Composing Dataflow Analyses and Transformations

Sorin Lerner
Univ. of Washington
lerns@cs.washington.edu

David Grove
IBM T.J. Watson Research Center
groved@us.ibm.com

Craig Chambers
Univ. of Washington
chambers@cs.washington.edu

Abstract

Dataflow analyses can have mutually beneficial interactions. Previous efforts to exploit these interactions have either (1) iteratively performed each individual analysis until no further improvements are discovered or (2) developed "super-analyses" that manually combine conceptually separate analyses. We have devised a new approach that allows analyses to be defined independently while still enabling them to be combined automatically and profitably. Our approach avoids the loss of precision associated with iterating individual analyses and the implementation difficulties of manually writing a super-analysis. The key to our approach is a novel method of implicit communication between the individual components of a super-analysis based on graph transformations. In this paper, we precisely define our approach; we demonstrate that it is sound and it terminates; finally we give experimental results showing that in practice (1) our framework produces results at least as precise as iterating the individual analyses while compiling at least 5 times faster, and (2) our framework achieves the same precision as a manually written super-analysis while incurring a compile-time overhead of less than 20%.

1. INTRODUCTION

Dataflow analyses can interact in mutually beneficial ways, with the solution to one analysis providing information that improves the solution of another, and vice versa. A classic example is constant propagation and unreachable code elimination: performing constant propagation and folding may replace branch predicates with constant boolean values, enabling more code to be identified as unreachable; conversely, eliminating unreachable code can remove non-constant assignments to variables thus improving the precision of constant propagation. Many other combinations of dataflow analyses exhibit similar mutually beneficial interactions.

The possibility of mutually beneficial interactions between analyses is one source of the ubiquitous *phase ordering problem* in optimizing compiler design. If two or more analyses are mutually beneficial, then any ordering of the analyses in which each is run only once may yield sub-optimal results. The most common partial solution used today is to selectively repeat analyses in carefully tuned sequences that strive to enable "most" of the mutually beneficial interactions without performing "too much" useless work. At high optimization levels, some compilers even iteratively apply a sequence of analyses until none of the analysis results change. Unfortunately, in the presence of loops, even this iterative application of analyses can yield solutions that are strictly worse than a combined super-analysis that simultaneously performs all the analyses. When analyzing a loop, optimistic initial assumptions must be made simultaneously for all mutually beneficial analyses to reach the best solution; performing the analyses separately in effect makes pessimistic assumptions about the solutions of all other analyses, from which it is not possible to recover simply by iterating the separate analyses.

By solving all problems simultaneously, super-analyses avoid the phase ordering problem (there is only one phase). However, all previous definitions of super-analyses have required designing and implementing special versions of the analyses with explicit code to exploit the beneficial interactions. For example, each of Wegman and Zadeck's conditional constant propagation algorithms [29, 30] is a special-purpose monolithic super-analysis that simultaneously performs constant propagation and unreachable code elimination. Click and Cooper [9] provide a lattice-theoretic explanation of conditional constant propagation with special flow functions defined over the composed domain. Pioli and Hind [25] developed a monolithic analysis that combines constant propagation and pointer analysis using special combined flow functions. Chambers and Ungar manually combined class analysis, splitting, and inlining [7]. In all these cases, the analyses had to be combined manually in order for them to interact in mutually beneficial ways. In fact, Cousot and Cousot discussed product domains in abstract interpretation, and proved that special flow functions need to be used in order for the combination to produce results better than the analyses performed separately [11]. This is unfortunate, because it seems to demonstrate that it is not possible to simultaneously write dataflow analyses in a modular, reusable, replaceable fashion and achieve the best solutions for analyses that have mutually beneficial interactions.

We present an approach for defining dataflow analyses in a modular way, while also allowing analyses to be automatically combined and interact in mutually beneficial ways.

POPL '02, Jan. 16-18, 2002 Portland, OR USA

This is achieved by changing the way that optimizations are specified. Traditionally, an optimization is defined in two separate parts: (1) an analysis which produces dataflow information and (2) rules for transforming the program representation once the analysis has been solved. Merging these two specifications into one allows our framework to automatically process analyses in novel ways, and in particular it allows our framework to combine modular analyses profitably. The two specifications are combined by extending the definition of flow functions: whereas traditional flow functions only return dataflow values, our flow functions can also return a sub-graph with which to replace the current statement. Such *replacement graphs* are used by our framework in two ways:

- Replacement graphs are used to compute the dataflow information at the program point after the current statement. This is achieved by recursively analyzing the replacement graph in place of the original statement: the input edges of the replacement graph are initialized with the dataflow values flowing into the original statement, and then iterative dataflow analysis is performed on the replacement graph. When this recursive analysis reaches a fixed point, the values on the output edges of the replacement graph are propagated to the program point after the original statement. Once this is done, the replacement graph can be thrown away; the original statement remains unchanged. If the original statement is analyzed again (with more conservative inputs), the flow function can choose another (more conservative) graph transformation, or no transformation at all. Thus, during analysis, graph replacements are used as a convenient way of specifying what might otherwise be a complicated flow function.

- Replacement graphs indicate what final transformations the analysis wants to do. Once a sound fixed point has been reached, the last set of transformations selected during analysis can be applied, yielding an optimized program.

Our new method for specifying optimizations imposes no extra design effort, since the writer of an analysis needs to specify graph transformations anyway to perform changes to the intermediate representation. However, by changing the way in which optimizations are specified, our framework can automatically process analyses in novel ways:

- Because flow functions are allowed to return graph replacements, our framework can automatically simulate transformations *during* analysis, and therefore reach a better solution. For example, using simple and straightforward flow functions defined in our framework, an analysis writer can achieve the same effect as Wegman and Zadeck's more complicated conditional constant propagation algorithms [29, 30] (as will be shown in section 2).

