Combining Optimizations, Combining Theories

Todd L. Veldhuizen and Jeremy G. Siek

Indiana University, Bloomington, Indiana 47401 USA
`tveldhui@acm.org` `jsiek@osl.iu.edu`

Abstract. We consider the problem of how best to combine optimizations in imperative compilers. It is known that combined optimizations (or "super-analyses") can be strictly better than iterating separate improvement passes. We propose an explanation of why this is so by drawing connections between program analysis and the algebraic and coalgebraic views of programs and processes. We argue that "optimistic" analyses decide coinductively-defined relations and are based on bisimilarity. We relate combining program improvements to the problem of deciding combinations of theories. Iterating program improvements is similar to the Nelson-Oppen method of deciding combined theories: in Nelson-Oppen decision procedures communicate equalities, and iterated improvement passes implicitly communicate equalities via term replacements. To decide combined theories of bisimilarity, some "co-Nelson-Oppen" procedure is needed that propagates *inequalities* amongst decision procedures. Hence, iterating optimistic analyses fails to be effective because inequalities cannot be communicated by semantics-preserving rewrites. Superanalysis is conjectured to overcome this failing by behaving like a "co-Nelson-Oppen" decision procedure.

1 Introduction

The literature on compiler optimizations is vast, but comparatively little of it addresses the problem of how to effectively combine optimizations. A key problem is that no one sequence of improvement passes works well for all programs: the *phase ordering problem*. It is known that in the presence of loops and recursion, simultaneous improvements can be strictly better than iterating improvement passes. Although examples have been reported, there is no explanation of why this is so. Three important open questions are: (1) Why should combined improvements be better than separate, iterated improvement passes? (2) Which improvements are more effective if combined? (3) Are there effective methods to *automatically* combine analyses without manually specifying their interaction? The answers to these questions impact the structure of optimizing compilers in a fundamental way.

We investigate these questions by relating two flavours of imperative program improvement often called *pessimism* and *optimism* (e.g. [WZ91,CC95]) to ideas from algebra and coalgebra. We consider a restricted subset of program improvements that fit the following model: the improvement decides some relation (for example, def-use, congruence, must-alias) and then transforms the program based on this relation. Within this model we contend that the following correspondences hold:

	Pessimistic Improvement	**Optimistic Improvement**
Intermediate results are	sound	unsound
Relations are defined	inductively	coinductively
Guiding notion of equivalence	congruence	bisimilarity
Improvement corresponds to proofs of equivalence by	rewrite proofs	coinduction
Effective method of combining improvements	iterating improvement passes	"superanalysis"
Underlying theme is	algebraic	coalgebraic

We use these correspondences to draw connections between combining improvements and the theorem-proving literature on deciding combinations of theories. This leads to useful insights on why and when improvements should be combined.

Organization. We start by reviewing phase ordering problems in compilers (Section 1.1), combined improvements (Section 1.2), and pessimistic and optimistic analyses (Section 2). We use the example of unreachable code elimination (Section 2.1) to show how optimism and pessimism relate to induction and coinduction (Section 2.2). In Section 2.3 we compare congruence and bisimulation, and in Section 2.4 we discuss how these relate to the problem of removing redundant computations. The second half of our paper draws connections between program improvement and theorem proving. In Section 3.1, we describe a correspondence between iterating program improvements and the Nelson-Oppen procedure for deciding combined theories. In Section 3.2 we argue that optimistic improvers decide coinductively-defined equivalences. In Section 3.3, we argue that a coinductive version of Nelson-Oppen is required to decide combinations of theories of bisimilarity, and this offers an explanation of why superanalysis is strictly better than iterating improvements.

1.1 Phase ordering problems

Suppose we have improvement passes A and B, and for a specific example we take A to be unreachable code elimination and B to be constant propagation. We assume $A, B : \mathsf{Program} \to \mathsf{Program}$, so we can sequence these improvements either as $A \circ B$ or $B \circ A$. These two compositions are generally not equal. For example, both the programs of Figure 1 are equivalent to "return 1." For the first program, we'd like to do unreachable (or dead) code elimination first (to remove the if), then constant propagation. For the second program, we'd like to do constant propagation first (to propagate $y =$ false) followed by unreachable code elimination. In general for a program p, $(B \circ A)(p) \neq (A \circ B)(p)$, which creates the so-called *phase ordering problem.* A common solution is to iterate the passes until a fixpoint is reached (e.g. [Muc00]). We'll call this approach *iterating improvement passes.*

```
Program 1            Program 2

x ← 1                y ← false
if false then        if y then
    x ← 2                return 2
return x             return 1
```

Fig. 1: Two programs to illustrate a phase-ordering problem.

1.2 Combining improvements

Surprisingly, iterating improvement passes is not always the optimal solution. Wegman and Zadeck [WZ91] describe a combination of unreachable code elimination with constant propagation called *conditional constant propagation* (CCP). CCP achieves results strictly better than iterating the two passes separately. Consider the program code:

```
        $x \leftarrow 0$
loop :  if $x = 0$ then
            return $x$
        else
            $x \leftarrow 1$
            goto loop
```

Neither unreachable code nor constant propagation (in their usual versions) can improve this code: unreachable code cannot eliminate the if branch because the value of $x = 0$ is not decided, and constant propagation cannot decide $x = 0$ at the branch because both the assignments $x \leftarrow 0$ and $x \leftarrow 1$ might reach the use of x. However, CCP is able to transform this program to "return 0" because it performs both analyses simultaneously using what are called *optimistic* assumptions.

