**).

Definition 4. For any concrete categories **M** and **N** over a category **B**, define $\mathbf{M} \times_{\mathbf{B}} \mathbf{N}$ to be the pullback of $\phi_{\mathbf{M}}$ and $\phi_{\mathbf{N}}$; that is, $\mathbf{M} \times_{\mathbf{B}} \mathbf{N}$ is the category in which the objects are pairs $\langle M, N \rangle$ consisting of an object M of **M** and an object N of **N** such that $\phi_{\mathbf{M}}(M) = \phi_{\mathbf{N}}(N)$, and an arrow $(A, B) \to (A', B')$ is a pair $\langle f, g \rangle$ such that $f \in \mathbf{M}(A, A')$ and $g \in \mathbf{N}(B, B')$ and $\phi_{\mathbf{M}}(f) = \phi_{\mathbf{N}}(g)$. $\mathbf{M} \times_{\mathbf{B}} \mathbf{N}$ naturally forms a concrete category over **B**.

For instance, consider the category of transition formulas **TF**. The sequential composition operator is a *partial* function on **TF**, which is defined for any pair of transition formulas F and G that operate on the same set of variables (formalized in three equivalent ways: (1) $\phi_{\mathbf{TF}}(F) = \phi_{\mathbf{TF}}(G)$; (2) (F, G) is an object of $\mathbf{TF} \times_{\mathbf{Vect}_{\mathbb{Q}}} \mathbf{TF}$; (3) F and G belong to the same fiber of $\phi_{\mathbf{TF}}$). Sequential composition is robust in the sense that for any transition formulas $F, F' \in \mathbf{TF}(X)$, and $G, G' \in \mathbf{TF}(Y)$ and any linear transformation $f : \mathbb{Q}^X \to \mathbb{Q}^Y$ such that f is a simulation from F to G and F' to G', we have that f is a simulation from $F \cdot F'$ to $G \cdot G'$—this corresponds precisely to the fact that sequential composition is a concrete functor $\mathbf{TF} \times_{\mathbf{Vect}_{\mathbb{Q}}} \mathbf{TF} \to \mathbf{TF}$. The $+$, 0, and 1 operators can similarly be understood as concrete functors, as well as any lifted iteration operator following the recipe of Section 5. Generalizing this structure, we have the following definition.

Definition 5. Let **B** be a category. A **robust algebraic analysis** (over **B**) is a concrete category **A** over **B** equipped with concrete functors $0 : \mathbf{B} \to \mathbf{A}$, $1 : \mathbf{B} \to \mathbf{A}$, $(-) \cdot (-) : \mathbf{A} \times_{\mathbf{B}} \mathbf{A} \to \mathbf{A}$, $(-) + (-) : \mathbf{A} \times_{\mathbf{B}} \mathbf{A} \to \mathbf{A}$, and $(-)^\star : \mathbf{A} \to \mathbf{A}$ such that

(1) Each fiber of **A** forms a Pre-Kleene algebra;

(2) For any objects a and b, we have $a \preceq b$ iff $a \leq b$ (i.e., the approximation order derived from the concrete category structure coincides with the natural order from the PKA structure).

Recall from Section 5.4 that "lifted" operators (derived from a base operator using the recipe of Section 5) satisfy many of the same algebraic laws as their base. As a consequence, **TF** forms a robust algebraic analysis (over $\mathbf{Vect}_{\mathbb{Q}}$) where the $(-)^\star$ operator is instantiated to the polyhedral iteration operator from Section 2, or the operator of [Cyphert and Kincaid 2024].

For any arrow f of $\mathbf{B}$ and any A, B in $Ob(\mathbf{A})$ such that $A \overset{f}{\Longmapsto} B$, we say that B **simulates** A **modulo** f. Assuming that $\mathbf{A}$ is a robust algebraic analysis, observe that the $\overset{f}{\Longmapsto}$ relation is compatible with all of the operations and the natural order; that is,

Observation 1 (Compatibility)**.** For all A_1, A_2, B_1, B_2 such that $A_1 \overset{f}{\Longmapsto} B_1$ and $A_2 \overset{f}{\Longmapsto} B_2$:

