The Impact of Program Transformations on Static Program Analysis

Kedar S. Namjoshi[1] and Zvonimir Pavlinovic[2]

[1] Bell Labs, Nokia kedar.namjoshi@nokia-bell-labs.com
[2] New York University zvonimir@cs.nyu.edu

Abstract. Semantics-preserving program transformations, such as those carried out by an optimizing compiler, can affect the results of static program analyses. In the best cases, a transformation increases precision or allows a simpler analysis to replace a complex one. In other cases, transformations have the opposite effect, reducing precision. This work constructs a theoretical framework to analyze this intriguing phenomenon. The framework provides a simple, uniform explanation for precision changes, linking them to bisimulation relations that justify the correctness of a transformation. It offers a mechanism for recovering lost precision through the systematic construction of a new, bisimulating analysis. Furthermore, it is shown that program analyses defined over a class of composite domains can be factored into a program transformation followed by simpler, equally precise analyses of the target program.

1 Introduction

It has been empirically observed that a semantics-preserving program transformation may alter the outcome of a static analysis, making the results more or less precise. Consider, for instance, the program on the left in Figure 1. A standard odd-even parity analysis will deduce that `x` is odd and `y` is even at the end of the program; but the parity of `z` is unknown, as the value of `y div x` could have either parity (consider $y = 10, x = 5$ and $y = 10, x = 3$). An application of constant propagation and folding, a standard compiler optimization that replaces expressions with equivalent constant values, produces the program on the right. Parity analysis on that program will deduce that `z` is even.

```
x := 3;                    x := 3;
y := x * 4;                y := 12;
z := (y div x)*z;          z := 4*z;
```

Fig. 1: A constant propagation transformation: source on left, target on right.

In this instance, the transformation enhances precision. Several tools (e.g., SMACK [4] and SeaHorn [17]) use transformations for this purpose. But not

all transformations enhance precision: as pointed out in [22], a translation to 3-address code can render certain relational analyses imprecise. Program analyses are, therefore, not robust under semantics-preserving transformations.

This observation raises three central questions: (1) How does an (arbitrary) transformation affect the results of an (arbitrary) analysis? (2) Is there a mechanism to recover lost precision? and (3) Are there systematic ways to simplify analysis through program transformation? In this work, we set up a mathematical framework to analyze these questions, and provide some answers.

The framework is built as follows. Static program analyses are modeled with standard concepts from abstract interpretation [7, 8]. Crucially, a semantics-preserving transformation is modeled as a proof-generator. In transforming a source program S to a program T, we suppose that a transformation also provides a bisimulation relation, B, which justifies the semantic equivalence between the two programs. The bisimulation links the state spaces of S and T, making it possible to transfer invariants (in particular, static analysis results) from one side to the other, allowing their relative precision to be compared. Using this framework, we establish general results that explain why precision is gained or lost, and how it may be regained.

We show that an analysis with domain D on program T can be converted to a *bisimulating analysis* on S, producing results that are (near-)equivalent – after transferring through the bisimulation – to the results on T. The bisimulating analysis is defined over a new abstract domain, D', constructed in terms of B and D. One can explain the effect of a transformation on precision by comparing the relative strengths of D and D'. This provides a uniform explanation of precision changes observed in different settings, including the ones discussed above.

Moreover, the analysis designed in [22] to counteract the loss of precision is essentially the analysis induced by the new domain D'. The construction of D' thus provides a systematic, general method to form a new domain and its associated analysis to recover from a loss of precision.

Finally, we establish that any analysis over a one-way reduced product of domains C and D can be factored into a program transformation defined using C, followed by an analysis of the resulting program over domain D, with equally precise results. This provides a systematic method to design transformations which simplify analysis without losing precision.

Together, these results provide a firmer understanding of how transformations influence precision. This should help in practice to choose (or construct) the right set of transformations to simplify an analysis task.

2 Overview

In this section we provide a high-level overview of how we model the effect of program transformations on static analyses. Table 1 summarizes the transformation and parity analysis for the introductory constant propagation example. Parity domain elements E and O represent even and odd numbers, respectively,

and $\top$ represents all integers. The analysis maintains an *abstract state* for every location, a map from variables to elements of the parity domain. To avoid clutter, we show only the changes to the abstract state.

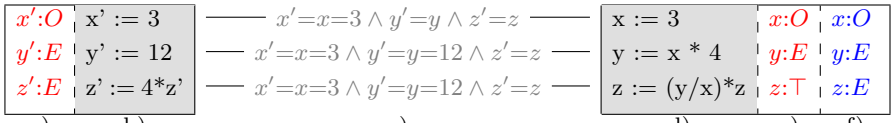

Table 1: a) the results of the parity analysis for the optimized program, b) the optimized program, c) the bisimulation relation witnessing the correctness of the optimization, d) the original program, e) the results of the parity analysis on the original program, f) the results of the bisimulating parity analysis

.

Bisimulation relation. The relation is symbolically presented in Table 1c). The bisimulation relates corresponding program states iff (1) they share the same location and (2) their variable valuations satisfy the predicates appearing on the horizontal line connecting identical program locations.

Bisimulating analysis. The new *bisimulating analysis* combines the parity domain and the bisimulation relation. It operates on the original program as follows. In each step, the analysis uses the bisimulation to move from the source to the transformed program, transforming the current abstract state through the bisimulation. Parity analysis on the optimized program with the transformed state produces a new abstract state, which is back-propagated to the source, again using the bisimulation (technically, its inverse). In effect, this process refines abstract states using bisimulation information.

Consider the point just before the last line of the source. The current abstract state, $[x : O, y : E, z : \top]$, is transferred to the same location in the transformed program using the middle horizontal line. This results in the abstract state $[x' : O, y' : E, z' : \top]$, which parity analysis uses to analyze the last command. This produces $[x' : O, y' : E, z' : E]$, which is back-propagated using the bottom horizontal line, resulting in the state $[x : O, y : E, z : E]$, as shown in Table 1f).

Precision. One can view the bisimulating analysis, roughly speaking, as operating on a domain that is a product of the parity domain and the domain used for constant analysis. That is, to obtain the same precision as on the transformed program, one must analyze the source with a domain that combines constants and parity. This explains the gain in precision provided by the transformation. One can reverse this view, and consider that a source analysis with a product domain (constants $\times$ parity) is factored into a transformation based only on the constants domain, and analysis based only on parity. These intuitions are made precise in the rest of the paper.