- Our way of defining flow functions allows modular analyses to be automatically combined and achieve mutually beneficial interactions. When one component analysis of a super-analysis selects a transformation, our framework recursively analyzes the replacement graph using the super-analysis, not just the component analysis that selected the transformation. As a result, all other analyses immediately see the transformation, and can benefit from it. This key insight allows analyses that are composed in our framework to *implicitly communicate through graph transformations.*

  This method of communication through graph transformations is natural because it is in fact the way that analyses communicate when they are run in sequence: one analysis makes changes to the program representation that later analyses observe. However, analyses that are automatically composed into a super-analysis cannot interact in this way if the transformations are only considered *after* the super-analysis finishes. Our method of automatically simulating transformations *during* analysis allows individual components of a super-analysis to communicate in the same natural and modular way that sequentially executed analyses do.

The key contributions of this paper can therefore be summarized as follows:

- We introduce a new technique for implicit communication between component analyses based on graph transformations. This allows analyses to be written modularly, while still getting mutually beneficial results when they are composed. Section 2 outlines the key ideas of our approach by showing how several optimizations can be defined in our framework.

- Using abstract interpretation [10], we formalize the use of graph transformations as a method of communication between analyses that are combined together. In particular, we show in sections 3 through 6 that our combined super-analysis terminates, is sound if the individual analyses are sound, and under a certain monotonicity condition is guaranteed to produce no worse results than running arbitrarily iterated sequences of the individual analyses.

- We have implemented our framework in the Vortex compiler [13] and more recently in the Whirlwind compiler. Section 7 provides experimental results showing that our framework can combine modular analyses automatically and profitably with low compile-time overhead.

2. OVERVIEW OF OUR APPROACH

This section highlights the key ideas of our approach by sketching how several optimizations can be defined in our framework and explaining how they work on a few examples. We first show how a single analysis and its transformations can be combined into an integrated analysis, and then we show how several integrated analyses can be automatically combined into a single super-analysis.

2.1 Integrating Analysis and Transformation

Imagine that we have a compiler that uses a simple control flow graph (CFG) intermediate representation. Flow functions usually take as input the analysis information computed at the program point before the statement being analyzed and return the information to propagate to the program point after the statement. The key novelty in our framework is that flow functions also have the option of returning a (possibly empty) sub-CFG with which to replace the current statement. To express this in our examples, we assume that flow functions return something akin to an ML datatype with two constructors: the ***PROPAGATE*** constructor, which specifies some dataflow value to propagate, and the ***REPLACE*** constructor, which specifies a replacement graph for the current statement.

As an initial example, we define a constant propagation and folding pass by giving a few of the key flow functions. The information propagated by this analysis consists of maps, denoted by σ, that associate variables in the program text to either a constant, the symbol $\top$ (which indicates non-constant), or the symbol $\bot$ (which indicates no information).[1]

The first flow function for constant propagation handles statements that assign some constant k to some variable x:

$$\mathcal{F}_{const-prop}[\![\mathtt{x} := \mathtt{k}]\!]\sigma = \boldsymbol{PROPAGATE}(\sigma[x \mapsto k])$$

This flow function says that to analyze statements of this form, the analysis should simply propagate an updated version of its input map that associates x with k (but is otherwise unchanged). Next, consider the flow function for binary arithmetic operations:

$$\begin{array}{l}\mathcal{F}_{const-prop}[\![\mathtt{x} := \mathtt{y\ \ op\ \ z}]\!]\sigma = \\ \quad \textbf{let } t := \sigma[y]\ \ \widehat{op}\ \ \sigma[z] \textbf{ in} \\ \quad\quad \textbf{if } \mathtt{constant}(t) \textbf{ then} \\ \quad\quad\quad \boldsymbol{REPLACE}([\![\mathtt{x} := \mathtt{t}]\!]) \\ \quad\quad \textbf{else} \\ \quad\quad\quad \boldsymbol{PROPAGATE}(\sigma[x \mapsto t])\end{array}$$

This flow function first computes the new abstract value t for x using the $\widehat{op}$ operator, which is the standard extension of op to $\top$ and $\bot$:

$$a\ \widehat{op}\ b = \begin{cases} \bot & \text{if } a = \bot \lor b = \bot \\ \top & \text{if } \neg(a = \bot \lor b = \bot) \land (a = \top \lor b = \top) \\ a\ op\ b & \text{otherwise} \end{cases}$$

If t is a constant, then the flow function performs constant folding by replacing the current statement ($\mathtt{x} := \mathtt{y\ \ op\ \ z}$) with a sub-CFG containing a single statement ($\mathtt{x} := \mathtt{t}$) that assigns x the value computed by the constant folded operation. If t is not constant (either $\top$ or $\bot$) then the flow function simply propagates its input map after updating the binding for x.

Finally, we define a flow function for conditional branches:

$$\begin{array}{l}\mathcal{F}_{const-prop}[\![\mathtt{if\ \ b\ \ then\ \ goto}\ \mathrm{L}_1\ \ \mathtt{else\ \ goto}\ \mathrm{L}_2\ \ \mathtt{end}]\!]\sigma = \\ \quad \textbf{if } \mathtt{constant}(\sigma[b]) \textbf{ then} \\ \quad\quad \textbf{if } \sigma[b] = true \textbf{ then} \\ \quad\quad\quad \boldsymbol{REPLACE}([\![\mathtt{goto}\ \mathrm{L}_1]\!]) \\ \quad\quad \textbf{else} \\ \quad\quad\quad \boldsymbol{REPLACE}([\![\mathtt{goto}\ \mathrm{L}_2]\!]) \\ \quad \textbf{else} \\ \quad\quad \boldsymbol{PROPAGATE}(\sigma)\end{array}$$

This flow function either optimizes a constant conditional branch by replacing it with a direct jump to the appropriate target, or simply propagates its input map unchanged to both targets if the branch condition is not constant.