- $A_1 + A_2 \overset{f}{\Longmapsto} B_1 + B_2$
- $A_1 \cdot A_2 \overset{f}{\Longmapsto} B_1 \cdot B_2$
- $A_1^\star \overset{f}{\Longmapsto} B_1^\star$
- $0 \overset{f}{\Longmapsto} 0$
- $1 \overset{f}{\Longmapsto} 1$
- If $A_1 \leq A_2$, $A_2 \overset{f}{\Longmapsto} B_1$, and $B_1 \leq B_2$, then $A_1 \overset{f}{\Longmapsto} B_2$

Finally we may define the category $\mathbf{S_A}$ of summary assignments for a robust algebraic analysis $\mathbf{A}$. The objects of $\mathbf{S_A}$ are summary assignments—i.e., functions $s : U \to \phi_{\mathbf{A}}^{-1}(X)$ for some set U (of vertices) and some $X \in Ob(\mathbf{B})$. An arrow in $\mathbf{S_A}$ from a summary assignment $s : U \to \phi_{\mathbf{A}}^{-1}(X)$ to $t : V \to \phi_{\mathbf{A}}^{-1}(Y)$ consists of a pair $\langle h, f \rangle$, comprising a "structure transformation" $h : U \to V$ and "weight transformation" $f \in \mathbf{B}(X, Y)$ such that for all $u \in U$, we have that $s(u) \overset{f}{\Longmapsto} t(h(u))$ (that is, for each vertex $u \in U$, the summary assigned to u by s is simulated by the summary assigned to its corresponding vertex $h(u)$ by t, modulo the transformation f). Thus, the natural base category over which to understand summary assignments is the product category $\mathbf{Set} \times \mathbf{B}$, where for a summary assignment $s : U \to \phi_{\mathbf{A}}^{-1}(X)$, we have $\phi_{\mathbf{S_A}}(s) = \langle U, X \rangle$.

6.2 Robustness of Summarization Under Isomorphisms

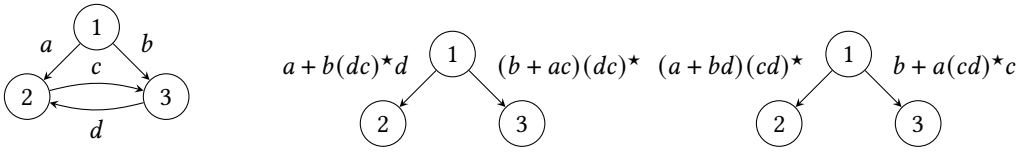
We present an abstract view of summarization algorithms based on *vertex elimination*, which can be thought of as a variant of Gauss-Jordan elimination, or Kleene's algorithm for converting a finite automaton to a regular expression. Practical summarization algorithms, such as Tarjan [1981b]'s, can be thought of as vertex elimination algorithms that use various techniques to improve computational efficiency, but produce the same results as vertex elimination (assuming that the underlying algebra obeys the semiring laws).

In defining the vertex elimination operation, it is convenient to extend the weight function w_G to the domain $V_G \times V_G$, by assigning each pair u and v with $\langle u, v \rangle \notin E_G$ the weight 0. We do not distinguish between a weight function and its total extension. Eliminating a vertex v from a graph G produces another flow graph G/v in which v has no outgoing edges, but the paths in G are preserved in the sense that for any path π in G from r_G to any vertex u, there is a path π' in G/v such that $w_G(\pi) \leq w_{G/v}(\pi')$. The graph G/v has the same vertices and root as G, and its edges are defined as follows:

$$E_{G/v} \triangleq \{\langle u, u' \rangle \in E : u' \neq v\} \cup \{\langle p, s \rangle : \langle p, v \rangle \in E \wedge \langle v, s \rangle \in E\}$$

$$w_{G/v}(u, u') \triangleq \begin{cases} w_G(u, u') + w_G(u, v) w_G(v, v)^\star w_G(v, u') & u \neq v, u' \neq v \\ w_G(u, v) w_G(v, v)^\star & u' = v \\ 0 & \text{otherwise} \end{cases}$$

We extend this notation to sequences of vertices by defining $G/\epsilon = G$ and $G/v_1, v_2, \ldots, v_n = (G/v_1)/v_2, \ldots, v_n$. To compute a summary assignment using vertex elimination, we enumerate the non-root vertices of G in some order $v_1, \ldots, v_n$ and then eliminate them to yield a graph $H \triangleq G/v_1, \ldots, v_n$. Since eliminated vertices have no outgoing edges, the only edges in H are of the



(a) A flow graph G (b) $G/2/3$ (c) $G/3/2$

Fig. 5. An irreducible flow graph G along with the result of eliminating vertices 2 and 3 in both possible orders.

form $\langle r_G, v\rangle$, and the weight of this edge must approximate the weight of any path from r_G to v in G. Thus, the function mapping $v \mapsto w_H(r_G, v)$ is a sound summary assignment.

