Combining Analyses, Combining Optimizations

CLIFF CLICK and KEITH D. COOPER
Rice University

Modern optimizing compilers use several passes over a program's intermediate representation to generate good code. Many of these optimizations exhibit a phase ordering problem. Getting the best code may require iterating optimizations until a fixed point is reached. Combining these phases can lead to the discovery of more facts about the program, exposing more opportunities for optimization.

This paper presents a framework for describing optimizations. It shows how to combine two such frameworks and how to reason about the properties of the resulting framework. The structure of the framework provides insight into when a combination yields better results. To make the ideas more concrete, this paper presents a framework for combining constant propagation, value numbering, and unreachable code elimination. It is an open question as to what other frameworks can be combined in this way.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — compilers, optimization

Additional Keywords and Phrases: data-flow analysis, optimizing compilers, constant propagation, value numbering

1 Introduction

Modern optimizing compilers make several passes over a program's intermediate representation to generate good code. Many of these optimizations exhibit a phase ordering problem. Different facts are discovered (and different code generated) depending on the order in which the optimizations are executed. Getting the best code requires iterating several optimizations until a fixed point is reached. We will show that by combining optimization passes the compiler discovers more facts about the program, giving more opportunities for optimization.

This has been shown in an *ad hoc* way in previous work — for example Wegman and Zadeck presented an algorithm to combine *constant propagation* and *unreachable code elimination* [10]. This paper provides a more formal basis for describing combinations and shows when and why these combinations yield better results. We present a proof that the simple iterative technique efficiently solves these combined optimizations. Finally, we combine *Conditional Constant Propagation* (CCP) and *Global Value Numbering* (GVN) to get an optimization that is more than the sum of its parts [10, 1].

2 Overview

2.1 Intermediate Representations

Before we describe our algorithms, we need to describe our programs. A program is represented by a *Control Flow Graph* (CFG), where the edges denote flow of control and the vertices are basic blocks. Basic blocks contain a set of assignment statements and a special final statement that may be a **Return**, an **If** or empty (fall through). Program variables are always written in lower case letters (*e.g.*, x, y). To simplify the presentation, we restrict the program to integer arithmetic. The set of all integers is represented by $\mathcal{N}$.

Assignment statements have a single function on the right-hand side and a variable on the left. The function op is of small constant arity (*i.e.*, $x \leftarrow a\, _{op} b$). op may be a constant or the identity function, and is limited to being a k-ary function. This is a reasonable assumption for our application. We run

This work has been supported by ARPA through ONR grant N00014-91-J-1989.

the algorithm over a low-level compiler intermediate representation, with $k \leq 3$. We call the set of *op* functions OP.

We assume the program has been converted into *Static Single Assignment* (SSA) form [3]. In the original program, names are assigned values at multiple definition points. In the SSA form, each name corresponds to a single definition point. By convention, we generate the new names of SSA form by adding subscripts to the original names. This makes the relationship textually obvious to a human reader. Wherever two definitions of the same original variable reach a merge point in the program's control flow, a ϕ-function is inserted in the SSA form. The ϕ-function defines a unique name for the merged value. This is written $x_i \leftarrow \phi(x_f, x_g)$.

Since expressions only occur on the right-hand side of assignments, every expression is associated with an assigned variable. There is a one-to-one correlation between variables and expressions; the variable name can be used as a direct map to the expression that defines it. In our implementation we require this mapping to be fast.[1] Finally, we define N to be the number of statements in the SSA form. The SSA form can be quadratic in the size of the original code; the community's practical experience to date with SSA has shown that it is usually linear or nearly linear in the size of the original code.

2.2 Monotone Analysis Frameworks

In order to combine several optimizations we first describe them in a common *monotone analysis framework* [2, 6, 8]. Briefly, a monotone analysis framework is:

- A set of inferences we make about the program, described as a complete lattice $\mathcal{L} = \{A, \top, \bot, \sqcap\}$ with height d, where:
  - A is an arbitrary set of inferences.
  - $\top$ and $\bot$ are distinguished elements of A, usually called "top" and "bottom" respectively.
  - $\sqcap$ is the *meet* operator such that for any $a, b \in A$,

    $$\begin{array}{ll} a \sqcap a = a, & \text{(idempotent)} \\ a \sqcap b = b \sqcap a, & \text{(commutative)} \\ a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c, & \text{(associative)} \\ a \sqcap \top = a, & \\ a \sqcap \bot = \bot, & \end{array}$$
  - d is the length of the longest chain in $\mathcal{L}$.
- A set of k-ary monotone functions F used to approximate the program, defined as $F \subseteq \{f : \mathcal{L} \mapsto \mathcal{L}\}$ containing the identity function ι and closed under composition and pointwise meet.
- A map $\gamma : OP \mapsto F$ from program primitives to approximation functions. In generally, all the approximation functions are simple tables, mapping lattice elements to lattice elements.[2] We do not directly map the CFG in the framework because some frameworks do not maintain a correspondence with the CFG. Incidentally, this means our solutions are not directly comparable to a *meet over all paths (MOP)* solution like that described by Kam and Ullman [6]. Our value numbering solution is such a framework.