3 Preliminaries

For convenience, we abstract from programming syntax and represent programs by their induced transition systems and program transformations as transition system transformations. Representing program transformations semantically is uncommon, but was also followed in, e.g. [9], for similar reasons. We represent static analyses formally using the framework of abstract interpretation [7].

3.1 On Notation

We follow the notation of Dijkstra and Scholten from [12] for algebraic calculations. Sets are identified with predicates, Boolean operators stand for set operations, e.g., $A \cap B$ is written as $A \ \wedge \ B$, and the "boxed" form $[\varphi]$ represents that the predicate φ is true (equivalently, that the set φ is universal). Thus, $[X \ \rightarrow \ Y]$ expresses that set X is a subset of set Y. A calculational proof is a sequence of proof steps, each one being a weakening (indicated by $\rightarrow$) or an equivalence (indicated by $\equiv$). A proof step establishing $[f \ \rightarrow \ g]$, say, is displayed as follows.

$$\begin{array}{ll} & f \\ \rightarrow & g \qquad\qquad \{\, \text{hint why } f \text{ is stronger than } g \,\} \end{array}$$

3.2 Programs and Program Transformations

Transition Systems. A program is represented by its induced transition system [3]. A transition system is defined by a tuple (S, I, Σ, δ), where S is a set of *states*, I is a non-empty subset of *initial* states, Σ is a set of *actions*, and $\delta \subseteq S \times \Sigma \times S$ is the *transition relation*. For a triple $(s, a, s') \in \delta$, we say that s' is a *successor to s on a*. We use the notation $\delta(Y)$, for a set of states Y, to denote the successors of Y by δ, i.e., $s' \in \delta(Y)$ if, and only if, there is a state s in Y such that $\delta(s, a, s')$ holds for some action label a. An *execution* of the transition system from state s is a sequence of alternating states and actions, of the form $s = s_0, a_0, s_1, a_1, \ldots$, where for each i, (s_i, a_i, s_{i+1}) is a transition in δ. Its trace is the sequence $a_0, a_1, \ldots$. A *computation* is an execution from some initial state. A state is *reachable* if it appears along some computation. The *language* of a transition system T, denoted as $\mathcal{L}(T)$, is the set of traces of its finite and infinite computations.

Program Transformations and Correctness. A program transformation, viewed semantically, is a function mapping one transition system to another with the same action set. A transformation from S to T is correct if $\mathcal{L}(T) \subseteq \mathcal{L}(S)$. I.e., for every computation x of T, there is a computation y of S such that x and y have the same[3] trace.

[3] To allow stuttering, one may define a subset of actions to be observable, and let the trace of an execution be the sequence of observable actions on it.

Simulation and Bisimulation. A relation R connecting states of transition system T to states of transition system S is a *simulation* (of T by S) – also called a "refinement mapping" – if:

- For states t, s such that $(t, s) \in R$, for every action a, and every successor t' of t on a, there is a successor s' of s on a such that $(t', s') \in R$, and
- For every initial state t of T, there is an initial state s of S where $(t, s) \in R$.

Relation R is a *bisimulation* if both R and its inverse relation, R^{-1} are simulations. Establishing (bi)simulation is a standard proof technique for showing correctness, thanks to the following standard results.

Theorem 1. *Let S, T be transition systems, and let R be a relation connecting states of T to those of S. If R is a simulation, then $\mathcal{L}(T) \subseteq \mathcal{L}(S)$. If R is a bisimulation, then $\mathcal{L}(T) = \mathcal{L}(S)$.*

Relational Operators. For any relation R on any domain, the modal operators pre_R and post_R, are defined as follows. For any set S,

$$u \in \mathsf{pre}_R(S) = (\exists v : uRv \ \wedge \ v \in S) \qquad \mathsf{post}_R(S) = \mathsf{pre}_{R^{-1}}(S)$$

I.e., $\mathsf{pre}_R(S)$ is the pre-image of S under R; it is the set of all elements that are related by R to some element of S. Likewise, $\mathsf{post}_R(S)$ is the image of S by R; it consists of all elements that are connected to elements in S by R.

A set of states, X, is an inductive invariant of a transition system (S, I, Σ, δ) if it includes all initial states, i.e., $[I \ \rightarrow \ X]$, and is closed under the transition relation, i.e., $[\mathsf{post}_\delta(X) \ \rightarrow \ X]$. Invariants of S can be transformed into invariants of T through a simulation B, as follows.

Theorem 2. *(cf. [27]) Let R be a simulation from T to S. For any inductive invariant φ of S, the set $\mathsf{pre}_R(\varphi)$ is an inductive invariant of T.*

Transformation Witnesses. We assume that every semantic-preserving program transformation has an associated bisimulation relation which acts as a "witness" (i.e., a proof) for correctness. Common compiler transformations, e.g., constant propagation, dead store removal, static single assignment (SSA) conversion and loop invariant code motion have simple witnesses [2, 28]. Abadi and Lamport's result [1] shows that every language inclusion has a simulation witness (after adding auxiliary history and prophecy information).

3.3 Static Program Analysis

We briefly review standard notions. A static program analysis is usually defined by specifying (1) a concrete domain as a partial order $(C, \leq_C)$, (2) an abstract domain as a partial order $(A, \leq_A)$, and (3) a pair of functions (α, γ), called a Galois connection, between the two domains where $[\alpha(c) \leq_A a \ \equiv \ c \leq_C \gamma(a)]$. The concrete semantics of a program is defined as the least fixpoint of a *transformer* $\tau : C \rightarrow C$. The Galois connection induces a transformer $\alpha \circ \tau \circ \gamma$

whose least fixpoint over A defines the most precise abstract semantics, which is an over-approximation of the concrete one [7].

In this work, the concrete domain consists of sets of states ordered by subset inclusion. As we combine aspects of static analysis with those of invariants and (bi)simulation, it is convenient to work entirely within the concrete domain instead of carrying around an abstract domain and a Galois connection. We use an equivalent formulation of abstract domains in terms of closure operators on the concrete domain. An operator cl is a (up-)closure if it is monotonic, i.e., $[X \rightarrow Y]$ implies that $[\mathsf{cl}(X) \rightarrow \mathsf{cl}(Y)]$; increasing, i.e., $[X \rightarrow \mathsf{cl}(X)]$; and idempotent, i.e., $[\mathsf{cl}(\mathsf{cl}(X)) \equiv \mathsf{cl}(X)]$. Given a Galois connection (α, γ), the operator $\gamma \circ \alpha$ is a closure, with closed sets corresponding to abstract elements.