The observation that combined analyses can be stronger than separate analyses goes back to early work of Cousot and Cousot [CC77,CC79] who considered combined abstract interpretation domains and showed that "automatic combination" of domains would not yield optimal results, and introduced the idea of a "reduced product." More recent work on combining abstract interpretations can be found in [CMB$^+$93,CCH94]. In the imperative world, Click and Cooper [CC95] took CCP as inspiration and developed a general approach to combining analyses. In their approach, one formulates each analysis as a system of lattice equations, and encodes interactions between them using special terms. The equations are then solved simultaneously using a global "optimistic" assumption. Click also demonstrated an efficient combined analysis for unreachable code, constant propagation and global value numbering. See [CDG96,PH99,LGC02] for recent developments in combining imperative program analyses.

2 Pessimism and Optimism

In this section we compare two flavours of program analysis known as *pessimism* and *optimism* in the imperative compilers literature. We prefer these terms to least and greatest fixpoints since optimism generally refers to least fixpoints in abstract interpretation, and greatest fixpoints in the imperative compilers literature. In a pessimistic analysis, intermediate results are semantics-preserving. An *optimistic* analysis assumes that a program may be transformed maximally; intermediate results are not sound.

We start by noting that there are two major approaches to program improvement: the first is term rewriting (see e.g. [Vis01]), in which we apply semantics-preserving rules such as:

$$\begin{array}{rcl} \textsf{if true then } a \textsf{ else } b & \mapsto & a \\ \textsf{if false then } a \textsf{ else } b & \mapsto & b \end{array}$$

Rewrite systems are an example of the *pessimistic* approach, in the sense that a term is transformed via a sequence of semantics-preserving steps.

The second, more common approach is to perform a program analysis followed by a transformation. Analyses are usually lattice-based, and are usually abstract interpretations [CC77,JN95] in the functional community, and dataflow analyses (or more generally, monotone analysis frameworks e.g. [KU77]) in the imperative community.

2.1 Unreachable code elimination

To illustrate the difference between pessimism and optimism we consider the problem of unreachable code elimination, that is, removing code that is unreachable in any execution of a program. We review two well-known approaches, one pessimistic and one optimistic. For the purposes of unreachable code elimination a program may be modelled as a graph $G = (V, E)$ whose vertices represent sequential code (basic blocks) and edges model control flow; Figure 2 shows an example. We define a set $\mathcal{U}_0$ of unreachable vertices by $x \in \mathcal{U}_0$ if and only if vertex x is unreachable in any execution. Obviously $\mathcal{U}_0$ is undecidable in general; instead we decide a conservative approximation $\mathcal{U} \subseteq \mathcal{U}_0$, and delete all program points $x \in \mathcal{U}$ found by our analysis to be unreachable.

We assume a distinguished start node Start $\in V$, and we define $\text{pred}(x)$ and $\text{succ}(x)$ to be the predecessors and successors of x (i.e. $\text{pred}(x) = \{y \mid (y,x) \in E\}$). We write $\overline{\mathcal{U}} = V \setminus \mathcal{U}$ for the complement: the set of *reachable* vertices. How we handle $\textsf{if}$ branches is not relevant to this example; we can put in both possibilities, and optionally handle $\textsf{if true} \ldots$ and $\textsf{if false} \ldots$ with a single edge.

Here are two approaches to defining the set $\mathcal{U}$, the first pessimistic and the second optimistic:

1. Initially assume all vertices are reachable i.e. $\mathcal{U} \leftarrow \emptyset$ and $\overline{\mathcal{U}} \leftarrow V$, a *pessimistic* assumption. To define $\mathcal{U}$ we use the rule: A vertex is unreachable if all its

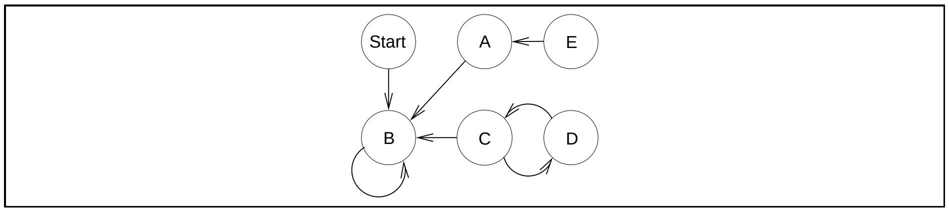


Fig. 2: A simple example for unreachable code elimination. Basic blocks are represented by vertices. The vertex Start is reachable, as is B; the remaining vertices are unreachable. Such a graph might result from function calls, with each vertex viewed as a procedure.

predecessors are unreachable; or, more formally:

$$\frac{\vdash \mathrm{pred}(v) \subseteq \mathcal{U}}{\vdash \{v\} \subseteq \mathcal{U}}$$

from which we can inductively define the set $\mathcal{U}$ of unreachable vertices. Intermediate results of the analysis are semantics-preserving: we can halt the analysis at any point, delete the vertices in $\mathcal{U}$ and have a correct program. (We can view this approach as akin to a graph rewrite system that recognizes single vertices with no incident edges and removes them from the graph.) Figure 3(a) illustrates this approach for the graph of Figure 2.

2. Initially assume all vertices are unreachable i.e. $\mathcal{U} \leftarrow V$ and $\overline{\mathcal{U}} \leftarrow \emptyset$, an *optimistic* assumption. We build the set of reachable vertices $\overline{\mathcal{U}}$ from the rules: (i) Start is reachable; (ii) All successors of a reachable vertex are reachable; or:

$$\frac{}{\vdash \{\mathrm{Start}\} \subseteq \overline{\mathcal{U}}} \qquad \frac{\vdash \{v\} \subseteq \overline{\mathcal{U}}}{\vdash \mathrm{succ}(v) \subseteq \overline{\mathcal{U}}}$$

from which we can inductively define the set $\overline{\mathcal{U}}$ of reachable vertices. Intermediate results are *not* semantics-preserving, since we start by assuming we can throw out all the code in the program. Figure 3(b) illustrates this approach for the graph of Figure 2.