[1] In fact, our implementation replaces the variable name with a pointer to the expression that defines the variable. Performing the mapping requires a single pointer lookup.

[2] Some of our mappings take into account information from the CFG. Here we are taking some notational liberty by defining the mapping from only OP to F. We correct this in our implementation by using an operator-level (instead of basic-block level) *Program Dependence Graph* (PDG) in SSA form. [4, 7] Such a representation does not have a CFG or any basic blocks. Control information is encoded as inputs to functions like any other program variable.

We say that $a \sqsupseteq b$ if and only if $a \sqcap b = b$, and $a \sqsupset b$ if and only if $a \sqsupseteq b$ and $a \neq b$. Because the lattice is complete, $\sqcap$ is closed on A. H The lattice height d is the largest n such that a sequence of elements $x_1, x_2, \ldots, x_n$ in $\mathcal{L}$ form a *chain*: $x_i \sqsupset x_{i+1}$ for $1 \leq i < n$. We require that $\mathcal{L}$ have the *finite descending chain property* – that is, d must be bounded by a constant. For the problems we will examine, d will be quite small (usually 2 or 3).

We assume that we can compute $f \in F$ in time $O(k)$, where k is the largest number of inputs to any function in F. Since $k \leq 3$ in our implementation, we can compute f in constant time.

2.3 Monotone Analysis Problems

Applying a monotone analysis framework to a specific program instantiates a *monotone analysis problem* – a set of simultaneous equations derived directly from the program. Variables in the equations correspond to points of interest in the program; each expression in the program defines its own variable. Associated with each variable is a single inference. Program analysis substitutes approximation functions (chosen from F using γ) for the actual program primitive functions and solves the equations over the approximation functions. (In effect, analysis "runs" an approximate version of the program.) By design, the equations have a minimal solution called the *Greatest Fixed Point* (*gfp*) [2, 9].[3]

Functions in F represent complete programs via composition. Monotonicity gives us some special properties: the composition of monotonic functions is also monotonic. Also the composition is a function onto $\mathcal{L}$:

$$\forall f, g \in F, f : \mathcal{L} \mapsto \mathcal{L}, g : \mathcal{L} \mapsto \mathcal{L} \Rightarrow f \circ g : \mathcal{L} \mapsto \mathcal{L}$$

To solve the set of equations, we can set each variable to $\top$ and then repeatedly compute and propagate local solutions. In effect, the variables "run downhill" to the solution. The limit to k-ary approximation functions ensures that we can compute quickly and that we do not propagate changes too far. The bound on chain length limits the number of forward, or downhill, steps to a finite number. Monotonicity ensures that running downhill locally cannot improve the global solution. This means that the first solution we find is also the best solution. We will prove this formally in Section 4.

2.4 Combining Frameworks

Given any two frameworks, $\mathcal{A}$ and $\mathcal{B}$, a framework that combines them yields better information if and only if their solutions interact. The solution to $\mathcal{A}$ must rely on the solution to $\mathcal{B}$; similarly $\mathcal{B}$ must rely on the solution to $\mathcal{A}$. If they are not interdependent, a careful ordering produces the same answers as the combined framework.

To combine two frameworks, we combine the sets of equations from each framework. The equations of $\mathcal{A}$ implicitly reference facts derived by the equations of $\mathcal{B}$, and *vice-versa*. We make explicit the implicit references to the equations from the other set. We must prove that the combined equations are still monotonic and therefore represent a monotone analysis framework. The combined framework still has a maximal solution; it may not be equal to the combined maximal solution of the individual problems. If, in fact, it is identical to the combination of the individual solutions, we have gained nothing by combining the frameworks (*i.e.*, it was not profitable).

To demonstrate these ideas more clearly, we will work an extended example — combining simple constant propagation and unreachable code elimination. These optimizations are well known and widely used. Wegman and Zadeck have shown an algorithm that combines them [10]. It is instructive to compare

[3] The properties described in Section 2.2 ensure the existence of a *gfp* [9]. These systems of equations can be solved using many techniques. These techniques vary widely in efficiency. The more expensive techniques can solve some systems of equations that the less efficient cannot. In general, the difference in efficiency has led classical compilers to use the least expensive method that will solve a given problem.

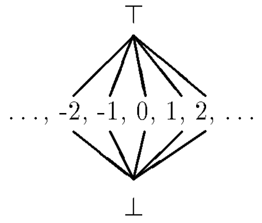
their algorithm with our framework-based approach. As a final exercise, we combine CCP with partition-based GVN [1]. to obtain a new optimization strictly more powerful than any number of repetitions of the separate optimizations. The combination runs in $O(N^2)$ time. This is faster than any previous algorithm.

In our technique, the equations are derived directly from the intermediate representation. For the purposes of this paper, the intermediate representation *is* the equations. We use a technique from *abstract interpretation*, and associate an approximation function chosen from F with every primitive function in the program. Because the form of the equations restricted (see Section 2.2), we can find the maximal solution using an iterative technique in time $O(nk^2d)$ where n is the number of equations to solve (often equal to N, the number of statements in the program), k is largest number of inputs to any one function, and d is the height of the lattice.