The set of *reachable states* of a transition system (S, I, Σ, δ) is the least fixpoint of the concrete transformer $\delta^+(X) = X \vee \delta(X)$ that includes the initial states, I. Following [8], we write this as $\mathsf{lfp}\ (\delta^+, I)$. The general form $\mathsf{lfp}\ (f, a)$ denotes the least fixpoint of f above a, which exists if $a \leq f(a)$ for monotone f, cf. [5]. The reachable states can also be expressed as $\mathsf{lfp}\ ((\lambda X : I \vee \delta(X)), \emptyset)$. In the abstract setting, we look for closed sets as solutions. Thus, we construct $\mathsf{lfp}\ (\mathsf{cl} \circ \delta^+, I)$ or, equivalently, $\mathsf{lfp}\ ((\lambda X : \mathsf{cl}(I \vee \delta(X))), \emptyset)$.

Theorem 3. *$\mathsf{lfp}\ (\mathsf{cl} \circ \delta^+, I)$ is well defined. It is the least closed set that is an inductive invariant of the transition system.*

Proof. As both cl and δ^+ are increasing, $[I \rightarrow \mathsf{cl} \circ \delta^+(I)]$. Thus, the least fixpoint exists. Let $L = \mathsf{lfp}\ (\mathsf{cl} \circ \delta^+, I)$. Then $[I \rightarrow L]$ by definition of L. Moreover, $[\mathsf{cl}(\delta^+(L)) \equiv L]$ by the fixpoint property; hence, L is closed, and $[\delta(L) \rightarrow L]$. Thus, L is a closed inductive invariant.

To show the minimality of L, let Y be any closed set that is also an inductive invariant. From inductiveness, $[I \rightarrow Y]$ and $[\delta(Y) \rightarrow Y]$ holds. Hence, $[\delta^+(Y) \equiv Y]$; since Y is closed, $[\mathsf{cl}(\delta^+(Y)) \equiv Y]$ holds. Thus, Y includes I and is a fixpoint of $\mathsf{cl} \circ \delta^+$. As L is the least such set, $[L \rightarrow Y]$. □

More approximate closed invariants are provided by $\mathsf{lfp}\ (\eta, I)$, where η is monotone, $[I \rightarrow \eta(I)]$, and η maps to closed sets of cl. To be *sound*, $\mathsf{lfp}\ (\eta, I)$ must be a superset of the reachable states. That is guaranteed if $\eta(X)$ is a superset of $\delta^+(X)$ for all X. We say that such η are *adequate*.

One mechanism to achieve finite convergence of the fixpoint computation is widening [9, 6, 7]. Let $\mathbb{D}$ be an abstract domain with elements denoted by D. A widening operator is a function $\triangledown : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$ such that:

- $[D_2 \subseteq D_1 \triangledown D_2]$
- $[D_2 \subseteq D_1 \rightarrow D_1 \triangledown D_2 = D_1]$
- Let $D_0, D_1, \ldots$ be an increasing chain of abstract elements. Let $D'_0, D'_1, \ldots$ be a chain of elements such that $D_i \subseteq D'_i$ for every i. Then there exists $n \in \mathbb{N}$ such that $\forall k \geq n : D_k \triangledown D'_k = D_n$.

Given an adequate transformer η and a widening $\triangledown$, the new transformer $\eta_\triangledown(X) \triangleq X \triangledown \eta(X)$ is adequate and for every initial approximation $D \in \mathbb{D}$ the sequence $\langle \eta^i_\triangledown(D), i \in \mathbb{N} \rangle$ becomes stationary [6]. The least fixpoint of $\eta_\triangledown$ is an over-approximation, sacrificing precision for guaranteed eventual termination.

4 Relating Analyses Under Bisimulation

In this section, we formulate a framework for analyzing the effect of transformations on static analysis results. For the remainder of this paper, we assume a source program S, a transformed program T, and a bisimulation B between T and S semantically modeling a semantic-preserving program transformation. We further assume an abstract domain underlying the desired analysis in terms of closures cl_S and cl_T for the source and transformed program, respectively, and corresponding widening operators $\triangledown_S$ and $\triangledown_T$.

The above assumptions fit the setting in which program verification tools such as SMACK [4] and SeaHorn [17] operate. Programs S and T are available in practice as the mentioned tools anyhow run the transformations. Also, tools performing semantic-preserving transformations implicitly have all of the information necessary to generate the underlying bisimulation information [33, 28, 18]. Lastly, the assumed closure and widening operators are essentially program-specific lifts of corresponding operators defined over readily available program-agnostic domains such as intervals, octagons [25], polyhedra [11], etc.

4.1 Comparing Invariants of S and T

Let G denote the invariant on S computed with cl_S, and let H denote the invariant on T computed with cl_T. In order to compare the relative strengths of the two invariants, we have to transform them from one state space to the other, as the state spaces of S and T may, in general, be different. The bisimulation B is used to perform this transformation, using Theorem 2.

Informally, we would consider H to be stronger than G if, after transferring H from T to S via B^{-1}, the resulting invariant in S is stronger than G, i.e., if $[\mathsf{post}_B(H) \rightarrow G]$. By the symmetry of bisimulation, we should also require that the invariant obtained by transferring G in the other direction, from S to T, is weaker than H. I.e., we want $[H \rightarrow \mathsf{pre}_B(G)]$ to also hold.

Thus, we take the two conditions (a) $[\mathsf{post}_B(H) \rightarrow G]$ and (b) $[H \rightarrow \mathsf{pre}_B(G)]$ as the definition of the property "H is stronger than G". Condition (a) is equivalent to $[\mathsf{cl}_S \circ \mathsf{post}_B(H) \rightarrow G]$, a form that is used in the proofs below.

4.2 Induced Closure for S

Suppose that H is stronger than G. In order to explain the precision gain T exhibits compared to S (equivalently, the precision loss of S subject to T), we formulate a new abstract domain on S (via a new closure operator) such that an analysis on S with this operator produces an invariant that is at least as strong as the transferred invariant $\mathsf{post}_B(H)$.

A natural way to reflect the computation from T into S is as follows: given a subset X of the states of S, its closure is computed by mapping X to its image Y in T through the relation B^{-1}; forming Y', the closure of Y in T through cl_T; and finally, mapping Y' back to a set X' in S through B. The new operator $\mathsf{cl}^{B,H}$ is formulated using this intuition. It is defined as a least fixpoint, $(\lambda X.\, \mathsf{lfp}\ (g, X))$,

where the function g is given below. The key to g is the composition $\mathsf{post}_B \circ \mathsf{pre}_B$ (ignoring the intervening closures); this composition formalizes the intuition of moving from S to T and back again.

$$g(Z) \triangleq Z \ \vee \ \mathsf{post}_B \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B \circ \mathsf{cl}_S(Z)) \tag{1}$$

The function g is increasing by its first term and monotone as all operators are monotone. It follows from standard arguments that

Lemma 1. $\mathsf{cl}^{B,H} = (\lambda Z.\, \mathsf{lfp}\ (g, Z))$ *is a closure operator on* S.