These two approaches are not equivalent, as seen in Figure 3. One might wonder whether we could strengthen the pessimistic approach to make it equivalent to the optimistic approach. We could make the pessimistic approach stronger by adding new rules to handle, for example, pairs of mutually reachable vertices (such as vertices C and D in Figure 2). However, a finite number of graph replacement rules that replace finite-size subgraphs can never remove all unreachable vertices: if the rules allow replacement of a subgraph of size at most k vertices, then a clique of size $k+1$ is not removable. If we added a rule of the form "Any subgraph whose predecessors are unreachable is unreachable," this would require a global analysis to implement, which would turn it into an optimistic, analyze-then-transform approach. If one takes the view that rewrites

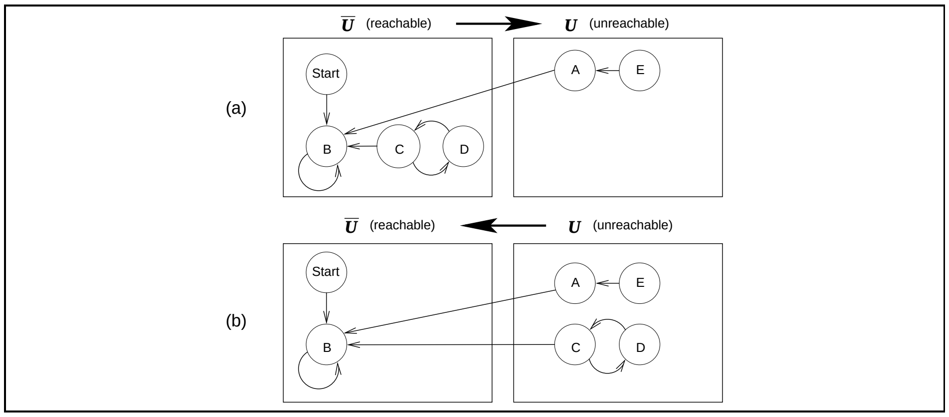


Fig. 3: Unreachable code analysis for the graph of Figure 2. (a) Results of the pessimistic approach: all vertices are initially assumed reachable and are moved into the unreachable set if they have no reachable predecessors. This allows E and then A to be moved into the set $\mathcal{U}$. (b) The optimistic approach, in which vertices are initially assumed unreachable; Start is moved into the reachable set, as are any vertices with reachable predecessors. In the final state only Start and B are in the set $\overline{\mathcal{U}}$.

are "oriented axioms," then it is known that in the presence of cycles, a finite number of axioms cannot capture interesting notions of equivalence (see e.g. [Sew94,BE00]). This implies that traditional rewriting cannot produce normal forms of interesting cyclic structures (note, though, that there is a literature on *cyclic* rewriting e.g. [AK96,Plu98] although we are unsure how this relates.)

2.2 Induction and Coinduction

We can better understand the difference between the pessimistic and optimistic approaches in terms of *inductive* and *coinductive* definitions of a set in some universe X:

- Informally, to define a set inductively we start with an empty set and add elements until everything is there that ought to be. That is, we have a function $f : \wp(X) \to \wp(X)$ that given some subset of X, gives us a larger subset: $A \subseteq f(A)$. The least fixpoint lfp f is the set defined inductively by f.
- To define a set *coinductively*, we start with the full set X and remove elements until everything that ought not to be in the set is gone. That is, we have a function $g : \wp(X) \to \wp(X)$ that given some subset of X, gives us a smaller subset: $g(B) \subseteq B$. The *greatest* fixpoint gfp g is the set defined coinductively by g. (There are various notions of coinduction; the version we use here is sometimes called *set-theoretic coinduction*. Also, coinduction is often presented in terms of a proof principle such as $B \subseteq g(B) \Rightarrow B \subseteq \text{gfp } g$; our presentation is equivalent.)

Compare these to the unreachable code analyses:

- In the pessimistic approach, we initially assumed $\mathcal{U} \leftarrow \emptyset$ and had rules by which we added vertices to $\mathcal{U}$. Thus, $\mathcal{U}$ was defined inductively.
- In the optimistic approach, we initially assumed $\mathcal{U} \leftarrow V$ and had rules by which we moved vertices out of $\mathcal{U}$ and into $\overline{\mathcal{U}}$. Thus, $\mathcal{U}$ was defined coinductively.

The opposite view – that $\overline{\mathcal{U}}$ is defined coinductively by the pessimistic analysis and inductively by the optimistic analysis – is also valid. Induction and coinduction are duals: we can define operators $f^\star(A) = X \setminus f(X \setminus A)$ and $g^\star(A) = X \setminus g(X \setminus A)$ that give a coinductive construction of $X \setminus \mathrm{lfp}\ f$ and an inductive construction of $X \setminus \mathrm{gfp}\ g$; that is, $X \setminus \mathrm{lfp}\ f = \mathrm{gfp}\ f^\star$ and $X \setminus \mathrm{gfp}\ g = \mathrm{lfp}\ g^\star$. Thus, if a set is defined inductively by f then its complement is defined coinductively by $f^\star$, and vice versa.