3 Simple Constant Propagation

Simple constant propagation looks for program expressions that compute the same value on all executions of the program. It can be cast in a monotone analysis framework. It conservatively approximates the program's control flow by assuming that all basic blocks are executable. Thus, control flow has no explicit expression in the data-flow equations. Each assignment defines an expression and a variable. The inference the variable holds tells us whether the expression computes a constant.

For our inferences we use the standard constant propagation lattice $\mathcal{L}_c$ with the elements $\{\top$, integer constants, $\bot\}$ shown in Figure 1. The meet operator $\sqcap$ is also defined in Figure 1. The notation "$(c_0 = c_1)?c_0 : \bot$" means "if c_0 is equal to c_1 then c_0 else $\bot$". For every primitive function in the program



$\sqcap : \mathcal{L}_c \times \mathcal{L}_c \mapsto \mathcal{L}_c$

$\sqcap$	$\top$	c_0	$\bot$
$\top$	$\top$	c_0	$\bot$
c_1	c_1	$(c_0 = c_1)?c_0 : \bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

Figure 1 The constant propagation lattice $\mathcal{L}_c$ and meet operator

we need a corresponding monotonic function $f \in F$. For ϕ-functions we use the meet operator. We extend the other primitives to handle $\top$ and $\bot$. As an example look at $+ : \mathcal{N} \times \mathcal{N} \mapsto \mathcal{N}$. The corresponding function is $f_+ : \mathcal{L}_c \times \mathcal{L}_c \mapsto \mathcal{L}_c$. Figure 2 defines f_+ by replacing *op* with $+$. The application $f_{op}(\top,\bot)$ is $\top$ instead of $\bot$, meaning that applying a function to undefined inputs yields undefined results (as opposed to unknown results).[4] This embodies the idea that we do not propagate information until all the facts are known.

For functions with zero elements we use a more precise (and more complex) extension. Multiply is extended in Figure 3 so that $f_*(0, \bot)$ is 0 rather than $\bot$. This reflects our intuition that zero times anything is zero. As shown in the figure, the function results are monotonically decreasing[5] both left to right and top to bottom. Thus f_* is a monotonic function, but maintains information in more cases than the corresponding function without the special case for zero.

In Figure 4 we generate the equations for a small program in SSA form. For every variable x in the program we have an equation defining V_x. For the assignment "$x_0 \leftarrow 1$" we generate the equation $V_{x_0} = 1$.

[4] *Unknown* values are values that cannot be discovered at compile time. There are many unknown values in a typical program. The compiler must emit code to compute these values at run time. For *undefined* values, the compiler can choose any value it wants.

[5] Two integers are incomparable in the lattice, thus 0 is neither greater than nor less than $c_0 \times c_1$.

$$f_{op} : \mathcal{L}_c \times \mathcal{L}_c \mapsto \mathcal{L}_c$$

f_{op}	$\top$	c_0	$\bot$
$\top$	$\top$	$\top$	$\top$
c_1	$\top$	$c_0\, op\, c_1$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$

Figure 2 Extending functions to $\mathcal{L}_c$

$$f_* : \mathcal{L}_c \times \mathcal{L}_c \mapsto \mathcal{L}_c$$

f_*	$\top$	0	c_0	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
0	$\top$	0	0	0
c_1	$\top$	0	$c_0 \times c_1$	$\bot$
$\bot$	$\top$	0	$\bot$	$\bot$

Figure 3 Extending multiply to $\mathcal{L}_c$

In this case "1" is a constant function from F. Because V_{x_0} is cumbersome, we will write x_0 instead. The different uses of x_0 will be obvious from context.

For the ϕ-functions we use the meet operator. The function $\neq : \mathcal{N} \times \mathcal{N} \mapsto \mathcal{N}$ returns 0 if the inputs are equal, and 1 if the inputs are not equal. We extend $\neq$ to $f_{\neq}$ as shown in Figure 2. Assume that the loop controlling predicate *pred*() is beyond the ability of the compiler to analyze, possibly a keyboard read. Solving the equations by inspection yields $x_1 = x_3 = b = \bot$. In short, no new constants are found.

Program	Equations
int $x_0 \leftarrow 1$;	$x_0 \quad = 1$
do { $x_1 \leftarrow \phi(x_0, x_3)$;	$x_1 \quad = x_0 \sqcap x_3$
$\quad b \leftarrow (x_1 \neq 1)$;	$b \quad = f_{\neq}(x_1, 1)$
$\quad$ if(b)	
$\quad\quad x_2 \leftarrow 2$;	$x_2 \quad = 2$
$\quad x_3 \leftarrow \phi(x_1, x_2)$;	$x_3 \quad = x_1 \sqcap x_2$
} while(*pred*())	$x_{pred} = \bot$
return(x_3);	

Figure 4 Simple Constant Propagation Example

4 Finding the Greatest Fixed Point

In this section we show that the iterative technique works on monotone analysis frameworks. That is, we show that initializing a system of monotonic equations to $\top$ and successively solving the equations eventually yields the *gfp*. We start with Tarski [9], who proved that every monotone function $f : \mathcal{L} \mapsto \mathcal{L}$ has a *gfp* in a complete lattice $\mathcal{L}$.