In all cases, when a ***REPLACE*** action is selected, the replacement graph is analyzed, and the result of the recursive analysis is propagated to the program point after the replaced statement. For example, when $\mathcal{F}_{const-prop}$ returns $\boldsymbol{REPLACE}([\![\mathtt{x} := \mathtt{t}]\!])$, the $\mathtt{x} := \mathtt{t}$ statement is automatically analyzed in place of the original statement, yielding a map that associates x with the constant t. Although returning $\boldsymbol{PROPAGATE}(\sigma[x \rightarrow t])$ in this case would produce the same dataflow value, it would not specify the constant folding transformation. As another example, if a constant branch is replaced with a direct jump, the framework automatically analyzes the direct jump in place of the conditional, and in doing so simulates the removal of the unreachable branch.[2] While in the middle of an optimistic iterative analysis of a statement in a loop, the ***REPLACE*** action is only simulated, with the replacement graph recursively analyzed but otherwise unused. Only after the analysis reaches a sound fixed point is the original CFG modified destructively.

Consider applying the integrated constant propagation and folding optimization to the following simple program:[3]

```
x := 10;
while (...) {
   if (x == 10) {
      DoSomething();
   } else {
      DoSomethingElse();
      x := x + 1;
   }
}
y := x;
```

As it enters the `while` loop, the analysis makes the optimistic assumption that `x` contains the constant 10. As a result, the flow function for the `if` statement chooses to replace the conditional by a jump to the true branch, implicitly deleting the false branch as dead code. However, this transformation is not actually applied yet, since further iteration might invalidate the inputs to the flow function. Instead, the transformation is only simulated, producing the information that `x` holds the value 10 at the end of the `while` loop. This matches the optimistic assumption made when entering the loop, and therefore a fixed point is reached. The most recently selected transformations can now be applied, which results in the following optimized code:

```
x := 10;
while (...) {
   DoSomething();
}
y := 10;
```

The conditional constant propagation algorithms of Wegman and Zadeck [29, 30] would produce the same optimized code as above. However, their algorithm manually simulates the effects of an optimization during analysis, whereas our framework can do this work automatically.

Now consider what happens when the integrated analysis is applied to a very similar program:

[1] Throughout the paper we use the abstract interpretation convention that $\bot$ represents no behaviors of the program and $\top$ represents all possible behaviors. Thus, opposite to the dataflow analysis literature, $\bot$ is the most optimistic information, and $\top$ is the most conservative information.

[2] Unreachable code elimination can either be built into the framework, as in Vortex, or done by a modular pass composed with all other analyses, as in Whirlwind. In this section, we assume the framework provides unreachable code elimination because it makes the examples easier to follow.

[3] For clarity, we use structured constructs such as if and while instead of gotos. However, the underlying flow functions are still evaluated over the nodes of the CFG.

```
x := 10;
while (...) {
   if (x == 10) {
      DoSomething();
      x := x - 1;
   } else {
      DoSomethingElse();
      x := x + 1;
   }
}
y := x;
```

Again, as it enters the `while` loop, the analysis makes the optimistic assumption that `x` contains the constant 10. It also simulates the replacement of the `if` statement with its true sub-statement. However, because of the decrement of `x`, the analysis now associates `x` with 9 at the end of the loop. When the join operations are applied at the loop head to determine whether or not a fixed point has been reached, the framework discovers that its initial assumption was unsound, and it is forced to re-analyze the loop with the more conservative assumption that `x` is $\top$ ($10 \sqcup 9 = \top$). The second time through the loop, the `if` statement does not choose to do a transformation because `x` is not constant, and thus in the end no optimizations are performed.

2.2 Combining Multiple Integrated Analyses and Transformations

The next example illustrates how our approach allows multiple modular analyses to communicate through graph replacements when they are automatically combined into a super-analysis. As a result of the communication through graph replacements, the composed super-analysis is able to exploit mutually beneficial interactions even though the individual analyses were written separately. We first define two more optimizations, class analysis and inlining. For class analysis (which maps each variable to the set of classes of which values in the variable might be instances), we provide flow functions for new statements, message send statements (virtual function calls), and instance-of tests. These flow functions use two helper functions: `subclasses`, which returns the subclasses of a given class, and `method_lookup`, which returns the function that results from doing a method lookup on a given class and a message id.

$$\mathcal{F}_{class-analysis}[\![\texttt{x := new C}]\!]\sigma = \textbf{\textit{PROPAGATE}}(\sigma[x \mapsto \{C\}])$$

$$\begin{array}{l}
\mathcal{F}_{class-analysis}[\![\texttt{x := send y.ID(z}_1,\ldots,\texttt{z}_n\texttt{)}]\!]\sigma = \\
\quad \textbf{let } \mathit{methods} = \bigcup_{c\in\sigma[y]} \texttt{method_lookup}(c, \mathit{ID}) \textbf{ in} \\
\quad\quad \textbf{if } \mathit{methods} = \{F\} \textbf{ then} \\
\quad\quad\quad \textbf{\textit{REPLACE}}([\![\texttt{x := F(y,z}_1,\ldots,\texttt{z}_n\texttt{)}]\!]) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{\textit{PROPAGATE}}(\sigma[x \mapsto \top])
\end{array}$$

$$\begin{array}{l}
\mathcal{F}_{class-analysis}[\![\texttt{x := y instanceof C}]\!]\sigma = \\
\quad \textbf{if } \sigma[y] \subseteq \texttt{subclasses}(C) \textbf{ then} \\
\quad\quad \textbf{\textit{REPLACE}}([\![\texttt{x := true}]\!]) \\
\quad \textbf{else if } \sigma[y] \cap \texttt{subclasses}(C) = \emptyset \textbf{ then} \\
\quad\quad \textbf{\textit{REPLACE}}([\![\texttt{x := false}]\!]) \\
\quad \textbf{else} \\
\quad\quad \textbf{\textit{PROPAGATE}}(\sigma[x \mapsto \{\mathit{Bool}\}])
\end{array}$$

The key flow function for the inlining optimization phase is shown below, where `should_inline` is an inlining heuristic that determines if a particular function should be inlined, and `subst_formals` is used to substitute the formals and the result in the body of the inlined function:[4]

$$\begin{array}{l}
\mathcal{F}_{inlining}[\![\texttt{x := F(y}_1,\ldots,\texttt{y}_n\texttt{)}]\!]\sigma = \\
\quad \textbf{if } \texttt{should_inline}(F) \textbf{ then} \\
\quad\quad \textbf{let } G = \texttt{body}(F) \textbf{ in} \\
\quad\quad\quad \textbf{let } G' = \texttt{subst_formals}(G, x, y_1, \ldots, y_n) \textbf{ in} \\
\quad\quad\quad\quad \textbf{\textit{REPLACE}}([\![\texttt{G}']\!]) \\
\quad \textbf{else} \\
\quad\quad \textbf{\textit{PROPAGATE}}(\sigma)
\end{array}$$