In general for some relation R we can argue that an analysis is inductive or coinductive depending on whether we talk about R or its complement $\overline{R}$. For this paper we take the view that *larger* relations should cause *more* program improvement. Intuitively, a program improvement is a function that maps a program p to a "better" version of p. We formalize this idea by assuming some preorder $\leq$ capturing an intuitive notion of "better" programs. If p is a program and A is an improver, then we require $A\ p$ to be better:

$$p \leq A\ p \tag{1}$$

Also, we expect that for a finite program (in the sense of finite textual size), there is a "most improved program" corresponding to any p_0; that is, there are no infinite ascending chains $p_0 < p_1 < \ldots$. Two plausible definitions of $\leq$ are (i) $p_1 \leq p_2$ only if p_2 is textually smaller (or the same size as) p_1; (ii) $p_1 \leq p_2$ only if p_2 requires fewer (or as many) computational steps than p_1 in any execution.

We investigate a subset of program improvements that might be called *relation-based improvements*. Such improvements decide a relation R, then use this to transform a program p. We use the notation $p' = \mathcal{T}_p(R)$ to indicate transformation of p by $\mathcal{T}_p$ using a relation R; we adopt the convention that a larger relation causes more improvement:

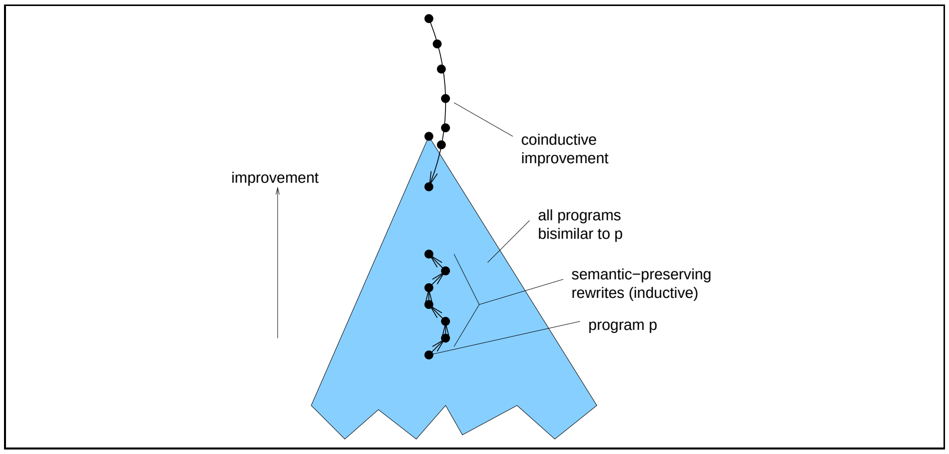
$$R_1 \subseteq R_2 \quad \Rightarrow \quad \mathcal{T}_p(R_1) \leq \mathcal{T}_p(R_2) \tag{2}$$

Using this convention, the pessimistic (rewrite-like) unreachable code elimination is inductive, since we pick $\mathcal{U}$ as the relation; and the optimistic approach is coinductive. We justify this apparently arbitrary convention later by observing that pessimistic improvement appears to correspond to deciding inductively-defined notions of equivalence, and optimistic improvement corresponds to coinductively-defined bisimilarity.

This convention implies some counter-intuitive choices of relations for common analyses. For example, def-use is usually thought of in terms of a binary relation $\mathit{reaches}(x, y)$ which is true only if some definition x might reach a use

y. (In the program "$a \leftarrow {}^{(1)}2;$ `return` ${}^{(2)}a$", the program point ${}^{(1)}$ is a definition, and ${}^{(2)}$ is a use, and we'd say $\mathit{reaches}(1, 2)$). However, the larger the *reaches* relation, the less improvement we can perform; thus, for this paper we'd think instead about a relation $\mathit{notreaches}(x, y)$ – the more reaching definitions we can rule out, the more we can improve a program.

We contend that "optimistic" program improvement might properly be called *coinductive* program improvement: in optimistic analysis one starts from an assumption that allows the program to be improved maximally; thus if an optimistic improvement decides a relation R, by our convention of Eqn. (2), R is defined coinductively. (Later we strengthen this connection by arguing that under reasonable assumptions, optimistic analyses decide coinductively-defined equivalences). An intuitive view of coinductive program improvement is shown in Figure 4.



coinductive improvement
improvement
all programs bisimilar to p
semantic−preserving rewrites (inductive)
program p

Fig. 4: Coinductive (optimistic) program improvement. The shaded region represents the set of programs bisimilar to p. More improved versions of p are higher; less-improved versions are lower, and the shaded region extends infinitely downwards. Inductive program improvement, based on semantics-preserving steps, follows a path from p upwards. Coinductive program improvement starts from a "maximally improved" program and retreats until a program is found in (a decidable approximation of) the bisimilarity class of p.

2.3 Congruence and bisimilarity

Improvements aim to find faster or smaller programs that "do the same thing." There are two relevant notions of what programs *do*, each leading to a different idea of what it means for two programs to *do the same thing*:

1. Programs are *algorithms* which take an input, perform a computation and (ideally) halt with an output. We say two programs are equivalent if by

examining their text we can prove they produce the same output for the same input. This version of program equivalence draws on the algebraic idea of *congruence*, that operations on equivalent inputs produce equivalent outputs. (In an algebra, if $\cong$ is a congruence and f is a k-ary function, then $f(a_1, \ldots, a_k) \cong f(b_1, \ldots, b_k)$ if $a_i \cong b_i$ for $1 \leq i \leq k$). Such equivalences are defined inductively over terms.

2. Programs are *processes* which communicate with their environment and may be designed to never halt (Figure 5). To decide whether two programs are equivalent, we take a black box view: all we can observe are the interactions of a program with its environment, called its *behaviour*. Two programs are equivalent if we can't prove they have different behaviour. This equivalence is called bisimilarity or behavioural equivalence, and has its roots in concurrent communicating processes [Mil89], non-well-founded sets [Acz88,BM96], and coalgebra, which is becoming a popular theory of systems and infinite objects [Rut00,Kur01]. Bisimilarity relations are defined coinductively.