Since f is monotone, successive applications of f to $\top$ descend monotonically:

$$\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq \cdots$$

The lattice is bounded so a fixed point $u = f^d(\top) = f^{d+1}(\top)$ is eventually reached. We cannot have $u \sqsupset gfp$ because gfp is the *greatest* fixed point (and u is a fixed point). Suppose we have the reverse situation, $gfp \sqsupset u$. Then the applications of f must form a descending chain that falls past the gfp. There must be some i such that $\cdots\ f^i(\top) \sqsupset gfp \sqsupset f^{i+1}(\top)\ \cdots$. By the monotonicity of f we have $f^i(\top) \sqsupset gfp \Rightarrow f^{i+1}(\top) \sqsupseteq f(gfp)$. Since $f(gfp) = gfp$ we have $f^{i+1}(\top) \sqsupseteq gfp$, a contradiction. Therefore $u = gfp$, and successive applications of f to $\top$ yield the gfp.

We now need to represent a system of monotonic equations as a simple monotonic function f. We extend the lattice $\mathcal{L}$ to tuples in lattice $\vec{\mathcal{L}}$. We define $\vec{\mathcal{L}}$ to be a lattice whose elements are n-tuples of $\mathcal{L}$ elements. We use the notation $\{x_0, x_1, \ldots, x_n\} = \vec{x} \in \vec{\mathcal{L}}$ to refer to a tuple of $\mathcal{L}$ elements. We define $\vec{\mathcal{L}}$'s meet operator as element-wise meet over $\mathcal{L}$ elements:

$$\vec{x}\ \vec{\sqcap}\ \vec{y} = \{x_0 \sqcap y_0, x_1 \sqcap y_1, \ldots, x_n \sqcap y_n\}.$$

$\vec{\mathcal{L}}$ is a complete lattice with the finite descending chain property. We define $\vec{f} : \vec{\mathcal{L}} \mapsto \vec{\mathcal{L}}$ as a monotonic function that is a collection of monotonic functions $f_i : \vec{\mathcal{L}} \mapsto \mathcal{L}$ from the tuple lattice to the regular lattice:

$$\vec{f}(\vec{x}) = \{f_0(\vec{x}), f_1(\vec{x}), \ldots, f_n(\vec{x})\}.$$

Therefore $\vec{y} = \vec{f}(\vec{x})$ defines a system of monotonic equations:

$$\begin{array}{rcl} y_0 & = & f_0(\vec{x}) \\ y_1 & = & f_1(\vec{x}) \\ \vdots & & \vdots \\ y_n & = & f_n(\vec{x}) \end{array}$$

Each of the functions $f_0, f_1, \ldots, f_n$ takes an n-length tuple of $\mathcal{L}$ elements. In the problems that we are solving, we require that only k $(0 \le k \le n)$ elements from $\mathcal{L}$ are actually used. That is, f_i takes as input a k-sized subset of $\vec{x}$. Unfortunately the solution technique of repeatedly applying $\vec{f}$ until a fixed point is reached is very inefficient. $\vec{\mathcal{L}}$ has a lattice height of $O(nd)$, and each computation of $\vec{f}$ might take $O(nk)$ work for a running time of $O(n^2kd)$. The next section presents a more efficient technique based on evaluating the single variable formulation (f_i rather than $\vec{f}$).

To provide efficient solutions, we solve these equations $\vec{y} = \vec{f}(\vec{x})$ with a simple worklist iterative technique [5]. The sparseness of the equations makes the algorithm efficient.

1. Initialize all equation variables x_i to $\top$.
2. Place all equations on a worklist w.
3. While w is not do empty do:
   (a) Remove an equation "$y_i = f_i(\vec{x})$" from worklist w.
   (b) Solve for y_i using the values of other x_i variables.
   (c) If the solution for y_i changes, set x_i to y_i and place all equations that use x_i back on w.

The functions f_i are all monotonic. As the inputs to the functions drop in the lattice, the defined variable can only go lower in the lattice. Because the lattice has height d, a variable can drop (and change) at the most d times. Each function (expression in the program) is on the worklist once per time an input drops in the lattice, or $O(kd)$ times. Each time a function is removed from the worklist it takes time $O(k)$ to evaluate. Total evaluation time per function is $O(k^2d)$, and the total running time is $O(nk^2d)$.

For simple constant propagation, the number of equations n is the number of statements N in the SSA form of the program. If k, the arity of the functions, is small and d is 2, the running time is $O(N)$.

5 Unreachable Code Elimination

We do the same analysis for unreachable code elimination that we did for simple constant propagation. We seek to determine if any code in the program is not executable, either because a control-flow test is based on a constant value or because no other code jumps to it. Our inferences are to determine whether or not a section of code is reachable, expressed as a two-element lattice $\mathcal{L}_u$ with elements $\{\mathcal{U}, \mathcal{R}\}$. $\mathcal{U}$ is unreachable, $\mathcal{R}$ is reachable. We define the functions $+$ and $\cdot$ similar to boolean **or** and **and** in Figure 5. During this analysis the only constant facts available are literal constants. So we define $\not\equiv$ to return $\mathcal{U}$ if both inputs are textually equal and $\mathcal{R}$ otherwise.