If A happens to be a Kleene algebra, then the elimination order is irrelevant. But if A is a *Pre-Kleene algebra*, different elimination orders can produce different results (i.e., different sound over-approximations of program behavior). From this abstract perspective, the crucial difference between different summarization algorithms is the selection of elimination orders.

The most basic question concerning the robustness of a summarization algorithm is whether it is robust with respect to graph isomorphisms—that is, given flow graphs G and H that differ only in the names of vertices, does the algorithm produce exactly the same result for G and H? The answer is, unfortunately, no. For instance, consider the flow graph in Figure 5a. The map that swaps the vertices 2 and 3 is a structural isomorphism for this graph, so we cannot depend on the order that an algorithm will eliminate them. However, the order of elimination matters—in particular, for the order 2, 3 the $(-)^\star$ operator is applied to dc, and for the 3, 2 order it is applied to cd, and in general there is no relationship between $(dc)^\star$ and $(cd)^\star$.

For reducible flow graphs, however, Tarjan [1981a]'s algorithm *is* robust with respect to isomorphisms. To prove this result, we define a class of *admissible* elimination orders, and show that any two admissible elimination orders of the same reducible graph produce the same analysis result. It follows that any vertex elimination algorithm that eliminates vertices in an admissible order (of which Tarjan's algorithm is an example) is robust with respect to isomorphisms (for reducible flow graphs).

Let G be a flow graph. Say that u **dominates** v if every path from r_G to v passes through u. Dominance forms a partial order, with least element r_G. A flow graph G is said to be **reducible** if for every cycle in G, there is some vertex along the cycle that dominates all other vertices. A **local cycle** of a vertex v is a path π in G such that (1) $v = src(\pi) = dst(\pi)$, and (2) v dominates every vertex along π. Define $L_G(v)$ to be the set of vertices that appear in some local cycle of v. Say that a sequence $v_1, \ldots, v_n$ of vertices is **admissible** if there exists no $i < j$ such that $v_j \in L(v_i)$. Intuitively, one could think that vertices in an inner loop should be eliminated before those in the outer loop.

Theorem 4. Let G be a reducible flow graph, and let $v_1, \ldots, v_n$ and $u_1, \ldots, u_n$ be enumerations of some subset $U \subseteq V_G$. If both $v_1, \ldots, v_n$ and $u_1, \ldots, u_n$ are admissible, then $G/v_1, \ldots, v_n = G/u_1, \ldots, u_n$.

In the following, we use $[\![G]\!]$ to denote the summary assignment produced by a vertex elimination algorithm that eliminates vertices in a reducible order. Formally, let G be a reducible flow graph over a PKA A. Define $[\![G]\!]$ to be the function $V_G \to A$ that maps each vertex $v \in V_G$ to $w_H(r_G, v)$, where $H \triangleq G/v_1, \ldots, v_n$ for some admissible order $v_1, \ldots, v_n$ of the non-root vertices of G. $[\![G]\!]$ is well-defined by Theorem 4 above.

6.3 Robustness of Summarization Under Loop-Preserving Stuttering Simulations

Let $\mathbf{A}$ be a robust algebraic program analysis over some base category $\mathbf{B}$. Recall that the natural base category over which we understand the results of vertex elimination is $\mathbf{Set} \times \mathbf{B}$. In this section,

we wish to define a category $\mathbf{G_A}$ of *reducible* flow graphs over $\mathbf{A}$ such that $[\![-]\!]$ can be understood to be a concrete functor from $\mathbf{G_A}$ to $\mathbf{S_A}$. Given that we wish to view $\mathbf{G_A}$ as a concrete category over $(\mathbf{Set} \times \mathbf{B})$, we can make this question more concrete. Suppose G and H are flow graphs with weights in $\phi_{\mathbf{A}}^{-1}(X)$ and $\phi_{\mathbf{A}}^{-1}(Y)$ respectively (for some $X, Y \in Ob(\mathbf{B})$). For which functions $h : V_G \to V_H$ and arrows $f : X \to Y$ should the pair $\langle h, f \rangle$ be considered an acceptable transformation from G to H (such that we may conclude that $\langle h, f \rangle$ is an arrow from $[\![G]\!]$ to $[\![H]\!]$)? The answer provided in this section is that $\langle h, f \rangle$ should be a *stuttering simulation* that *preserves loops*.