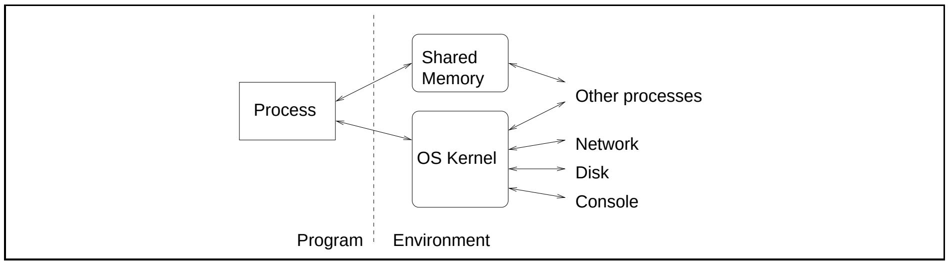


Fig. 5: A program viewed as a *process* that communicates with the operating system kernel and other processes via shared memory. The dotted line is the *interface* through which communication occurs.

Yet another notion of equivalence is *observational equivalence* in pure functional languages. This does not relate directly to the arguments we wish to make.

2.4 Removing redundant computations

For a more obvious example of coinductive program improvement we turn to the problem of removing redundant computations. This is an old and well-studied improvement, known in various incarnations as common subexpression elimination [CS70], value numbering [BCS97], lazy code motion [KRS94] and partial redundancy elimination [KCL$^+$99] (see e.g. [Muc00] for a survey). The inductive approach is straightforward: given some straight-line code such as

$$\begin{aligned} w &\leftarrow +(b, c) \\ x &\leftarrow +(a, w) \\ y &\leftarrow +(b, c) \\ z &\leftarrow +(a, y) \end{aligned}$$

one wishes to eliminate redundant computations – in this example, y and z are redundant since $y \cong w$ and $z \cong x$. The classic inductive approach to this problem is congruence closure (e.g. [DST80]), based on algebraic congruence in term algebras. Inductive approaches use partition *merging*: that is, the congruence quotient is a partition on variables, which are initially assumed to be in separate partitions, and the algorithms advance by merging partitions (deciding variables are congruent).

Unsurprisingly, the inductive approach fails to find all congruences in the presence of loops and recursion. There is a corresponding literature on what we would call coinductive approaches, exemplified by the Alpern-Wegman-Zadeck (AWZ) algorithm [AWZ88]. In this approach, variable definitions in an SSA-form program are viewed as a set of corecursive definitions, and Hopcroft's DFA-minimization algorithm [Hop71] is used to find a maximal "congruence." This approach is based on partition *refinement* (e.g. [PT87]), in which all variables are initially assumed equal and are moved into separate partitions as inequalities are discovered. DFA minimization is *the* canonical example of bisimulation in coalgebra e.g. [Rut98,Kur01]. Clearly, the "congruence" found by the AWZ algorithm is a bisimulation, and has a coinductive definition.

3 Combining Optimizations, Combining Theories

In this section we draw connections between the problem of combining program improvements and the theorem-proving literature on deciding combinations of theories. We consider the ability of improvements to decide whether two computations are equal. If an improver is sound, we can use it to decide (a decidable approximation of) equivalence between two computations. For example, to decide whether the identity $y + 1 \approx 1 + y$ held in the integers, we could pose this problem as a set of variable definitions, encode those definitions as a program, run an improver A on it, and examine the output to determine the equivalence decided by the improver:

$$\begin{array}{ccccccc} & & \vdots & & \vdots & & \\ x_1 = y+1 & \overset{\text{encode}}{\to} & x1 \leftarrow y+1 & \overset{A}{\to} & x1 \leftarrow y+1 & \to & x_1 \sim x_2 \\ x_2 = 1+y & & x2 \leftarrow 1+y & & \text{print } x_1 & & \\ & & \text{print } x_1 & & \text{print } x_1 & & \\ & & \text{print } x_2 & & & & \end{array}$$

The definitions of x_1, x_2 are encoded as a program which is then improved by A, and we deduce from the improved program that $x_1 \sim x_2$ by examining the arguments of the "print" statements. More interestingly, we can encode bisimilarity decision problems by encoding corecursive definitions using recursion, loops, or laziness; the corresponding "print" sections might iterate over infinite objects.

Combining multiple improvers corresponds in a straightforward way to the literature on deciding *combined theories*. This leads to an interesting view of superanalysis as deciding combinations of theories of bisimilarity, which in turn suggests a reason why iterated improvements are not as good as superanalysis.

3.1 The Nelson-Oppen procedure

Recall that a first-order language consists of logical symbols $\forall, \exists, \neg, \wedge, \vee, \rightarrow$, relation symbols such as $=, \leq$ and function symbols such as $+, -$; and that a *structure* gives semantic interpretation to the relation and function symbols in some universe. For example, the integers $(\mathbb{Z}; =, \leq, +, -)$ are a structure for a first-order language with relations $=, \leq$ and function symbols $+, -$. The *theory* of a structure $\mathbb{Z}$ is the set $\mathrm{Th}(\mathbb{Z})$ of all sentences true in the structure. For example, $\forall x(x \leq x+1) \in \mathrm{Th}(\mathbb{Z})$, but $\forall x(-x \leq x) \notin \mathrm{Th}(\mathbb{Z})$ since $-(-2) \not\leq -2$. A *decision procedure* for $\mathbb{Z}$ decides whether or not a sentence φ is in $\mathrm{Th}(\mathbb{Z})$.