$+ : \mathcal{L}_u \times \mathcal{L}_u \mapsto \mathcal{L}_u$

$+$	$\mathcal{U}$	$\mathcal{R}$
$\mathcal{U}$	$\mathcal{U}$	$\mathcal{R}$
$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$

$\cdot : \mathcal{L}_u \times \mathcal{L}_u \mapsto \mathcal{L}_u$

$\cdot$	$\mathcal{U}$	$\mathcal{R}$
$\mathcal{U}$	$\mathcal{U}$	$\mathcal{U}$
$\mathcal{R}$	$\mathcal{U}$	$\mathcal{R}$

Figure 5 **Or**, **And** defined over $\mathcal{L}_u$

In Figure 6 we generate the equations for the same example as before. For readability we define $\mathcal{F}$ as a synonym for 0 and $\mathcal{T}$ as any integer other than 0. The starting statement S_0 is clearly reachable. Statement S_1 is reachable if we can fall into it from S_0 or we can reach the bottom of the loop (statement S_6) and the loop predicate is not always false. The remaining equations are generated in a similar fashion. Our equations are compositions of monotonic functions, and are monotonic.

Because b and *pred* are not literal constants all the $\not\equiv$ tests must return $\mathcal{R}$. The solution is straightforward: everything is reachable.

Program	Equations
int $x_0 \leftarrow 1$;	$S_0 = \mathcal{R}$
do { $x_1 \leftarrow \phi(x_0, x_3)$;	$S_1 = S_0 + S_6 \cdot (pred \not\equiv \mathcal{F})$
$b \leftarrow (x_1 \neq 1)$;	$S_2 = S_1$
if(b)	$S_3 = S_2$
$x_2 \leftarrow 2$;	$S_4 = S_3 \cdot (b \not\equiv \mathcal{F})$
$x_3 \leftarrow \phi(x_1, x_2)$;	$S_5 = S_4 + S_3 \cdot (b \not\equiv \mathcal{T})$
} while(*pred*())	$S_6 = S_5$
return(x_3);	$S_7 = S_6 \cdot (pred \not\equiv \mathcal{T})$

Figure 6 Unreachable code elimination example

If we look closely at this example we can see that when the program is run, x_0 is set to 1, the **if**'s predicate b is always false, and the consequent of the **if** test (S_4) is never executed. Neither simple constant propagation nor unreachable code elimination discovers these facts because each analysis needs a fact that can only be discovered by the other.

6 Combining Analyses

To improve the results of optimization we would like to combine these two analyses. To do this, we need a framework that allows us to describe the combined system, to reason about its properties, and to answer some critical questions. In particular we would like to know: is the combined transformation correct –

that is, does it retain the meaning-preserving properties of the original separate transformations? Is the combination profitable – that is, can it discover facts and improve code in ways that the separate techniques cannot?

If the combined framework is monotonic, then we know a *gfp* exists, and we can find it efficiently. We combine analyses by unioning the set of equations and making explicit the implicit references between the equations. Unioning equations makes a bigger set of unrelated equations. However, the equations remain monotonic so this is safe.

However, if the analyses do not interact there is no profit in combining them. We make the analyses interact by replacing implicit references with functions that take inputs from one of the original frameworks and produce outputs in the other. The reachable equations for S_3 and S_4 use the variable b, which is defined in the constant propagation equations. Instead of testing b against a literal constant we test against an $\mathcal{L}_c$ element. We replace the $\not\equiv$ function with $\leq : \mathcal{L}_c \times \mathcal{N} \mapsto \mathcal{L}_u$ defined in Figure 7. The $\leq$ function is an example of a function that mixes inputs and outputs between the frameworks. $\leq$ takes an input from the $\mathcal{L}_c$ framework and returns a result in the $\mathcal{L}_u$ framework.

The meaning of $\leq$ is easily determined. If we know that the test is not a constant (*i.e.*, $\bot$) then both exit paths are reachable. Otherwise the test is a constant and only one exit path is reachable. At first glance this function looks non-monotonic, but $\mathcal{F}$ and $\mathcal{T}$ are not comparable in $\mathcal{L}_c$.

The important question to answer is:

Are the functions that represent interactions between frameworks monotonic?

If they are, then the combined framework is monotonic and we can use all the tools developed to handle monotonic frameworks. If the original frameworks are monotonic and these transfer functions are monotonic, then the combined framework is monotonic. In combining two frameworks, the implementor *must prove* this property. Fortunately, reasoning about the monotonicity of these transfer functions is no harder than reasoning about the original frameworks. The transfer functions we use in our examples can be represented with small tables. We determine the functions are monotonic by inspection.

6.1 Profitability of Combined Frameworks

When is the combined framework profitable? It is profitable when the *gfp* solution computes a value for a mixed function that is higher in the output lattice than the implicit function that the mixed function replaced. If the mixed function's solution is the same as the implicit function's, the equations using the mixed function have no better results than when they use the implicit function. This means the combined solution fares no better than when the implicit functions were used, which implies the combined solution is no better than the individual solutions.