Now imagine that our framework is used to automatically combine these three modularly defined analyses (constant propagation, class analysis, and inlining) into a single super-analysis. The information propagated by the super-analysis is the tuple of the information propagated by the individual analyses, and the flow function for a particular statement is a combination of the flow functions of the individual analyses. The combined flow function performs each of the individual flow functions, accumulating the individual analysis information to propagate. If any individual analysis selects a transformation action, then that transformation action is selected by the composed flow function, causing the whole super-analysis to be applied to the replacement graph, in lieu of the original statement. On the other hand, if all individual analyses select propagation actions, then the overall action of the composed flow function is propagation of the tuple of the individual analysis informations. The separate individual analyses interact through transformations: when one analysis selects a transformation action, all the other analyses are applied to the replacement graph, thereby benefitting from the simplifications of the program representation even if they cannot independently justify the optimization.

Consider applying this super-analysis to the following sample program, where `C` and `D` are unrelated subclasses of the `A` class:

```
     decl x:A;
     x := new C;
     while (...) {
S1:     decl b: Bool;
        b := x instanceof C;
S2:     if (b) {
           x := send x.foo();
        } else {
           x := new D;
        }
S3:  }
     class A {
        method foo():A { return new A; }
     };
     class C extends A {
        method foo():A { return self; }
     };
     class D extends A {
     };
```

The composed analysis function of the first assignment statement selects a propagation action, as all the individual analysis functions select propagation actions. On the first pass through the while loop, optimistic iterative analysis will compute at label `S1` the 3-tuple of information

[4] Note that the inlining optimization is a pure transformation, and all of its flow functions ignore the input dataflow value.

$([x \mapsto \top], [x \mapsto \{C\}], \top)$.[5] The composed analysis of the `instanceof` statement will select the transformation action replacing the computation with `b := true`, since class analysis elects to fold the `instanceof` test. However, the control flow graph is not modified, since the information on entry to the flow function is only tentative and may be invalidated by later iterative approximation. Instead, the replacement graph is analyzed recursively, yielding a combined propagation action that yields the tuple $([x \mapsto \top, b \mapsto true], [x \mapsto \{C\}, b \mapsto \{Bool\}], \top)$ at label `S2`. Analysis proceeds to the `if` statement, where the constant propagation flow function selects a transformation action replacing the conditional branch with a direct jump to the true sub-statement (implicitly deleting the false sub-statement as dead code). As a result, analysis now proceeds to the true sub-statement, where the flow function for class analysis selects a transformation action replacing the message send with a direct procedure call to the `C::foo` procedure. This replacement sub-statement is analyzed, at which time the call is replaced with inlined code, yielding the statement `x := x`. Recursive analysis of this statement doesn't spawn any additional transformation actions, finally propagating dataflow information to label `S3` of $([x \mapsto \top, b \mapsto true], [x \mapsto \{C\}, b \mapsto \{Bool\}], \top)$. After dropping the bindings for out of scope variables (`b`), iterative analysis detects that a fixed point has been reached, at which point the most recently selected transformations are applied, yielding the following optimized code (a later dead-assignment elimination phase could clean up this code further):

```
        decl x:A;
        x := new C;
        while (...) {
S1:         decl b: Bool;
            b := true;
S2:         x := x;
S3:     }
```

This optimized version is sound; it has the same behavior as the original code. But no single optimization phase alone, nor arbitrarily iterated sequences of separate optimization phases, could have produced this code. Class analysis is the only optimization that can fold the `instanceof` test, but it requires constant propagation to fold the `if` statement and thereby delete the other assignment to `x`, and it requires inlining to expose the implementation of the `foo` method to the (intraprocedural) class analysis. If the analyses were run separately, no optimizations at all could be performed.

2.3 Uses in Practice

Our framework has been implemented in the Vortex compiler, which uses a standard CFG representation, and more recently in the Whirlwind compiler, which uses a dataflow graph (DFG) representation augmented with control-edges. Both implementations support forward and backward dataflow analyses, although the analyses that are composed into a super-analysis must all have the same directionality. The Vortex framework has been used to define a number of interesting analyses and optimizations, including constant propagation and folding, symbolic assertion propagation and folding, copy propagation, common sub-expression elimination (CSE), must-point-to analysis, redundant load and store elimination, dead assignment elimination, dead store elimination, class analysis, splitting [7], and inlining. However, some optimizations over the CFG, such as loop-invariant code motion and instruction scheduling, do not currently benefit from the special features of our framework because their optimizations cannot be expressed as local graph transformations. We are currently looking at ways of relaxing the locality of graph replacements, as is explained in an accompanying technical report [22]. Nevertheless, even with the local graph replacement restriction, compiler writers are no worse off using our framework for implementing such analyses than they would be using any other extant dataflow analysis framework: our framework supports writing a *pure analysis* pass (i.e., one that makes no transformations) that can be followed by a separate transformation pass, and in fact this is how loop-invariant code motion has been implemented using our framework in the Vortex compiler.

[5] Recall that inlining is a pure transformation, and does not propagate any meaningful dataflow information. We therefore arbitrarily choose $\top$ as the third element of the tuple to denote the inlining dataflow information.

3. PRELIMINARIES

Now that we have outlined the key ideas of our approach, we proceed to the formalization. In this section we define basic notation and the abstract intermediate representation that we assume throughout the rest of the paper. Section 4 reviews the well-know definition of a single analysis followed by transformations, and serves as a foundation for the formalization of the novel parts of our framework in sections 5 and 6.