Suppose we have several theories, for example a theory $\mathrm{Th}(\mathcal{E})$ of unevaluated function symbols, a theory Th(List) of lists under car, cdr and cons, and a theory $\mathrm{Th}(\mathbb{Z})$ of the integers, and decision procedures for each of these. The problem of deciding *combined theories* is to decide sentences of mixed function and predicate symbols; for example, does $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z,w)))$ hold?

There are several well-known approaches, of which the two most widely used are Nelson-Oppen [NO79] and Shostak's [Sho84]. Of these, the Nelson-Oppen method [NO79] for deciding quantifier-free (or ground) combined theories has an interesting correspondence to combining program improvements. The main idea behind Nelson-Oppen is to split a mixed sentence such as $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z,w)))$ into smaller sentences, each of which contain function and relation symbols of a single theory (here, we introduce new variables a, d, e to split the sentence into fragments):

$\mathrm{Th}(\mathbb{Z})$	Th(List)	$\mathrm{Th}(\mathcal{E})$
$a = z+1$	$d = \mathsf{car}(\mathsf{cons}(z,w))$	$f(a) = f(e)$
$e = 1+d$		

Each decision procedure is responsible for finding equalities implied by their theories; these equalities are then propagated to the other decision procedures. Nelson-Oppen only applies when there are no predicate symbols shared between theories. Each decision procedure may decide things about its own predicate symbols, but this information only becomes visible to other decision procedures when it implies an equality. (For example, the decision procedure for $\mathrm{Th}(\mathbb{Z})$ may deduce from $x \geq y$ and $y \geq x$ that $x = y$).

In the above example, the decision procedure for Th(List) can deduce $\mathsf{car}(\mathsf{cons}(z, w)) = z$ and therefore $d = z$. The equality $d = z$ is then propagated to the other theories. The decision procedure for $\mathrm{Th}(\mathbb{Z})$ can then deduce $z+1 = 1+d$ (since $1+d = 1+z = z+1$) and therefore $a = e$. Since $a = e$, the decision procedure for $\mathrm{Th}(\mathcal{E})$ deduces $f(a) = f(e)$. (We're assuming the function f is free of side effects.) Therefore $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z)))$ is true in the combined theory. Nelson-Oppen uses an inductively defined notion of equality-across-theories, and follows a partition merging approach (c.f. Section 2.4).

Compare this to how an optimizing compiler composed of separate improvement passes would transform a program containing the expression $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z)))$. In a typical intermediate language this expression would be lowered to a set of simple definitions in A-normal [FSDF93] or quad form [Muc00]. Figure 6 illustrates how an optimizing compiler might improve the code.

(1)	(2)	(3)	(4)	(5)
$a \leftarrow z+1$	$a \leftarrow z+1$	$a \leftarrow z+1$	$a \leftarrow z+1$	$h \leftarrow \mathit{true}$
$b \leftarrow f(a)$	$b \leftarrow f(a)$	$b \leftarrow f(a)$	$b \leftarrow f(a)$	
$c \leftarrow \mathsf{cons}(z,w)$	$e \leftarrow 1+z$	$g \leftarrow f(a)$	$h \leftarrow b = b$	
$d \leftarrow \mathsf{car}(c)$	$g \leftarrow f(e)$	$h \leftarrow b = g$		
$e \leftarrow 1+d$	$h \leftarrow b = g$			
$g \leftarrow f(e)$				
$h \leftarrow b = g$				

Fig. 6: Example of how an optimizing compiler might optimize the expression $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z,w)))$ in a sequence of improvement passes: (1) the initial program in lowered form; (2) after list optimizations; (3) after an integer arithmetic pass; (4) after common subexpression elimination (we assume the function f is free of side-effects); (5) after dead variable elimination, assuming h is needed at later program points but a, b are not.

There is a straightforward correspondence between Nelson-Oppen and iterated improvement passes: in Nelson-Oppen, decision procedures communicate by propagating equalities they discover. In iterated improvement passes, there is an implicit communication of equalities between passes via rewriting and replacement. If by a series of improvements the compiler turns a term t into a term t', the process is akin to a *rewrite proof* of $t \cong t'$ (e.g. [Jou95]). Hence one can think of improvement passes communicating equalities via rewrite proofs.

Both approaches are based on an *inductively*-defined equality relation across theories, and both approaches require that we iteratively apply decision procedures (improvement passes) to decide a combined theory (combined improvement).

Term graphs (e.g. [Plu98,AK96]) provide a useful view of this problem. Figure 7 represents the expression $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z,w)))$ as a term graph, and we have grouped vertices into subgraphs corresponding to their appropriate theory. Note that this term graph is acyclic; neither Nelson-Oppen nor iterated improvement passes can decide equivalence of cyclic definitions that span theories, since this would require a coinductive approach.

3.2 Improvements and the equivalences they decide

In this section we offer evidence that under certain reasonable assumptions, optimistic improvements decide coinductively defined equivalences, and pessimistic improvements decide inductively defined equivalences. Our approach is to con-