If we have functions from framework $\mathcal{L}_c$ to framework $\mathcal{L}_u$ but not vice-versa, we have a simple phase-ordering problem. We can solve the $\mathcal{L}_c$ framework before the $\mathcal{L}_u$ framework and achieve results equal to

$\leq : \mathcal{L}_c \times \mathcal{N} \mapsto \mathcal{L}_u$

$\leq$	$\top$	$\mathcal{F}$	$\mathcal{T}$	$\bot$
$\mathcal{F}$	$\mathcal{U}$	$\mathcal{R}$	$\mathcal{U}$	$\mathcal{R}$
$\mathcal{T}$	$\mathcal{U}$	$\mathcal{U}$	$\mathcal{R}$	$\mathcal{R}$

Testing a predicate

$[\Rightarrow] : \mathcal{L}_u \times \mathcal{L}_c \mapsto \mathcal{L}_c$

$[\Rightarrow]$	$\mathcal{U}$	$\mathcal{R}$
$\top$	$\top$	$\top$
c	$\top$	c
$\bot$	$\top$	$\bot$

Unreached code computes $\top$

Figure 7 Mixed functions for the combined equations a combined framework. The combined framework is only profitable if we also have functions from $\mathcal{L}_u$ to $\mathcal{L}_c$ and the combined *gfp* improves on these mixed functions' results.

Back to our example: unreachable code always computes $\top$, so each of the constant propagation equations gets an explicit reference to the appropriate reachable variable. The reachable variable is used as an input to the monotonic infix function $[\Rightarrow] : \mathcal{L}_u \times \mathcal{L}_c \mapsto \mathcal{L}_c$, also defined in Figure 7.

Instead of one equation per statement we have two equations per statement. Each equation has grown by, at most, a constant amount. So, the total size of all equations has grown by a constant factor, and remains linear in the number of statements.

Program	Equations
int $x_0 \leftarrow 1$;	$S_0 = \mathcal{R}$
	$x_0 = [S_0 \Rightarrow 1]$
do { $x_1 \leftarrow \phi(x_0, x_3)$;	$S_1 = S_0 + S_6 \cdot (pred \leq \mathcal{T})$
	$x_1 = [S_1 \Rightarrow (x_0 \sqcap x_3)]$
$b \leftarrow (x_1 \neq 1)$;	$S_2 = S_1$
	$b = [S_2 \Rightarrow f_{\neq}(x_1, 1)]$
if(b)	$S_3 = S_2$
$x_2 \leftarrow 2$;	$S_4 = S_3 \cdot (b \leq \mathcal{T})$
	$x_2 = [S_4 \Rightarrow 2]$
$x_3 \leftarrow \phi(x_1, x_2)$;	$S_5 = S_4 + S_3 \cdot (b \leq \mathcal{F})$
	$x_3 = [S_5 \Rightarrow (x_1 \sqcap x_2)]$
} while(*pred*())	$S_6 = S_5$
	$pred = \perp$
return(x_3);	$S_7 = S_6 \cdot (pred \leq \mathcal{F})$

Figure 8 Combined example

6.2 An Example

The complete new equations are given in Figure 8. Solving the equations is straightforward but tedious. A solution using the iterative technique given in Section 4 is presented in Figure 9. In this example the $\boxed{\text{boxed}}$ terms represent y_i in step 3b (the variable being computed).

The main points of interests are the interactions between the separate analyses in the equations for x_1, b, S_3, x_2 and x_3.

Time 0: x_3 (along with all other variables) is initialized to $\top$ or $\mathcal{U}$.

Time 4: x_1 meets x_0 and x_3 to get the value 1.

Time 6: b is a constant $\mathcal{F}$.

Time 8: S_4 is left marked unreachable ($\mathcal{U}$). Since S_4 did not change values, the equation setting x_2 is never placed on the worklist.

Time 10: x_3 meets x_1 and x_2 to get the value 1. Since x_3 changed from $\top$ to 1 users of x_3 (including x_1's equation) go back on the worklist.

Time 14: x_1's original value remains unchanged.

The solution stabilizes. Statement S_3 is marked unreachable and x_3 is known to be a constant 1. These facts are not found by the individual analyses. Repeated applications of these separate analyses cannot