4.3 Induced Static Analysis

We now turn to the invariants computed with static analysis using the new closure operator on S and the best abstract transformer, $\mathsf{cl}_S^{B,H} \circ \delta_S^+$. We show that the resulting inductive invariant is *at least as precise* as the invariant $\mathsf{post}_B(H)$ obtained by transferring the analysis result H from T to S.

Theorem 4. *Let* $G = \mathsf{lfp}\ (\mathsf{cl}_S \circ \delta_S^+, I_S)$ *be the result of the static analysis on* S. *Let* $H = \mathsf{lfp}\ (\eta_T, I_T)$ *be the result of a sound static analysis on* T *with closure* cl_T. *Let* $G^{B,H} = \mathsf{lfp}\ (\mathsf{cl}_S^{B,H} \circ \delta_S^+, I_S)$ *be the invariant computed on* S *with the newly defined closure operator. If* H *is stronger than* G, *then* $[G^{B,H} \ \rightarrow \ \mathsf{post}_B(H)]$.

Proof. We prove the claim by showing that $\mathsf{post}_B(H)$ is a superset of I_S, and a pre-fixpoint of the function $\mathsf{cl}_S^{B,H} \circ \delta_S^+$.

As $\mathsf{post}_B(H)$ is an inductive invariant of S, it includes the initial states; hence, $[I_S \ \rightarrow \ \mathsf{post}_B(H)]$.

Next, we establish that $[\mathsf{cl}_S^{B,H} \circ \delta_S^+(\mathsf{post}_B(H)) \ \rightarrow \ \mathsf{post}_B(H)]$. As $\mathsf{post}_B(H)$ is inductive for S, this is equivalent to $[\mathsf{cl}_S^{B,H}(\mathsf{post}_B(H)) \ \rightarrow \ \mathsf{post}_B(H)]$, which holds if $\mathsf{post}_B(H)$ is a pre-fixpoint of the function g used to define $\mathsf{cl}_S^{B,H}$ (in Equation 1). By the form of g, we only need to consider its second term:

$$\begin{aligned}
&\mathsf{post}_B \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B \circ \mathsf{cl}_S(\mathsf{post}_B(H))) && \\
\rightarrow\ &\mathsf{post}_B \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(G)) && \{\, H \text{ is stronger than } G, \text{ condition (a)} \,\} \\
\equiv\ &\mathsf{post}_B \circ \mathsf{cl}_T(H) && \{\, H \text{ is stronger than } G, \text{ condition (b)} \,\} \\
\equiv\ &\mathsf{post}_B(H) && \{\, H \text{ is closed under } \mathsf{cl}_T \text{ by property of } \eta_T \,\}
\end{aligned}$$

□

Discussion This theorem shows how to construct a new domain on S that matches (or improves) the gain of precision obtained by transforming S to T. The new domain is constructed from the bisimulation B, the abstract domain of T, as well as its invariant H. The structure of $\mathsf{cl}_S^{B,H}$ shows how a transformation, in the form of its bisimulation relation, influences the precision of an analysis. This is a somewhat indirect demonstration: an intriguing open question is whether it is possible to determine directly from B and cl_T if precision is lost or gained.

As bisimulation is symmetric, a loss of precision in a transformation from S to T is a gain of precision when viewed from T towards S. Therefore, if precision is lost, this theorem can be applied to construct a new domain in T which recovers the greater precision of analysis in S – for instance, in the introductory 3-address code translation example.

The reason why the new analysis in S can be strictly more precise than the back-propagated invariant of T is that some transformations can introduce complexity in the transformed program. For instance, envision a transformation that replaces a constant in the program with, say, a binary expression that provably always evaluates to that constant. The induced analysis reaps the benefits of the bisimulation and the simplicity of the original program for such transformations.

5 Practicality Extensions

If verification tools were to implement the induced analysis, the only new operation they are required to implement is refinement of an abstract domain element with the bisimulation information $\mathsf{post}_B \circ \mathsf{pre}_B$ (modulo intervening closures). This operation is often feasible as the bisimulation for many common transformations is essentially a conjunction of equalities between the variables and expressions of T and S at corresponding points in the two programs [28, 22, 18]. However, the induced analysis has several shortcomings that hinder its usability.

First, the results of Section 4.3 hold for the best transformer δ_S^+ of the source program, which might not be easily computable. In fact, the abstract transformer for the source program might not be even available in some cases. Several verification frameworks translate programs written in higher-level source languages to a bytecode representation to support multitude of different programming languages [22, 13]. In that case, only the transformer for program T is available. Second, the analysis operates and produces results over the new (induced) abstract domain. The widening operators for this domain are not immediate. Third, the closure operator $\mathsf{cl}_S^{B,H}$ is defined as a fixpoint which might be expensive to compute in practice. Lastly, the new domain relies on the precomputed invariant H of T. We now address these practical limitations of the induced analysis.

5.1 Bisimulating Analysis

The results in Section 4.3 show that the least fixpoint of the best transformer induced by $\mathsf{cl}_S^{B,H}$ is at least as strong as the back-propagated invariant from T. In this part, we exhibit a simpler *bisimulating* analysis with a similar property. In essence, the new transformer η_S uses the bisimulation to jump to the transformed program, makes an analysis step there, and then comes back to the source program.

$$\eta_S(X) \triangleq \mathsf{cl}_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(X)) \tag{2}$$

An important property of η_S is that, although it is defined over the source program, the analysis uses only the provided adequate transformer η_T for the

transformed program. Hence, it does not depend on the source transformer that, as pointed our earlier, sometimes might not be even available in practice. We also remind the reader that this is the analysis we used in our example of Section 2. Furthermore, this analysis avoids the fixpoint calculation in the closure operator $\mathsf{cl}_S^{B,H}$.

The new analysis is step-wise more precise than the provided analysis on the transformed program. That is, in each iteration the bisimulating analysis does not lose precision. The following results formalize this intuition.

Lemma 2. *The new bisimulating analysis operator η_S is sound for S.*

Proof. The operator η_S is monotonic, as all operations in its defining expression are monotonic. We show that $\mathsf{lfp}\ (\eta_S, I_S)$ is well-defined and that the result over-approximates the reachable states.