Figure 4 depicts an example of a stuttering simulation between two flow graphs. The intuition behind simulation is that the simulator has *at least* as many behaviors in the sense that every execution of the simulatee can be mapped to a corresponding execution of the simulator. In the abstract, we view "executions" as paths in a flow graph, and correspondence as a consistent approximation relation on weights of those paths.

Definition 6. Let $\mathbf{A}$ be a robust algebraic program analysis over a base category $\mathbf{B}$. Let G and H be flow graphs over $\phi_{\mathbf{A}}^{-1}(X)$ and $\phi_{\mathbf{A}}^{-1}(Y)$, respectively. A **stuttering simulation** is a pair $\langle h, f \rangle$ where $h : V_G \to V_H$ and $f \in \mathbf{B}(X, Y)$ such that for each edge $\langle u, v \rangle \in E_G$, we have

- $\langle h(u), h(v) \rangle \in E_H$ and $w_G(u, v) \overset{f}{\Longmapsto} w_H(h(u), h(v))$, *or*
- $h(u) = h(v)$ and $w_G(u, v) \overset{f}{\Longmapsto} 1$

Observe that for any stuttering simulation $\langle h, f \rangle$ between flow graphs G and H there exists a function r that maps any path π in G to a simulating path $r(\pi)$ in H, which can be constructed as:

$$r(\epsilon) = \epsilon$$

$$r(\pi\, \langle u, v \rangle) = \begin{cases} r(\pi)\, \langle h(u), h(v) \rangle & \text{if } \langle h(u), h(v) \rangle \in E_H \text{ and } w_G(u, v) \overset{f}{\Longmapsto} w_H(h(u), h(v)) \\ r(\pi) & \text{if } h(u) = h(v) \text{ and } w_G(u, v) \overset{f}{\Longmapsto} 1 \end{cases}$$

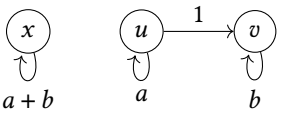
Note that the above does not define r uniquely, because an edge $\langle u, v \rangle$ may be simulated both by 1 and by $w_H(h(u), h(v))$. We say that any function meeting the conditions above is a **witness** for the stuttering simulation $\langle h, f \rangle$.

Example 6.1. Sharma et al. [2011] introduce a technique for improving the precision of static analyses by splitting loops into phases—an example of the transformation appears below (the program on the left is transformed into the program on the far right). This kind of transformation *heuristically* improves precision, but in some cases can actually decrease precision. For robust algebraic program analyses, the transformation always improves precision (not necessarily strictly): there is a stuttering simulation from the transformed program to the original. A schematic illustration appears below to the right: $\langle h, 1 \rangle$ is a stuttering simulation from the rightmost graph to the one to its left, where $h = \{u \mapsto x, v \mapsto x\}$.

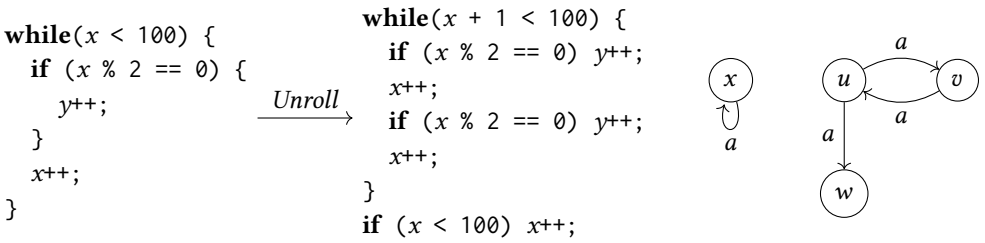
```
while(x < 100) {
  x++;
  if (x > 50) y++;
}
```

Split →

```
while(x ≤ 49) { x++; }
while(x < 100 ∧ x > 49) {
  x++; y++;
}
```



Example 6.2. Loop unrolling is a transformation widely used in compilers to improve performance by reducing the overhead of loop-control branches and exposing more instructions to the scheduler [Aho et al. 2006; Hennessy and Patterson 2011]. For robust algebraic program analyses, this transformation is (not necessarily strictly) precision-improving because a stuttering simulation exists from the unrolled program (right) to the original (left): $\langle h, 1 \rangle$ is a stuttering simulation from the transformed graph to the original, where $h = \{u \mapsto x, v \mapsto x, w \mapsto x\}$.