3.1 Notation

If A is a set, then A^* is the set $\bigcup_{i\geq 0} A^i$, where $A^k = \{(a_1, \ldots, a_k) | a_i \in A\}$. We denote the i^{th} projection of a tuple $x = (x_1, \ldots, x_k)$ by $x[i] \triangleq x_i$. Given a function $f : A \to B$, we extend f to work over tuples by defining $\overrightarrow{f} : A^* \to B^*$ as $\overrightarrow{f}((x_1, \ldots, x_k)) \triangleq (f(x_1), \ldots, f(x_k))$. We also extend f to work over maps by defining $\widetilde{f} : (O \to A) \to (O \to B)$ as $\widetilde{f}(m) \triangleq \lambda o.f(m(o))$.

We extend a binary relation $R \subseteq 2^{D\times D}$ over D to tuples by defining the $\overrightarrow{R}$ relation by: $\overrightarrow{R}((x_1, \ldots, x_k), (y_1, \ldots, y_k))$ iff $R(x_1, y_1) \wedge \ldots \wedge R(x_k, y_k)$. Finally, we extend a binary relation $R \subseteq 2^{D\times D}$ to maps by defining the $\widetilde{R}$ relation as: $\widetilde{R}(m_1, m_2)$ iff for all elements o in the domain of both m_1 and m_2, it is the case that $R(m_1(o), m_2(o))$. To make the equations clearer, we drop the tilde and arrow annotations on binary relations when they are clear from context.

3.2 Intermediate Representation

We assume that programs are represented by directed multigraphs with nodes representing computations that produce values on their output edges on the basis of the values consumed from their input edges. The exact type of nodes, edges, values, and the relative sparseness/denseness of a particular program representation are orthogonal to the main ideas of this paper. Therefore we suppress them by using an abstract intermediate representation (IR) in which computations are represented by graphs. For example, if the compiler uses a CFG representation, then nodes are program statements, and edges are control-flow edges. If instead the compiler uses a DFG representation, then nodes are primitive computations, and edges are dataflow edges.

A graph in our IR is a tuple $g = (N, E, In, Out, InEdges, OutEdges)$ where $N \subseteq Nodes$ is a set of nodes (with $Nodes$

being a predefined infinite set), $E \subseteq \mathit{Edges}$ is a set of edges (with Edges being a predefined infinite set), $\mathit{In} : N \to E^*$ specifies the input edges for a node, $\mathit{Out} : N \to E^*$ specifies the output edges for a node, $\mathit{InEdges} \in E^*$ specifies the input edges of the graph, and $\mathit{OutEdges} \in E^*$ specifies the output edges of the graph. Each node n in N represents a primitive computation mapping input edges $\mathit{In}(n)$ to output edges $\mathit{Out}(n)$, while a graph analogously represents a computation from input edges $\mathit{InEdges}$ to output edges $\mathit{OutEdges}$. When necessary, we use subscripts to extract the components of a graph. For example, if g is a graph, then its nodes are N_g, its edges are E_g, and so on.

Note that our definition of the intermediate representation uses ordered tuples to represent input and output edges, because unordered sets would not be sufficient to capture some useful representations. For example, the two control-flow output edges of a branch node in a CFG are ordered: one leads to the `true` computation, and the other leads to the `false` computation. Similarly, the two dataflow inputs to the "minus" node in a DFG are ordered.

4. A SINGLE ANALYSIS FOLLOWED BY TRANSFORMATIONS

This section reviews the well-known lattice-theoretic formulation of dataflow analysis frameworks using abstract interpretation [10]. It shows how we use this formulation to define analyses and transformations over the abstract IR defined in the previous section, and provides the foundation for describing our approach in sections 5 and 6.

4.1 Definition

An *analysis* is a tuple $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ where $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ is a complete lattice, $\alpha : D_c \to D$ is the abstraction function, and $F : \mathit{Node} \times D^* \to D^*$ is the flow function for nodes. The elements of D, the domain of the analysis, are dataflow facts about edges in the IR (which would correspond to program points in a CFG representation). The flow function F provides the interpretation of nodes: given a node and a tuple of input dataflow values, one per incoming edge to the node, F produces a tuple of output dataflow values, one per outgoing edge from the node. D_c is the domain of a distinguished analysis, the concrete analysis $\mathcal{C} = (D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, id, F_c)$, which specifies the concrete semantics of the program. For example, one can define $\mathcal{C}$ over a CFG representation using a collecting semantics, with the elements of D_c being sets of concrete stores. Alternatively, for a DFG representation that computes over integer values, the elements of D_c could be sets of integers. $\mathcal{C}$ is fixed throughout the paper, and we assume that F_c and α are continuous.

The solution of an analysis $\mathcal{A}$ over a domain D is provided by the function $S_{\mathcal{A}} : \mathit{Graph} \times D^* \to (\mathit{Edges} \to D)$. Given a graph g and a tuple of abstract values for the input edges of g, $S_{\mathcal{A}}$ returns the final abstract value for each edge in g. This is done by initializing all edges in g to bottom, and then applying the flow functions of $\mathcal{A}$ until a fixed point is reached. A detailed definition of $S_{\mathcal{A}}$ can be found in appendix A.[6]

An *Analysis followed by Transformations*, or an *AT-analysis* for short, is a pair $(\mathcal{A}, R)$ where $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ is an analysis, and $R : \mathit{Node} \times D^* \to \mathit{Graph} \cup \{\epsilon\}$ is a *local replacement function*. The local replacement function R specifies how a node should be transformed after the analysis has been solved. Given a node n and a tuple of elements of D representing the final dataflow analysis solution for the input edges of n, R either returns a graph with which to replace n, or ϵ to indicate that no transformation should be applied to this node. To be syntactically valid, a replacement graph must have the same number of input and output edges as the node it replaces, and its nodes and edges must be unique (so that splicing a replacement graph into the enclosing graph does not cause conflicts). We denote by RF_D the set of all replacement functions over the domain D, or in other words $RF_D = \mathit{Node} \times D^* \to \mathit{Graph} \cup \{\epsilon\}$.