We first establish that $[I_S \ \rightarrow\ \eta_S(I_S)]$, to ensure that $\mathsf{lfp}\ (\eta_S, I_S)$ is defined. As H is an invariant of T and B is a bisimulation, I_T is a subset of $(H \cap \mathsf{pre}_B(I_S))$. By adequacy of η_T, it follows that I_T is a subset of $\eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(I_S))$. As B is a bisimulation, I_S is a subset of $\mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(I_S))$, and therefore of $\eta_S(I_S)$.

Next, we establish that $[R^{k+1} \ \rightarrow\ \eta_S(R^k)]$ for all k, which establishes that $\mathsf{lfp}\ (\eta_S, I_S)$ includes all reachable states. Consider a state s' in $R^{k+1} = \delta_S^+(R^k)$. There are two cases.

(i) s' is in R^k. Then s' is also in $\mathsf{post}_B(H)$, as that is an invariant of S. Hence, there is a state t' in T such that $t'Bs'$ holds, and $t' \in H$. Therefore, t' is in $H \cap \mathsf{pre}_B(R^k)$ and thus in the closure of that set under cl_T. By adequacy of η_T, the state t' is in $\eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(R^k))$. As t' is related to s' by B, s' is in $\mathsf{cl}_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(R^k))$, i.e., s' is in $\eta_S(R^k)$.

(ii) s' is a successor of a state s in R^k. As s is in $\mathsf{post}_B(H)$, there is a state t of T such that tBs and t is in $H \cap \mathsf{pre}_B(R^k)$. As B is a simulation relation, this state has a successor, t', such that $t'Bs'$ holds. By adequacy of η_T, t' is in $\eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(R^k))$. As t' is related to s' by B, s' is in $\mathsf{cl}_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(R^k))$, i.e., s' is in $\eta_S(R^k)$. □

We now show that the result of analyzing program S with η_S is as precise as the transferred invariant $\mathsf{post}_B(H)$, when expressed as a closed set using cl_S. Note that as H is presumed to be stronger than G, by condition (a) of that definition, $\mathsf{cl}_S \circ \mathsf{post}_B(H)$ is stronger than G.

Theorem 5. *Let $G = \mathsf{lfp}\ (\mathsf{cl}_S \circ \delta_S^+, I_S)$ be the result of the original analysis on S, and $\hat{G} = \mathsf{lfp}\ (\eta_S, I_S)$ be the result of the analysis on S using the new η_S. Let $H = \mathsf{lfp}\ (\eta_T, I_T)$ be the result of the static analysis on T using an adequate transformer η_T. If H is stronger than G, then $[\hat{G} \ \rightarrow\ \mathsf{cl}_S \circ \mathsf{post}_B(H)]$.*

Proof. We prove this by showing that $\mathsf{cl}_S \circ \mathsf{post}_B(H)$ is a pre-fixpoint of η_S and that it includes I_S. As $\mathsf{post}_B(H)$ is an invariant of S, we have that $[I_S \ \rightarrow\ \mathsf{post}_B(H)]$. Hence, $[I_S \ \rightarrow\ \mathsf{cl}_S \circ \mathsf{post}_B(H)]$. Now consider the pre-fixpoint claim.

$$
\begin{aligned}
& \eta_S(\mathsf{cl}_S \circ \mathsf{post}_B(H)) \\
\equiv\ & cl_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(\mathsf{cl}_S(\mathsf{post}_B(H)))) && \{\text{ definition }\} \\
\rightarrow\ & cl_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H \cap \mathsf{pre}_B(G)) && \{\ H \text{ is stronger than } G, \text{ condition (a)}\ \} \\
\rightarrow\ & cl_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T(H) && \{\ H \text{ is stronger than } G, \text{ condition (b)}\ \} \\
\equiv\ & cl_S \circ \mathsf{post}_B(H) && \{\ H \text{ is closed under } \mathsf{cl}_T \text{ and is a fixpoint of } \eta_T\ \}
\end{aligned}
$$

□

Widenings By relying on the abstract domain and transformer of T, we can also use the widening operator for T to ensure finite convergence of the bisimulating analysis. We assume the abstract transformer on T is $\eta_T^{\triangledown}(Y) \triangleq Y \ \triangledown_T \ \eta_T(Y)$, where η_T is an adequate monotone function as usual. We therefore use η_T to define η_S, as shown above, and then define the widened bisimulating transformer as $\eta_S^{\triangledown}(X) \triangleq X \ \triangledown_S \ \eta_S(X)$. The analysis based on this transformer is guaranteed to converge in a finite number of steps but it may be less precise than the propagated invariant computed by $\eta_T^{\triangledown}$. The reason for this is that, although η_S is more precise than the back-propagated η_T, the widening operators are not necessarily monotone [6]. We leave for future work the investigation of the actual ramifications of this imprecision in practice as well as the construction of more precise bisimulating widening operators.

5.2 Optimizing Domain Calculations under Bisimulation Closure

The formulations of the new closure operator (Section 4.3) and the bisimulating analysis (Section 5.1) rely on the invariant H on T. We show below that this dependence can be removed if H is known to be closed under bisimulation within T – i.e., if state s is in H, so is any other state s' that is bisimular to s. This is guaranteed if all closed sets in T are closed under bisimulation, as H is one such. Intuitively, bisimulation-closure asserts that indistinguishable concrete states do not negatively effect the precision of an abstract domain. Formally, we define

Assumption 1 (Bisimulation closure) $[\mathsf{pre}_B \circ \mathsf{post}_B(Y) \ \rightarrow \ Y]$ *holds for all closed sets* Y *of* cl_T.

Assuming bisimulation-closure, the definitions can be simplified by eliminating H, as shown below, while retaining the properties shown previously.

$\mathsf{cl}^{B,H}(X) \triangleq \mathsf{lfp}\ (g, X))$, where $g(Z) \triangleq Z \ \vee \ \mathsf{post}_B \circ \mathsf{cl}_T \circ \mathsf{pre}_B \circ \mathsf{cl}_S(Z)$ and

$\eta_S(X) \triangleq \mathsf{cl}_S \circ \mathsf{post}_B \circ \eta_T \circ \mathsf{cl}_T \circ \mathsf{pre}_B(X)$

Bisimulation-closure holds if B has a functional form, as shown below. Several common program optimizations have functional bisimulation relations. Examples include constant propagation, dead-code removal, and loop unrolling. Even transformations that reorder execution, such as loop inverse, induce a bisimulation relation that maps every source state to a single target state.