Informally, vertex elimination is robust with respect to stuttering simulations that *also* respect syntactic loop structure, such as nesting hierarchy. Such simulations (which include Examples 6.1 and 6.2) are called *loop-preserving*. We will now make loop preservation precise. For a path $\pi = e_1 e_2 \dots e_n$ and a vertex $u \in V_G$, define the u-**length** of π to be the number of edges e_i such that the source of e_i is u. Denote the u-length of a path π by $|\pi|_u$.

Definition 7. Let G and H be flow graphs, and let $\langle h, f \rangle$ be a stuttering simulation from G to H. Say that $\langle h, f \rangle$ is **loop-preserving** if

- *(Loop membership)* For any $u, v \in V_G$, if $u \in L_G(v)$, then $h(u) \in L_H(h(v))$;
- *(Non-nesting)* For any distinct $u, v \in V_G$ such that $h(u) = h(v)$ and $u \in L_G(v)$, we must have $L_G(u) = \emptyset$;
- *(Consistent unrolling)* There is a witness r for $\langle h, f \rangle$ such that for any $v \in V_G$, there is some $n \geq 1$ such that every primitive local cycle (meaning that it is not obtained by repeating some other local cycles multiple times) π of v satisfies $|r(\pi)|_{h(v)} = n$.

We now present the main result: for any robust algebraic program analysis $\mathbf{A}$, vertex elimination $[\![-]\!] : \mathbf{G_A} \to \mathbf{S_A}$ is a robust operation.

Theorem 5. Let $\mathbf{A}$ be a robust algebraic program analysis over a category $\mathbf{B}$. Let G and H be reducible flow graphs, and let $\langle h, f \rangle$ be a stuttering simulation from G to H that is loop-preserving. Then for any vertex $v \in V_G$, we have that $[\![G]\!]\,(v)$ is simulated by $[\![H]\!]\,(h(v))$ modulo f.

7 Conclusion

It is the opinion of the authors that *robustness* should be an important design consideration for practical program analyses. We provide a unified framework for articulating a class of robustness properties in the language of concrete category theory, by thinking of robustness as preserving certain relationships between programs.

We give a general design strategy for achieving robustness, and establish robustness of several existing analyses by way of showing that they are instances of this strategy. A natural avenue for future work is to instantiate this strategy in new ways—i.e., for what other sub-Turing models of computation, and what other classes of simulations, can this recipe be followed? We also study robustness of algebraic program analyses "in the large"—that is, how guarantees of robustness of individual program operations translate to end-to-end analysis guarantees upon which users may rely. A limitation of this theory is that we have restricted our attention to *reducible* flow graphs. We leave open the question of whether robustness can be established for irreducible flow graphs, and the correct notion of simulation for algebraic termination analysis.

Acknowledgments

This work was supported in part by the NSF under grant number 1942537. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

Jirí Adámek, Horst Herrlich, and George E. Strecker. 2009. *Abstract and Concrete Categories - The Joy of Cats.* Dover Publications.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

Gianluca Amato, Maurizio Parton, and Francesca Scozzari. 2010. Deriving Numerical Abstract Domains via Principal Component Analysis. In *Static Analysis*, Vol. 6337. 134–150. doi:10.1007/978-3-642-15769-1_9

Corinne. Ancourt, Fabien Coelho, and François Irigoin. 2010. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electr. Notes Theor. Comp. Sci.* 267, 1 (Oct. 2010), 3–16. doi:10.1016/j.entcs.2010.09.002

Michael Barr and Charles Wells. 1995. *Category theory for computing science (2. ed.).* Prentice Hall.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. doi:10.1145/1646353.1646374

Alexandru Costan, Stéphane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. 2005. A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs. In *CAV*. 462–475. doi:10.1007/11513988_46

Patrick Cousot. 2015. Abstracting Induction by Extrapolation and Interpolation. In *VMCAI*. 19–42. doi:10.1007/978-3-662-46081-8_2