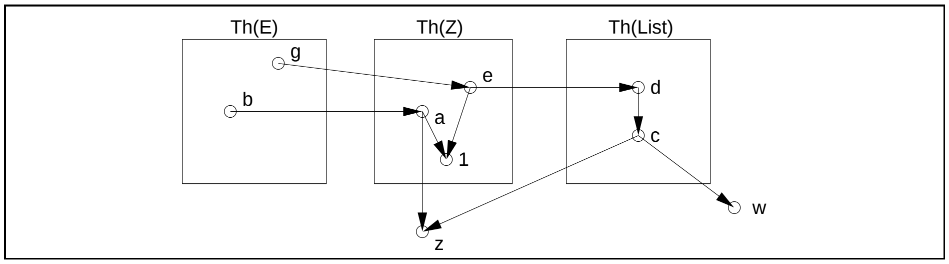


Fig. 7: Term graph of the expression $f(z+1) = f(1+\mathsf{car}(\mathsf{cons}(z,w)))$; Refer to Figure 6(1) for the definitions of the variables $a \ldots g$. Subgraphs represent the theories of (from left to right) unevaluated function symbols $\mathrm{Th}(\mathcal{E})$, integers $\mathrm{Th}(\mathbb{Z})$, and $\mathrm{Th}(\mathrm{List})$.

sider a set of (possibly corecursive) definitions:

$$\begin{aligned} x_1 &= F_1(x_1,\ldots,x_n,y_1,\ldots,y_m) \\ &\vdots \qquad\qquad \vdots \\ x_n &= F_k(x_1,\ldots,x_n,y_1,\ldots,y_m) \end{aligned}$$

We're interested in the power of an improvement A to decide such equivalences. We assume these definitions may be encoded in a program p in such a way that there exists some function to extract from $A\ p$ an equivalence that has been decided on $\mathcal{X} = \{x_1,\ldots,x_n\}$ (see the example of Section 3). That is, we have a function $\mathrm{Eq} : \mathsf{Program} \to \mathcal{X} \times \mathcal{X}$ that yields for some improved version $A\ p$ an equivalence $\sim\ = \mathrm{Eq}(A\ p)$ over $\mathcal{X}$. The improvement A may be implemented by deciding a relation such as def-use or must-alias that is not an equivalence; however, we use A to obtain an equivalence over the variables $\mathcal{X}$.

A reasonable assumption is that more improved programs correspond to larger equivalences on $\mathcal{X}$:

$$p_1 \le p_2 \ \Rightarrow\ \mathrm{Eq}(p_1) \subseteq \mathrm{Eq}(p_2) \tag{3}$$

That is, the more computations are redundant, the more a program can be improved.

We start with the case of a optimistic (coinductive) improvement A, and argue that if Eqn. (3) holds then the equivalence $\mathrm{Eq}(Ap)$ is defined coinductively. Recall that since A is a coinductive improvement, it decides some relation R coinductively and $A\ p = \mathcal{T}_p(R)$, where $\mathcal{T}_p$ is a transformation satisfying Eqn. (2); that is, larger relations cause more improvement. Let $\mathcal{R}_p$ be the universe for the relation (that is, $R \subseteq \mathcal{R}_p$ for any R). Since R is defined coinductively, the analysis computes R via some function $g_p : \wp(\mathcal{R}_p) \to \wp(\mathcal{R}_p)$ such that $g_p(R') \subseteq R'$ for any R' and $R = \mathrm{gfp}\ g_p$. We write $g_p^i(\mathcal{R}_p)$ to mean $g_p^0(\mathcal{R}_p) = \mathcal{R}_p$ and $g_p^{i+1}(\mathcal{R}_p) = g_p(g_p^i(\mathcal{R}_p))$. We "peek" at intermediate analysis results by defining $A^{(i)}p = \mathcal{T}_p(g_p^i(\mathcal{R}_p))$ for $i \ge 0$. From $g_p(R') \subseteq R'$ for any R' we have $g_p^{i+1}(\mathcal{R}_p) \subseteq g_p^i(\mathcal{R}_p)$. By Eqn. (2), $A^{(i+1)}(p) \le A^{(i)}(p)$. Then by Eqn. (3), $\mathrm{Eq}(A^{(i+1)}(p)) \subseteq \mathrm{Eq}(A^{(i)}(p))$.

Therefore the equivalence $\mathrm{Eq}(A\ p)$ follows a descending chain in $\subseteq$ as analysis progresses (i.e. partition refinement), and is therefore decided coinductively.

By a similar argument, if A is a pessimistic (inductive) improvement then $\mathrm{Eq}(A\ p)$ follows an ascending chain in $\subseteq$ as analysis progresses (i.e. partition merging), and therefore is decided inductively.

From this we have strong evidence that iterating improvement passes is an effective method of combining *pessimistic* improvements, since pessimistic improvements decide inductively-defined equivalences, and combinations can be decided by communicating equalities via rewriting and term replacements. In the next section we consider the problem of deciding combinations of coinductively-defined equivalences, i.e. combining optimistic improvements.

3.3 Co-Nelson-Oppen?

In this section we consider the problem of deciding combinations of theories of bisimilarity. To our knowledge a coinductive version of Nelson-Oppen has not been described in the literature; we don't tackle this (ambitious) problem here, but merely sketch a plausible approach and relate this to superanalysis.

Recall that bisimilarity is defined coinductively, and thus by duality the complement of bisimilarity (let's call it inequivalence) is inductively defined. Typically it is this inequivalence relation for which we have a useful inductive definition, leading to a partition refinement scheme. A classic example is the Myhill-Nerode theorem (see e.g. [Hop71]) and its characterization of inequivalent (or distinguishable) states of a DFA.

Since inequivalence is inductively defined, intuition suggests that a coinductive version of Nelson-Oppen should propagate *inequalities* amongst the theories. That is, as in Myhill-Nerode, one should initially assume that all computations are bisimilar, and follow a partition refinement approach, splitting bisimilarity classes when inequivalences are discovered. This is close in spirit to the approach proposed in [CKV74] for minimizing mutually recursive equations, in which the complement of bisimulation is constructed; for a modern account see [AK96].