After analysis completes, the intermediate representation is transformed in a separate pass by a transformation function $T : RF_D \times \mathit{Graph} \times (\mathit{Edges} \to D) \to \mathit{Graph}$. Given a replacement function R, a graph g, and the final dataflow analysis solution, T replaces each node in g with the graph returned by R for that node, thus producing a new graph. A detailed definition of T can be found in appendix B. The effect of an analysis followed by transformations is therefore summarized as follows: given an analysis $\mathcal{A}$ over domain D, a replacement function R, an initial graph g, and abstract values $\iota \in D^*$ for the input edges of g, the final graph that $(\mathcal{A}, R)$ produces is $T(R, g, S_{\mathcal{A}}(g, \iota))$.

4.2 Soundness

We want the graph produced by $(\mathcal{A}, R)$ to have the same concrete semantics as the original graph. This if formalized in the following definition of soundness of $(\mathcal{A}, R)$:

Def 1. *Let $(\mathcal{A}, R)$ be an AT-analysis with $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$. Let $(g, \iota_c, \iota_a) \in \mathit{Graph} \times D_c^* \times D_a^*$ such that $\overrightarrow{\alpha}(\iota_c) \sqsubseteq \iota_a$ and let $r = T(R, g, S_{\mathcal{A}}(g, \iota_a))$. We say that $(\mathcal{A}, R)$ is sound iff:*

$$\overrightarrow{S_{\mathcal{C}}(r, \iota_c)}(\mathit{OutEdges}_r) = \overrightarrow{S_{\mathcal{C}}(g, \iota_c)}(\mathit{OutEdges}_g)$$

We define here two conditions that together are sufficient to show that an AT-analysis is sound. First, the analysis $\mathcal{A}$ in $(\mathcal{A}, R)$ must be *locally sound* according to the following definition:

Def 2. *We say that an analysis $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$ is locally sound iff it satisfies the following local soundness property:*

$$\begin{aligned} &\forall (n, cs, ds) \in \mathit{Node} \times D_c^* \times D_a^* . \\ &\quad \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_a(n, ds) \end{aligned} \tag{1}$$

If $\mathcal{A}$ is *locally sound*, then it is possible to show that $\mathcal{A}$ is *sound*, meaning that its solution correctly approximates the solution of the concrete analysis $\mathcal{C}$. This is formalized by the following definition and theorem, the latter of which is proved in an accompanying technical report [22].

Def 3. *We say that an analysis $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$ is sound iff:*

$$\begin{aligned} &\forall (g, \iota_c, \iota_a) \in \mathit{Graph} \times D_c^* \times D_a^* . \\ &\quad \overrightarrow{\alpha}(\iota_c) \sqsubseteq \iota_a \Rightarrow \widetilde{\alpha}(S_{\mathcal{C}}(g, \iota_c)) \sqsubseteq S_{\mathcal{A}}(g, \iota_a) \end{aligned}$$

Theorem 1. *If an analysis $\mathcal{A}$ is locally sound then $\mathcal{A}$ is sound.*

[6] Although the concrete solution function $S_{\mathcal{C}}$ is usually not computable, the mathematical definition of $S_{\mathcal{C}}$ is still perfectly valid. Our framework does not evaluate $S_{\mathcal{C}}$; we only use $S_{\mathcal{C}}$ to formalize the soundness of analyses.

Property (1) is sufficient for proving Theorem 1. Moreover it is weaker than the local consistency property of Cousot and Cousot (property 6.5 in [10]), which is:

$$\forall (n, cs, ds) \in Node \times D_c^* \times D_a^*. \quad \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_a(n, \overrightarrow{\alpha}(cs))$$

Indeed, the above property and the monotonicity of F_a imply property (1). We use the weaker condition (1) because in this way our formalization of soundness does not depend on the monotonicity of F_a. As shown in sections 5 and 6, the flow function F_a is usually generated by our framework and reasoning about its monotonicity requires additional effort on the part of the analysis writer. By decoupling our soundness result from the monotonicity of F_a, we can guarantee soundness even if F_a has not been shown to be monotonic.[7]

Second, R must produce graph replacements that are semantics-preserving. This is formalized by requiring that the replacement function R be *locally sound* according to the following definition:

Def 4. *We say that a replacement function R in $(\mathcal{A}, R)$ is locally sound iff it satisfies the following local soundness property, where $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$:*

$$\begin{aligned} &\forall (n, ds, g) \in Node \times D_a^* \times Graph. \\ &\quad R(n, ds) = g \Rightarrow \\ &\quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ &\quad\quad\quad F_c(n, cs) = \overrightarrow{S_C(g, cs)}(OutEdges_g)] \end{aligned} \tag{2}$$

Property (2) requires that if R decides to replace a node n with a graph g on the basis of some analysis result ds, then for all possible input tuples of concrete values consistent with ds, it must be the case that n and g compute exactly the same output tuple of concrete values. It is not required that n and g produce the same output for all possible inputs, just those consistent with ds. For example, if $\mathcal{A}$ determines that some input edge e to n will always have a value between 1 and 100 then n and g are not required to produce the same output for any input in which e is assigned a value outside of this range.

We say that $(\mathcal{A}, R)$ is *locally sound* iff both $\mathcal{A}$ and R are *locally sound*. If $(\mathcal{A}, R)$ is *locally sound*, then it is possible to show that $(\mathcal{A}, R)$ is *sound* according to definition 1, which means that the final graph produced by $(\mathcal{A}, R)$ has the same concrete behavior as the original graph. This is stated in the following theorem, which is proved in a technical report [22].

Theorem 2. *If an AT-analysis $(\mathcal{A}, R)$ is locally sound, then $(\mathcal{A}, R)$ is sound.*

4.3 Termination

If the lattice has finite height, then the termination of an analysis $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ is guaranteed from within $S_\mathcal{A}$, even if F is not monotonic: as iteration proceeds, $S_\mathcal{A}$ forces the dataflow values to monotonically increase by joining the next solution with the current solution at each step. If the lattice has infinite height, then the flow function for loop header nodes can include widening operators [10] to guarantee termination.