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. doi:10.1145/512950.512973

John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. 2019. Refinement of Path Expressions for Static Analysis. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–29. doi:10.1145/3290358

John Cyphert and Zachary Kincaid. 2024. Solvable Polynomial Ideals: The Ideal Reflection for Program Analysis. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 724–752. doi:10.1145/3632867

Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*. doi:10.1109/FMCAD.2015.7542253

Alexander Grothendieck. 1960. Technique de descente et théorèmes d'existence en géométrie algébrique. I. Généralités. Descente par morphismes fidèlement plats. In *Séminaire Bourbaki : années 1958/59 - 1959/60, exposés 169-204.* Number 5 in Séminaire Bourbaki. Société mathématique de France, 299–327. https://www.numdam.org/item/SB_1958-1960__5__299_0/ talk:190.

John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.

Paul C. Kainen. 1971. Weak Adjoint Functors. 122, 1 (1971), 1–9. doi:10.1007/BF01113560

Shin-ya Katsumata, Xavier Rival, and Jérémy Dubut. 2023. A Categorical Framework for Program Semantics and Semantic Abstraction. In *Math. Found. of Prog. Semantics (MFPS XXXIX) (ENTICS, Vol. 3)*. doi:10.46298/entics.12288

Zachary Kincaid. 2018. Numerical Invariants via Abstract Machines. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002)*, Andreas Podelski (Ed.). Springer, 24–42. doi:10.1007/978-3-319-99725-4_3

Zachary Kincaid, Thomas Reps, and John Cyphert. 2021. Algebraic Program Analysis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.), Vol. 12759. 46–83. doi:10.1007/978-3-030-81685-8_3

Zachary Kincaid and Shaowei Zhu. 2026. A Categorical Basis for Robust Program Analysis. arXiv:2604.11029 [cs.PL] https://arxiv.org/abs/2604.11029

K. Rustan M. Leino and Michał Moskal. 2010. Usable Auto-Active Verification. In *Usable Verification Workshop (UV 2010)*. Redmond, WA, USA.

K. Rustan M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 361–381. doi:10.1007/978-3-319-41528-4_20

Dorian Lesbre and Matthieu Lemerre. 2024. Compiling with Abstract Interpretation. *Proc. ACM Program. Lang.* 8, PLDI (2024), 368–393. doi:10.1145/3656392

Francesco Logozzo and Manuel Fähndrich. 2008. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *Comp. Construct.* 197–212. doi:10.1007/978-3-540-78791-4_14

David Monniaux and Julien Le Guen. 2012. Stratified Static Analysis Based on Variable Dependencies. *Electronic Notes in Theoretical Computer Science* 288 (Dec. 2012), 61–74. doi:10.1016/j.entcs.2012.10.008

Nikhil Pimpalkhare and Zachary Kincaid. 2024. Monotone Procedure Summarization via Vector Addition Systems and Inductive Potentials. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 1873–1899. doi:10.1145/3689777

Francesco Ranzato and Marco Zanella. 2018. Invertible Linear Transforms of Numerical Abstract Domains. In *Static Analysis*, Andreas Podelski (Ed.), Vol. 11002. 344–363. doi:10.1007/978-3-319-99725-4_21

Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg,

Berlin, Heidelberg, 703–719. doi:10.1007/978-3-642-22110-1_57

Jake Silverman and Zachary Kincaid. 2019. Loop Summarization with Rational Vector Addition Systems. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 97–115. doi:10.1007/978-3-030-25543-5_7

Bernhard Steffen, C. Barry Jay, and Michael Mendler. 1992. Compositional Characterization of Observable Program Properties. *RAIRO Theor. Informatics Appl.* 26, 5 (1992), 403–424. doi:10.1051/ita/1992260504031

Robert Endre Tarjan. 1981a. Fast Algorithms for Solving Path Problems. *J. ACM* 28, 3 (July 1981), 594–614. doi:10.1145/322261.322273

Robert Endre Tarjan. 1981b. A Unified Approach to Path Problems. *J. ACM* 28, 3 (July 1981), 577–593. doi:10.1145/322261.322272

Shaowei Zhu and Zachary Kincaid. 2021. Reflections on Termination of Linear Loops. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 51–74. doi:10.1007/978-3-030-81688-9_3

Received 2025-11-12; accepted 2026-04-03