Lemma 3. *If B is functional, i.e., $[tBs \wedge t'Bs \rightarrow t = t']$, then bisimulation-closure holds.*

Proof. Consider any subset Y of T. State t' is in $\mathsf{pre}_B \circ \mathsf{post}_B(Y)$ iff there are states s in S and t in Y such that tBs and $t'Bs$. As B is functional, $t = t'$; thus, t' is in Y. □

Transformations that can potentially invalidate the bisimulation closure are those that break-up the computation. For instance, 3-address code translation will break a single source statement into several target ones. Consider a source statement `assume (x - y ≤ 7)` and its 3-address translation `t1' := x'-y'; t2' := t1' ≤ 7; assume t2'`. A source state just before the original statement maps to several target states corresponding to the intermediate computation of the starting two statements in the target program. However, these statements only refine the relationship between the variables. That is, at the beginning of the target program no relationship between target variables `t1', x', y'` and `t2'` is known. Each consecutive statement does not invalidate existing relationships between other variables, yet it only refines the ones between the above mentioned target variables, satisfying the bisimulation closure assumption.

5.3 Counteracting Precision Loss in 3-address Code Transformation

We now exemplify how verification tools can use the new bisimulating analysis to counteract precision loss due to a transformation. Consider a relational static analysis that computes bounds on the difference between the values of pairs of variables. In other words, an abstract state is the conjunction of difference-bounds constraints of the form $x - y \leq c$ and $\pm x \leq c$, where x and y are program variables and c is an integer or real constant [24, 23]. For the example of three-address code transformation from [22], shown in Figure 2, the analysis will infer that $(x - y) \leq 7$ holds at the end of the source program (on the left). The same analysis, however, fails to infer any useful relation between `x` and `y` on the transformed program (on the right). As explained in [22], for an accurate result, it is necessary to track a relationship between *three* variables (e.g., $t'_1 = x' - y'$), which cannot be done precisely in the given analysis domain.

Bisimulation relation. The bisimulation is symbolically illustrated in Figure 2 using the horizontal lines and the attached predicates defined over program variables. The relation also contains "history" information connecting t'_1 and t'_2 to x' and y'. The bisimulation allows stuttering steps on T. The transformation engine can generate the bisimulation relation while performing the actual transformation [20, 21]. That is, the information about equality of live expressions can be extracted directly from the generated 3-address code.

Bisimulating analysis. Initially, the invariant approximant for the source program maps the top and bottom difference-bounds abstract element to the first and last source location, respectively. This approximant is then transformed

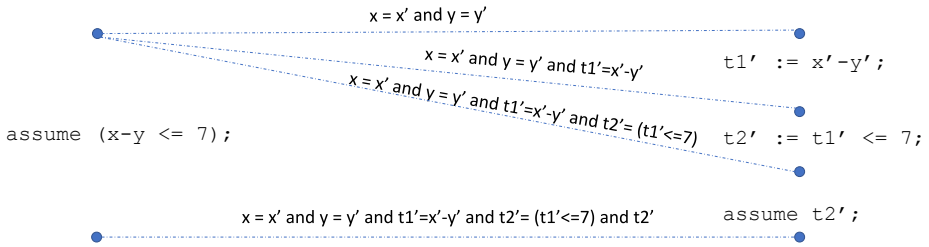

Fig. 2: A 3-address code translation (from [22]) and accompanying bisimulation relation

into an approximant for the transformed program using $\mathsf{cl}_T \circ \mathsf{pre}_B$. The resulting approximant assigns the top abstract element to the first three locations of T since the corresponding bisimulation information does not imply any useful difference-bounds of variables in T. The last location of T is assigned the bottom element as that is the element being forward propagated from S. As explained earlier, applying η_T results in the top element being assigned to every location of T. However, the resulting approximant can now be refined using the bisimulation information when propagating the information back to S using $\mathsf{cl}_S \circ \mathsf{post}_B$. That is, the information on the last horizontal line of Figure 2 implies $x - y \leq 7$.

Although the analysis technically works over the source program, the inference step is in fact made on the transformed program. The resulting invariant can again be converted into an invariant for T using the result of Theorem 2. The new operation verification tools are required to implement is refinement/strengthening of abstract elements with the bisimulation information. One possible way to implement this operation is to rely on known techniques for strengthening branch results with guard information when analyzing guard statements [7].

Precision. Logozzo and Fähndrich [22] show how the precision lost by the transformation can be restored if information about available program expressions, and equalities between them, are preserved at each location in T. But this is precisely the information provided by the bisimulation relation. To see this, one has to switch the roles of S and T, which is possible as B^{-1} is a bisimulation from S to T. The domain of the induced bisimulating analysis on T combines information about program expressions, such as the definition of `t1`, with the original difference domain. One can therefore derive the analysis of [22] systematically from the bisimulating analysis definition, and view their specific implementation as a particular form of the bisimulating analysis.

6 Transformations as Static Analyses

Consider, once again, the transformation shown in Figure 1. Parity analysis is less precise for variable `z` in the source program as it does not observe the actual values of variables `y` and `x`. That can be done with a second domain to track constant values, combining its information with the parity analysis to obtain a precise parity value for `z`. This is the role of the standard *reduced product* construction of [8]. Applied to the product domain $C \times D$ of domains C and D, a reduction operation transforms an abstract value (c, d) – where $c \in C$ and $d \in D$ – into a more precise abstract value (c', d') with the same concretization. Reduction is carried out by using the information in c to refine d to d', and the information in d to refine c to c'. In our example, the information flow is one-way: the constants domain is used to refine the parity result. The program transformation shown in Figure 1 is also based on the constants domain. One might conjecture from this that the transformation plays a role analogous to a one-way reduced product.

In this section, we establish a precise form of this conjecture. We show that an analysis based on a one-way reduced product of domains C with D, where information flows only from C to D, can be "factored" into a program transformation based on an analysis of the source program with domain C, followed by an analysis of the transformed program with D, obtaining results on D that are at least as precise as the original. Thus, an analysis expressed as a chain of one-way products of $C_1, C_2, C_3, \ldots, C_n = D$ where C_i is used to refine C_{i+1}, can be broken down into a chain of transformations, one for each C_i, ending with a program that is analyzed with D. (For a similar reduction over domains but without program transformations, see [15].)

The (simple) transformation eliminates the need to compute with a reduction operator, which can be a significant advantage in practice. It also shows that new program transformations may be designed solely for the purpose of simplifying program analysis, in addition to the use of standard compiler transformations, which are designed primarily to improve run-time performance.