Time	S_0	x_0	S_1	x_1	S_2	b	S_3	S_4	x_2	S_5	x_3	S_6	$pred$	S_7
0	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
1	$\boxed{\mathcal{R}}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
2	$\mathcal{R}$	$\boxed{1}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
3	$\mathcal{R}$	1	$\boxed{\mathcal{R}}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
4	$\mathcal{R}$	1	$\mathcal{R}$	$\boxed{1}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
5	$\mathcal{R}$	1	$\mathcal{R}$	1	$\boxed{\mathcal{R}}$	$\top$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
6	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\boxed{\mathcal{F}}$	$\mathcal{U}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
7	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\boxed{\mathcal{R}}$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
8	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\boxed{\mathcal{U}}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
9	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\mathcal{U}$	$\top$	$\boxed{\mathcal{R}}$	$\top$	$\mathcal{U}$	$\top$	$\mathcal{U}$
10	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\mathcal{U}$	$\top$	$\mathcal{R}$	$\boxed{1}$	$\mathcal{U}$	$\top$	$\mathcal{U}$
11	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\mathcal{U}$	$\top$	$\mathcal{R}$	1	$\boxed{\mathcal{R}}$	$\top$	$\mathcal{U}$
12	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\mathcal{U}$	$\top$	$\mathcal{R}$	1	$\mathcal{R}$	$\boxed{\bot}$	$\mathcal{U}$
13	$\mathcal{R}$	1	$\mathcal{R}$	1	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\mathcal{U}$	$\top$	$\mathcal{R}$	1	$\mathcal{R}$	$\bot$	$\boxed{\mathcal{R}}$
14	$\mathcal{R}$	1	$\mathcal{R}$	$\boxed{1}$	$\mathcal{R}$	$\mathcal{F}$	$\mathcal{R}$	$\mathcal{U}$	$\top$	$\mathcal{R}$	1	$\mathcal{R}$	$\bot$	$\mathcal{R}$

Figure 9 Solving the combined example

discover any more new facts because each new application starts out with no more facts than the first analysis did. However the combined framework does finds these facts.

The conditional constant propagation algorithm of Wegman and Zadeck discovers these same facts [10]. The improved solution (more knowledge) derives from the interaction made explicit in the equations. Specifically, the combined framework exposes unreachable code precisely because the reachablity equations rely on an assumption made by the constant propagation equations and vice-versa. The assumption is optimistic[6] in the sense that it is not justified by the facts at hand, but must be proven correct. If the assumption is not correct, the equations must be re-evaluated in light of the new (more pessimistic, but more correct) information. However, if the assumption is correct, something new may be discovered.

Thus the reason CCP is an improvement over simple constant propagation and unreachable code elimination is that there *are* interactions. If, in combining frameworks, no implicit interactions are made explicit, the combined framework can not discover addition facts. In such a case the combined framework consists of two independent sets of equations.

7 Global Value Numbering

We can formulate a monotone analysis framework for finding congruent expressions. We define two expressions to be *congruent* [1] when:

- The two expressions are the *same* expression (reflexivity).
- They compute equivalent functions on congruent inputs.
- They compute the same constant.
- One is an identity function on the other.

[6] Wegman and Zadeck use the term *optimistic assumption*.

Identity functions arise from direct copies in the input language, from algebraic identities (addition of zero), or from merging congruent values in ϕ-functions.[7]

Congruence is a relation that holds between pairs of expressions, so a program of size N has $O(N^2)$ inferences (and $O(N^2)$ equations to solve). We can derive equations involving constant inferences, reachable inferences, and congruence inferences in a straightforward manner. The equations do interact, so there is a benefit in solving the combined problem.

Our inferences are whether or not two expressions are congruent, expressed as a two-element lattice $\mathcal{L}_{\equiv}$ with elements $\{\equiv, \not\equiv\}$. We overload the functions $+$ and $\cdot$ to work with $\mathcal{L}_{\equiv}$ elements in the usual way as shown in Figure 5. We write C_{xy} for a variable that determines whether expression x is congruent to expression y, x_{op} for the primitive function at expression x, and x_i for the ith input to x_{op}.

The basic congruence equation between expressions x and y looks like:

$$\begin{array}{rll} C_{xy} = & (x = y) + & \text{Reflexive} \\ & (x_{op} = y_{op}) \cdot C_{x_1 y_1} \cdot C_{x_2 y_2} \cdots C_{x_k y_k} + & \text{Equal functions on congruent inputs} \\ & (x_{op} = c_0) \cdot (y_{op} = c_0) + & \text{Both are the same constant} \\ & (x_{op} = +) \cdot (x_{1\,op} = 0) \cdot C_{x_2 y} + & \text{Add of a zero} \\ & (x_{op} = +) \cdot C_{x_1 y_2} \cdot C_{x_2 y_1} + & \text{Add is commutative} \\ & (x_{op} = \phi) \cdot C_{x_1 x_2} \cdot C_{x_1 y} + & \text{Merge of congruent values} \\ & \cdots & \text{Other identities, } i.e.\text{, multiply by 1} \end{array}$$

The tests of function values ($i.e.$, $x_{op} = +$) are fixed by the program and can be pre-computed before we solve the general problem. The equations are still complex, but there are a fixed number of special cases, one special case per kind of algebraic identity. The number of inputs to the equation defining C_{xy} is thus fixed by some constant k. All the functions involved in the equation are monotonic, and the composition of these functions is also monotonic. Thus we can use the algorithm in Section 4, and achieve a running time of $O(nk^2 d)$. Since $n = N^2$ and k and d are constants, running time is $O(N^2)$.