We chose to enforce termination from within $S_\mathcal{A}$, instead of requiring F to be monotonic, for the same reason we chose the weaker soundness condition (1): flow functions are generated by our framework, and proving that they are monotonic requires additional effort on the part of the analysis writer. By having termination and soundness be decoupled from the monotonicity of F, we allow analysis designers the option of not proving that F is monotonic. The drawback of not having F be monotonic is that the fixed point computed by $S_\mathcal{A}$ is not necessarily a least fixed point anymore. As a result, the solution returned by $S_\mathcal{A}$ is not guaranteed to be the most precise one.

[7] Termination in the face of a non-monotonic flow function is discussed in section 4.3.

5. INTEGRATING ANALYSIS AND TRANSFORMATION

Now that we have defined a single analysis followed by some transformations, we proceed to formalizing how our framework integrates an analysis with its transformations.

5.1 Definition

An *Integrated Analysis* is a tuple $\mathcal{IA} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$ where $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ is a complete lattice, $\alpha : D_c \to D$ is the abstraction function, and $FR : Node \times D^* \to D^* \cup Graph$ is a *flow-replacement function*. The flow-replacement function FR takes a node and a tuple of input abstract values, one per incoming edge to the node, and returns either a tuple of output abstract values, one per outgoing edge from the node, or a graph with which to replace the node.

An integrated analysis is an analysis which has been combined with its transformations. The flow replacement function can now return graph transformations that are taken into account during the fixed point computation, and used after the fixed point has been reached to make permanent transformations to the graph. The flow functions defined in section 2 were in fact flow-replacement functions. The ***PROPAGATE*** datatype constructor corresponds to FR returning an element of D^*, whereas the ***REPLACE*** constructor corresponds to FR returning an element of $Graph$.

The meaning of an integrated analysis is defined in terms of an *associated AT-analysis*, for which the behavior has already been defined in section 4.1. Given an integrated analysis $\mathcal{IA} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$, we define the associated AT-analysis $\mathcal{AT}_{\mathcal{IA}}$ as $(\mathcal{A}, R)$, with $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$, where F and R are derived from FR as follows:

$$F(n, ds) = \begin{cases} FR(n, ds) & \text{if } FR(n, ds) \in D^* \\ SolveSubGraph_F(FR(n, ds), ds) & \text{otherwise} \end{cases}$$

$$SolveSubGraph_F(g, ds) = \overrightarrow{S_\mathcal{A}(g, ds)}(OutEdges_g)$$

$$R(n, ds) = \begin{cases} \epsilon & \text{if } FR(n, ds) \in D^* \\ SolveSubGraph_R(FR(n, ds), ds) & \text{otherwise} \end{cases}$$

$$SolveSubGraph_R(g, ds) = T(R, g, S_\mathcal{A}(g, ds))$$

The definition of F above shows how transformations are taken into account while the analysis is running. If FR returns a tuple of dataflow values, then that tuple is immediately returned. If, on the other hand, FR chooses to do a transformation, the replacement graph is recursively analyzed and the dataflow values computed for the output edges of the graph are returned. The next time the same node gets analyzed, FR can choose another graph transformation, or possibly no transformation at all. Transformations are only committed after the analysis has reached a final sound solution, as specified by the definition of R. If at the final dataflow solution, FR returns a tuple of dataflow values, then R returns ϵ, indicating that the analysis has chosen not to do a transformation. If, on the other hand, FR chooses a replacement graph, then R returns this replacement graph after transformations have been applied to it recursively. Although the definition of R above reanalyzes recursive graph replacements, an efficient implementation, such as the ones in Vortex and Whirlwind, can cache the solution of the last replacement graph computed by FR for each node, so that the transformation pass need not recompute them.

5.2 Soundness

An integrated analysis $\mathcal{IA}$ is sound if the associated AT-analysis $\mathcal{AT}_{\mathcal{IA}}$ is sound. We define here conditions that are sufficient to show that $\mathcal{AT}_{\mathcal{IA}}$ is sound, and therefore that $\mathcal{IA}$ is sound. Intuitively, we want the flow-replacement function FR to satisfy condition (1) when it returns a tuple of dataflow values, and condition (2) when it returns a replacement graph. Formally, this amounts to having $\mathcal{IA}$ be locally sound according to the following definition:

Def 5. *We say that an integrated analysis $\mathcal{IA} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$ is locally sound iff it satisfies the following two local soundness properties:*

$$\begin{aligned} &\forall (n, cs, ds) \in Node \times D_c^* \times D^*. \\ &\quad FR(n, ds) \in D^* \Rightarrow \\ &\quad\quad [\overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq FR(n, ds)] \end{aligned} \tag{3}$$

$$\begin{aligned} &\forall (n, ds, g) \in Node \times D^* \times Graph. \\ &\quad FR(n, ds) = g \Rightarrow \\ &\quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ &\quad\quad\quad F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)] \end{aligned} \tag{4}$$

Note that the first property is the same as (1) with F_a replaced by FR, except for the additional antecedent $FR(n, ds) \in D^*$, and the second property is the same as (2), with R replaced by FR.

Theorem 3. *If an integrated analysis $\mathcal{IA}$ is locally sound, then the associated AT-analysis $\mathcal{AT}_{\mathcal{IA}}$ is sound, and therefore $\mathcal{IA}$ is sound.*

Proving Theorem 3 involves showing that if FR satisfies properties (3) and (4), then F and R as defined in section 5.1 satisfy properties (1) and (2) respectively. A proof is given in an accompanying technical report [22].

5.3 Termination

As in the case of an AT-analysis, the function $S_{\mathcal{A}}$ forces the solution to monotonically increase as iteration proceeds, even if the flow function F is not monotonic. If the designer of the analysis puts in the effort to prove that F is monotonic, then $S_{\mathcal{A}}$ computes the least fixed point. Otherwise, the result computed by $S_{\mathcal{A}}$ is not necessarily a least fixed point, but it is nevertheless sound as long as properties (3) and (4) hold.