If this intuition about a coinductive Nelson-Oppen is correct, it suggests the following observation about combining coinductive program improvements: If we require that intermediate programs between improvement passes be sound, then improvement passes can communicate only *equalities* through rewriting. Therefore iterated improvement passes cannot effectively communicate *inequalities.* A plausible explanation for the effectiveness of superanalysis is that it combines coinductive theories by allowing the propagation of *inequalities* amongst theories. Figure 8 summarizes our arguments about combining improvements and combining theories.

4 Conclusions

We now return to the questions posed in the introduction. From the arguments of this paper we are able to make the following conjectures:

(a) Acyclic definitions: decidable by iterating pessimistic improvement passes. Rewriting propagates equalities amongst theories, à la Nelson-Oppen.

(b) Cycles within theories: decidable by iterating optimistic improvement passes. Coinduction within theories, induction between theories. Rewriting propagates equalities amongst theories.

(c) Cycles across theories: requires simultaneous coinduction across theories. Super-analysis propagates *inequalities* amongst theories.

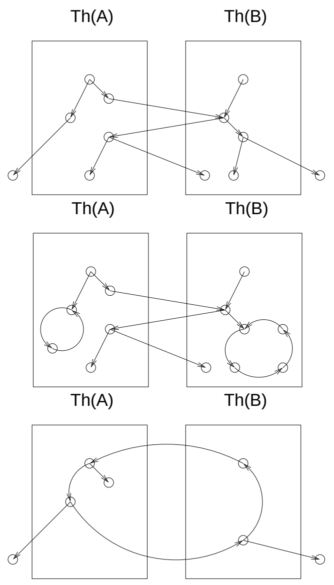


Fig. 8: Effectiveness of combined-theory deciders at deciding equivalences. The figures illustrate several cases of cyclic definitions in term graphs (Section 3.1).

We believe that iterating improvement passes is an effective means of combining *pessimistic* improvements due to the correspondence with Nelson-Oppen outlined in Section 3.1. We believe there is no reason (other than efficiency) to perform pessimistic improvements simultaneously.

We argued that optimistic improvements decide equivalences coinductively. In Section 3.3 we argued that deciding combinations of coinductive theories of equivalence would require propagating *inequalities* amongst decision procedures. Separate improvement passes can only communicate *equalities* via semantics-preserving term replacements. This suggests that optimistic improvements are better if combined, since "superanalysis" allows analyses to communicate directly.

We have not attempted to address the third question – of automatically combining analyses without specifying their interaction. Lerner et al. [LGC02] have proposed a clever method that relies on analyses communicating implicitly through replacement operations. Our arguments suggest that their approach may not be optimal; although they prove their approach is as effective as iterating analyses, they do not prove it is as effective as a manual combination (although they demonstrate this experimentally for some sample analyses). In our view, the third question remains open, and we hope that the connections drawn in this paper may spur research in this direction.

4.1 Acknowledgments

We thank Ron Garcia, Jaakko Järvi, Andrew Lumsdaine, Larry Moss, Amr Sabry, Dipwanita Sarkar, Oscar Waddell, and Jeremiah Willcock for helpful discussions and comments on drafts of this paper.

References

[Acz88] Peter Aczel. *Non-Well-Founded Sets.* Center for the Study of Language and Information, Stanford University, 1988. CSLI Lecture Notes, Volume 14.

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Fundamenta Informaticae*, 26(3/4):207–240, 1996. Extended version as University of Oregon Technical Report CIS-TR-95-16.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equalities of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[BCS97] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software - Practice and Experience*, 27(6):701–724, 1997.

[BE00] Stephen L. Bloom and Zoltan Esik. Iteration algebras are not finitely axiomatizable. In *Latin American Theoretical INformatics (Theory)*, pages 367–376, 2000.

[BM96] Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena.* CSLI Publications, Stanford, California, 1996.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Principles of Programming Languages*, pages 238–252, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM Press, 1979.

[CC95] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[CCH94] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 227–239. ACM Press, 1994.

[CDG96] C. Chambers, J. Dean, and D. Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report TR-96-11-02, University of Washington, Department of Computer Science and Engineering, November 1996.

[CKV74] B. Courcelle, G. Kahn, and J. Vuillemin. Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'équations récursives simples. In Jacques Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 200–213. Springer Verlag, August 1974.

[CMB+93] M. Codish, A. Mulkers, M. Bruynooghe, M. Garcià de la Banda, and M. Hermenegildo. Improving abstract interpretations by combining domains. In *Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 194–205. ACM Press, 1993.

[CS70] J. Cocke and J. Schwartz. Programming languages and their compilers. Technical report, NYU, Courant Inst., April 1970. Second Revised Version.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM (JACM)*, 27(4):758–771, 1980.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation*, 1993.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite-automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[JN95] N. D. Jones and F. Nielson. Abstract interpretation. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4. Oxford University Press, 1995. To appear.

[Jou95] J. Jouannaud. Rewrite proofs and computations. In H. Schwichtenberg, editor, *Proof and Computation*, volume 193 of *Computer and Systems Sciences*. Springer-Verlag, 1995.

[KCL+99] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, January 1977.

[Kur01] Alexander Kurz. Coalgebras and modal logic. ESSLLI 2001 course notes, 2001.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *ACM SIGPLAN Notices*, 31(1):270–282, January 2002.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Muc00] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[PH99] A. Pioli and M. Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical Report 21532, IBM T. J. Watson Research Center, March 1999.

[Plu98] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, , and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1998.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[Rut98] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). *Lecture Notes in Computer Science*, 1466:194–??, 1998.

[Rut00] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[Sew94] Peter Sewell. Bisimulation is not finitely (first order) equationally axiomatisable. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 62–70, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM (JACM)*, 31(1):1–12, 1984.

[Vis01] Eelco Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.