One-way Reduced Product Consider abstract domains C and D, specified by their closure functions, cl_C and cl_D. The Cartesian product of C and D is the domain formed by the closure function given by $\mathsf{cl}(X) = \mathsf{cl}_C(X) \cap \mathsf{cl}_D(X)$. For convenience, elements in this domain may be represented by a pair of sets (X, Y), where X is closed for C and Y is closed for D, with the interpretation that (X, Y) denotes the set $X \cap Y$.

A *one-way reduction* function ρ maps a pair (X, Y) of the form above to a set Y' that is closed for D, such that the interpretation of (X, Y) and (X, Y') is the same. (A two-way reduced product, in addition, reduces X to some X'.) The best one-way reduction of (X, Y) is given by $\mathsf{cl}_D(X \cap Y)$. This shows clearly that the reduction transfers information from the X component to the Y component, producing $Y' = \mathsf{cl}_D(X \cap Y)$ which, by its definition, is at least as precise as Y.

Fixpoint Analysis The standard construction of the best abstract transformer adds reduction as the final step. I.e., to obtain the best abstract representation from a starting point (X, Y), one computes $X' = \mathsf{cl}_C \circ \delta_S^+(X \cap Y)$ and $Y' = \mathsf{cl}_D \circ \delta_S^+(X \cap Y)$ and reduces (X', Y') to $(X', \rho(X', Y'))$. We relax this construction using the common simplification which applies the transformers for C and D individually, i.e., letting $X' = \mathsf{cl}_C \circ \delta_S^+(X)$ and $Y' = \mathsf{cl}_D \circ \delta_S^+(Y)$.

Theorem 6 (Factoring). *Consider the least fixpoint analysis of program S with a one-way reduced product of domains C and D and the relaxed best transformer. Equally or even more precise result can be obtained by transforming S to a program T, based on the analysis of S over domain C, followed by analysis of T over domain D.*

Proof. The proof outline is as follows. We first establish that the least fixpoint analysis can be sequentialized. We use the fixpoint over C to define the transformation from S to T, and prove that analysis of T over D produces the same result as the original fixpoint.

Let $(\bar{c}, \bar{d})$ be the least fixpoint of the relaxed transformer defined earlier that includes the initial states of S. This is a simultaneous fixpoint definition over the vector (X, Y).

We simplify this to a different, but equivalent form, starting from the empty set instead of from I_S. Let functions f_C and g_D be defined on a pair (X, Y) by $f_C(X) = \mathsf{cl}_C(I_S \vee \delta_S(X))$ and $g_D(X, Y) = \rho(f_C(X), \mathsf{cl}_D(I_S \vee \delta_S(Y)))$. Then the original fixpoint can be re-expressed as

$$(\bar{c}, \bar{d}) = \mathsf{lfp}\ ((\lambda(X, Y).\, (f_C(X), g_D(X, Y))), (\emptyset, \emptyset))$$

By a well-known result from Bekič (sometimes called the Scott-Bekič theorem), the fixpoint value for domain D can also be obtained with the "flattened" nested fixpoint defined below, where the outer fixpoint is over the closed sets Y of D, and the inner fixpoint over the closed sets X of C.

$$\text{let } \bar{d} = \mathsf{lfp}\ ((\lambda Y.\, g_D(\mathsf{lfp}\ ((\lambda X : f_C(X)), \emptyset), Y)), \emptyset)$$

As f_C is independent of Y, the inner fixpoint can be extracted to form the equivalent, simpler definition:

$$\begin{aligned} &\text{let } \bar{c} = \mathsf{lfp}\ ((\lambda X.\, f_C(X)), \emptyset) \\ &\text{let } \bar{d} = \mathsf{lfp}\ ((\lambda Y.\, g_D(\bar{c}, Y)), \emptyset) \end{aligned}$$

That is, the computation of the original fixpoint can be sequentialized, by first computing $\bar{c}$, and only then computing $\bar{d}$ in terms of $\bar{c}$. By Theorem 3, $\bar{c}$ is an inductive invariant of S.

We now use the value $\bar{c}$ to define a simple transformation from S to T. The program T has the same state space and the same set of initial states as S, but its transition relation is a restriction of that of S, defined by $[\delta_T(t, t') \equiv \bar{c}(t) \wedge \delta_S(t, t')]$. I.e., transitions are allowed only from states satisfying $\bar{c}$. As $\bar{c}$ is

inductive for δ_S, the expression for $\delta_T(t,t')$ is equivalent to $\bar{c}(t) \wedge \delta_S(t,t') \wedge \bar{c}(t')$. Hence, for a set Y of states, $[\delta_T(Y) \equiv \bar{c} \wedge \delta_S(Y \cap \bar{c})]$.

Define the relation B from T to S by $B(t,s) \equiv (t = s) \wedge \bar{c}(s)$. The fact that $\bar{c}$ is an inductive invariant of S helps establish that B is a bisimulation, we omit the simple proof.

The standard analysis with D on T results in $d^\sharp = \mathsf{lfp}\ ((\lambda Y.\, \mathsf{cl}_D(I_T \vee \delta_T(Y))), \emptyset)$. We show that this is at least as precise as $\bar{d}$, i.e., $[d^\sharp \rightarrow \bar{d}]$. This follows if $\bar{d}$ is a pre-fixpoint of the function used to define $d^\sharp$.

$$
\begin{array}{ll}
& \mathsf{cl}_D(I_T \vee \delta_T(\bar{d})) \\
\equiv & \mathsf{cl}_D(I_S \vee \delta_T(\bar{d})) & \{\text{ as } [I_T \equiv I_S]\ \} \\
\equiv & \mathsf{cl}_D(I_S \vee (\bar{c} \wedge \delta_S(\bar{d} \cap \bar{c}))) & \{\text{ by the relationship between } \delta_T \text{ and } \delta_S\ \} \\
\equiv & \mathsf{cl}_D(\bar{c} \wedge (I_S \vee \delta_S(\bar{d} \cap \bar{c}))) & \{\ [I_S \rightarrow \bar{c}] \text{ by inductiveness of } \bar{c} \text{ for } S\ \} \\
\rightarrow & \mathsf{cl}_D(\bar{c} \wedge \mathsf{cl}_D(I_S \vee \delta_S(\bar{d}))) & \{\text{ monotonicity}\ \} \\
\rightarrow & \rho(\bar{c}, \mathsf{cl}_D(I_S \vee \delta_S(\bar{d}))) & \{\text{ by definition of the best reduction}\ \} \\
\equiv & g_D(\bar{c}, \bar{d}) & \{\text{ by definition of } g_D\ \} \\
\equiv & \bar{d} & \{\text{ by fixpoint}\ \}
\end{array}
$$

□

From the careful examination of the above proof, it becomes clear that the transformation plays the role of the one-way reduction; as noted, the result obtained on the transformed program may even be stronger than that obtained by the one-way reduced product.