We make GVN interact with CCP by modifying the C_{xy} equations to use computed constants instead of textual constants, and noting that undefined values can be congruent to anything.

$$\begin{array}{rll} C_{xy} = & \cdots & \text{As before} \\ & (x = \top) + (y = \top) + (x = y = c_0) + & \text{Either is undefined or both compute same constant} \\ & (x_{op} = +) \cdot (x_1 \sqsupseteq 0) \cdot C_{x_2 y} + & \text{Add of a zero} \\ & \cdots & \end{array}$$

These congruence equations clearly use values from the $\mathcal{L}_c$ lattice. We add mixed functions for the reverse direction by defining the subtraction of congruent values to yield 0, and the compare of congruent values to yield $\mathcal{T}$.[8]

Since we have mixed functions going both to and from $\mathcal{L}_{\equiv}$ and $\mathcal{L}_c$ elements, we have the opportunity to find more congruencies, constants and unreachable code than can be found with separate analyses. Figure 10 is an example of such code.

Having solved our combined problem, what do we do with the results? If the N^2 C_{xy} congruencies form an equivalence relation we have a fairly obvious strategy: pick one expression from each partition to represent all the expressions in that partition. Build a new program from the selected expressions (dropping unreachable code and using constants where possible). However, partitions might contain expressions that

[7] This is a slightly stronger notion of congruence than Alpern, Wegman and Zadeck use. They do not allow for identity functions.

[8] If, during the course of solving the equations, the subtract is of the constants 5 and 3 yielding 2, and the constants are considered congruent we have a conflict. Instead of choosing between 2 or 0 we leave the subtract expression at $\top$. The situation is temporary as 5 and 3 cannot remain congruent.

```
main()
{
  int x = 1;                    //x is the constant 1
  int z = read();               //z is defined but unknown
  int y = z;                    //y is congruent to z here
  while( pred() ) {             //some unknown predicate
    if( y ≠ z )                 //if we know y and z are congruent
      x = 2;                    //then we do not destroy the constant at x
    x = 2 - x;                  //destroy easy congruence between x and 1
    if( x ≠ 1 )                 //if we know x is a constant
      y = 2;                    //then we do not destroy the y-z congruence
  }
  printf("x is %d\n",x);        //x is always 1 here
}
```

Figure 10 A subtle congruence, a little dead code

are algebraic identities of another expression in the same partition. These expressions should not be selected to represent the partition.

In the presence of undefined variables the resulting congruencies do not have to form an equivalence relation. This is because we define an expression x with $x = \top$ as being congruent to all other expressions. Such an expression x starts out being congruent to both the constant 3 and the constant 4. If x drops below $\top$ the problem is resolved as the equations are solved. If x remains at $\top$ (because x is undefined) the congruencies between x and 3 and 4 remain. This can lead to some unusual behavior. In the code in Figure 11 both tests against x are the constant $\mathcal{T}$. The analysis shows that *both* print statements are *always* reached.[9]

```
main()
{
  int x;                        //x is undefined
  if( x == 3 ) printf("x is 3\n");
  if( x == 4 ) printf("x is 4\n");
}
```

Figure 11 Using undefined variables

Most language standards fail to address this particular aspect of using an undefined variable. We believe our interpretation is correct given the facts at hand. The unusual behavior can be avoided by having the analysis look for congruence relations that do not form equivalence relations. Then an arbitrary congruence can be broken (some C_{xy} variable set to $\not\equiv$) and the analysis proceed until another (lower) fixed point is reached. This process can continue until an equivalence relation is formed. In the worst case, all congruencies are broken, and the result is clearly a trivial equivalence relation. This process of breaking congruencies is forcing the analysis to choose between having x congruent to 3 and x congruent to 4.

[9] And the moral of the story is: using undefined variables can lead to unexpected results.

8 Summary

In this paper, we have shown how to combine two code improvement techniques. We assume that the original transformations can be encoded as monotone analysis frameworks. In our model, we can combine two frameworks, reason about the monotonicity of the combined framework, and solve the combined framework efficiently. Furthermore, the structure of the combined framework shows when a combination can produce better results than repeated application of the original transformations, and, equally important, when it cannot.

To illustrate these points, we showed how to combine constant propagation with unreachable code elimination to derive an algorithm with properties similar to the conditional constant propagation algorithm presented by Wegman and Zadeck [10]. Because the combined framework includes transfer functions that take constant values into reachability and reachability values back into the constant propagation, it can discover more constants and more unreachable code than repeated applications of the original transformations.

Finally, we showed how to combine constant propagation, unreachable code elimination, and global value numbering. It is an open question as to what other frameworks can be combined and are profitable to combine. The asymptotic complexity of the presented analysis is $O(N^2)$. It can discover more facts than separate applications of the three transformations.

In general, combining two frameworks is important when it removes a phase ordering problem. The monotone analysis framework for the combined problem will run in $O(nk^2d)$ time, where n is the number of equations, k is the arity of the functions used in the equations, and d is the height of the inference lattice.

REFERENCES

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 1–11, 1988.

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth ACM Symposium on the Principles of Programming Languages*, pages 269–282, San Antonio, Texas, Jan. 1979.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, pages 25–35, 1989.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[5] M. Hecht. *Flow Analysis of Computer Programs.* American Elsevier, North Holland, 1977.

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

[7] R. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. Technical Report TR-90-1152, Cornell University, 1990.

[8] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[9] A. Tarski. A lattice-theoretical fixpoint theorem and it's applications. *Pacific J. Math*, pages 285–309, 1955.

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