7 Related Work and Conclusion

In this work, we introduced a formal account of the impact program transformations can have on static analyses. By modeling transformations semantically using bisimulations and static analyses using abstract interpretation, we show how the improved/decreased precision of an analysis on the transformed program can be explained in terms of the bisimulation. We assemble the bisimulation and a given abstract domain to form a new abstract domain. The newly constructed domain induces an analysis on the source program that is more precise than the given analysis for the transformed program. We also present a weaker but more practical *bisimulating* analysis that utilizes information already present in verification frameworks, allowing the transfer of theoretical results almost directly to practice. We also show, in the opposite direction, how 1-way reduced product static analyses can be broken into a transformation followed by a simpler analysis. Our framework thus provides a formal understanding and theoretical machinery for a more systematic design of program analysis tools that combine program transformations and static analyses. We now discuss related work.

The work most closely related to ours is the one by Logozzo and Fähndrich [22]. The authors exemplify how 3-address code transformation can introduce imprecision for static analyses working over relational abstract domains. They also

show how the lost precision can be recovered by additionally tracking available expressions, a technique introduced by Miné [26]. We already overviewed the mentioned imprecision phenomena and the recovering technique in Section 5.1. Our work is a substantial generalization. The framework supports any transformation whose correctness can be witnessed by a common general class of bisimulations. Furthermore, there is a general technique for recovering from precision, which specializes to the use of symbolic expressions in their setting. Our work also paves the way for implementing static analyses using transformations.

Cousot and Cousot introduce a general and language-independent framework for designing program transformations [9]. By adopting the view that syntax is an abstraction of semantics, the authors use abstract interpretation to formalize and argue the idea that syntactic transformations are an abstraction of possibly incomputable semantic transformations. Their formalization allows for a more systematic design of syntactic transformations and simpler arguments of their correctness. Our work, on the other hand, is concerned with formal understanding of how program transformations affect static analyses, how the negative effects can be remedied, and how to design static analyses using program transformations. The common theme of the two papers is the semantic view of program transformations. As their work shows, syntactic transformations overapproximate the semantic ones; we use bisimulations to recover the loss of information stemming from the (proper) overapproximation.

Ranzato and Tapparo show in [30, 32, 29] how strong preservation in abstract model checking, witnessed by a bisimulation, can be characterized and generalized by the notion of completeness in abstract interpretation [16, 8]. In effect, the authors show how bisimulations are a particular case of abstract interpretation. As a consequence, abstract models can be refined using domain refinement techniques of abstract interpretation in order to achieve preservation of properties from the concrete model [31]. This body of work and our paper are related by using bisimulations in the context of abstract interpretation, specifically domain refinement [15]. Cousot et al. devise an abstract interpretation framework for inferring invariants over arbitrary abstract domains for refactored code fragments [10]. Focusing on the method extraction refactoring, the authors show how to reuse the invariants computed for the original program to infer the most general correct pre- and post-conditions for the extracted method that are compatible with the method use in the program and do not violate any assertions of the method body. Our work focuses on transformations that can be modeled semantically using bisimulations and is concerned with remedying potential precision loss caused by transformations. Their work has the objective of inferring *good* annotations for the refactored piece of code by utilizing the information provided by the prior analysis of the original program. Fedyukovich et al. present techniques that infer simulation relations that in turn allow transfer of safe inductive invariants from (an abstraction) of a source program P to its arbitrarily modified version Q [14]. Our work assumes a bisimulation relation but is concerned with designing new abstract domains that capture how semantic-preserving program transformations affect static analyses.

SeaHorn is a fully automated framework for verifying safety properties of software [17]. Built on top of LLVM [19], the framework uses sophisticated SMT-based model checking techniques together with abstract interpretation to perform inter-procedural static analysis. As a preprocessing step, SeaHorn performs several known program transformations, such as static single assignment (SSA), function inlining, dead-code elimination, etc. This preprocessing step, as reported, is introduced to simplify the verification task. SMACK is a verification toolchain which is also based on LLVM [4]. As a pre-processing step, SMACK runs common program optimizations provided by LLVM since they, as reported, *improve the performance and accuracy of verification* [4].

Acknowledgements. This work was supported, in part, by NSF grant CCF-1563393 from the National Science Foundation. We would like to thank Patrick Cousot, Thomas Wies, and Siddharth Krishna for helpful discussions.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.
3. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
4. Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamaric, and Michael Emmi. SMACK software verification toolchain. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 589–592, 2016.
5. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.
6. Patrick Cousot. Abstracting induction by extrapolation and interpolation. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 19–42, 2015.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.
8. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282, 1979.
9. Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 178–190. ACM, 2002.

10. Patrick Cousot, Radhia Cousot, Francesco Logozzo, and Michael Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 213–232, 2012.
11. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.
12. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.
13. Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, pages 10–30, 2010.
14. Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Property directed equivalence via abstract simulation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 433–453, 2016.
15. Roberto Giacobazzi and Francesco Ranzato. Refining and compressing abstract domains. In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, pages 771–781, 1997.
16. Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
17. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.
18. Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: verified credible compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 631–645, 2018.
19. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.
20. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 42–54, 2006.
21. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
22. Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, pages 197–212, 2008.

23. Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings*, pages 155–172, 2001.
24. Antoine Miné. A few graph-based relational numerical abstract domains. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, pages 117–132, 2002.
25. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
26. Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 348–363, 2006.
27. Kedar S. Namjoshi. Lifting temporal proofs through abstractions. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, volume 2575 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2003.
28. Kedar S. Namjoshi and Lenore D. Zuck. Witnessing program transformations. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 304–323, 2013.
29. Francesco Ranzato and Francesco Tapparo. Making abstract model checking strongly preserving. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, pages 411–427, 2002.
30. Francesco Ranzato and Francesco Tapparo. Strong preservation as completeness in abstract interpretation. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 18–32, 2004.
31. Francesco Ranzato and Francesco Tapparo. An abstract interpretation-based refinement algorithm for strong preservation. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 140–156, 2005.
32. Francesco Ranzato and Francesco Tapparo. Generalized strong preservation by abstract interpretation. *J. Log. Comput.*, 17(1):157–197, 2007.
33. Martin Rinard. Credible compilation. Technical report, In Proceedings of CC 2001: International Conference on Compiler Construction, 1999.