However, having the solution monotonically increase is no longer sufficient to ensure termination: it is now possible for the flow functions to choose graph replacements that cause infinite recursion of nested graph analysis. For example, an inlining optimization could choose to inline a recursive function indefinitely. To ensure termination, we require that the user's graph replacements do not trigger such endless recursive transformations. Graph replacements either obviously simplify the program (such as deleting a node or replacing a complex node with several simpler ones), and thus cannot cause unbounded recursive graph replacements, or there are standard ways of avoiding endless recursive graph transformations (for instance, by marking selected nodes in the replacement graph as non-replaceable). Our framework does not enforce an arbitrary fixed bound on recursive graph replacements. Instead, we feel that individual dataflow analyses will have their own most appropriate solution, which can be explicitly implemented in the flow-replacement function. This non-termination issue with our framework is already present in any system that iteratively applies analyses and transformations. Such systems have either imposed some fixed bound on the number of iterations, or, as we do, require the analyses to avoid endless transformations.

6. COMBINING MULTIPLE ANALYSES

In this section, we define how our framework automatically combines several modular analyses, while still allowing mutually beneficial interactions.

6.1 Definition

The *Composition of k Integrated Analyses*, or a *Composed Analysis* for short, is a tuple $\mathcal{CA} = (\mathcal{IA}_1, \mathcal{IA}_2, \ldots, \mathcal{IA}_k)$, where each $\mathcal{IA}_i$ is an integrated analysis $(D_i, \sqcup_i, \sqcap_i, \sqsubseteq_i, \top_i, \bot_i, \alpha_i, FR_i)$.

Here again, we define the meaning of a composed analysis in terms of an *associated AT-analysis*. Given a composed analysis $\mathcal{CA} = (\mathcal{IA}_1, \mathcal{IA}_2, \ldots, \mathcal{IA}_k)$, we define the associated AT-analysis $\mathcal{AT}_{\mathcal{CA}}$ as $(\mathcal{A}, R)$, where $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$. We first define the lattice $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ of the composed analysis, then we define the composed abstraction function α, the composed flow function F and finally the composed replacement function R.

Composed lattice. The lattice $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ of the composed analysis is the product of the individual lattices, namely:

- $D \triangleq D_1 \times D_2 \times \ldots \times D_k$
- $\sqcup$ is defined by $(a_1, \ldots, a_k) \sqcup (b_1, \ldots, b_k) \triangleq (a_1 \sqcup_1 b_1, \ldots, a_k \sqcup_k b_k)$
- $\sqcap$ is defined similarly to $\sqcup$
- $\sqsubseteq$ is defined by $(a_1, \ldots, a_k) \sqsubseteq (b_1, \ldots, b_k) \triangleq a_1 \sqsubseteq_1 b_1 \wedge \ldots \wedge a_k \sqsubseteq_k b_k$
- $\top \triangleq (\top_1, \top_2, \ldots, \top_k)$ and $\bot \triangleq (\bot_1, \bot_2, \ldots, \bot_k)$

Composed abstraction function. The abstraction function $\alpha : D_c \rightarrow D$ is defined by $\alpha(c) \triangleq (\alpha_1(c), \ldots, \alpha_k(c))$.

Composed flow function. Before defining F, we must first introduce two helper functions, $c2s$ and $s2c$. The first function, $c2s$ (which stands for "composed to single"), is used to extract the dataflow values of an individual analysis from the dataflow values of the composed analysis. Given an integer i, and an n-tuple of k-tuples, $c2s$ returns an n-tuple whose elements are the i^{th} entries of each k-tuple. Formally:

$$c2s(i, (x_1, \ldots, x_n)) \triangleq (x_1[i], \ldots, x_n[i])$$

For example, if $ds \in D^*$ is a tuple of input values to a node in the composed analysis, then $c2s(i, ds)$ is the tuple of input values to that node for the i^{th} component analysis.

The second function, $s2c$ (which stands for "single to composed") has the exact opposite role as $c2s$: it combines the dataflow values of individual analyses to form the dataflow values of the composed analysis. Formally, it is defined by

$$s2c(x_1, \ldots, x_k) \triangleq ((x_1[1], \ldots, x_k[1]), \ldots, (x_1[n], \ldots, x_k[n]))$$

where each x_i is an n-tuple. For example, if $x_1, \ldots, x_k$ are n-tuples, each one being the result of a single analysis for the n output edges of a given node, then $s2c(x_1, \ldots, x_k)$ is the output tuple of the composed analysis for that node. Also, note that $c2s(i, s2c(x_1, \ldots, x_k)) = x_i$.

We are now ready to give the definition of F:

$$F(n, ds) = s2c(res_1, \ldots, res_k)$$

where for each $i \in [1..k]$:

$$res_i = lres_i \sqcap_i \bigsqcap_{g \in gs_i}{}_i \; c2s(i, SolveSubGraph_F(g, ds))$$

$$lres_i = \begin{cases} fres_i & \text{if } fres_i \in D_i^* \\ c2s(i, SolveSubGraph_F(fres_i, ds)) & \text{otherwise} \end{cases}$$

$$fres_i = FR_i(n, c2s(i, ds))$$

$$gs_i = Graph \cap \bigcup_{j \in [1..k] \wedge j \neq i} \{fres_j\}$$

and

$$SolveSubGraph_F(g, ds) = \overrightarrow{S_{\mathcal{A}}(g, ds)}(OutEdges_g)$$

The above definition looks daunting but it is in fact quite simple. To compute the result res_i of the i^{th} analysis, the composed flow function first determines what the i^{th} analysis would do in isolation by evaluating FR_i and storing the result in $fres_i$. The next step is to determine the dataflow value $lres_i$ that results from the selected action. $lres_i$ is either $fres_i$ if $fres_i$ is a tuple of dataflow values, or the result of a recursive analysis if $fres_i$ is a replacement graph. Finally, the expression for res_i takes into account not only the action of the i^{th} analysis, through the $lres_i$ term, but also the actions of *other* analyses, through the second term. This second term of res_i computes the result of the i^{th} analysis on all graph replacements (gs_i) selected by *other* analyses. Because graph replacements are required to be sound with respect to the concrete semantics, they are sound to apply for any analysis, not just the one that selected them. This means that the result produced by the i^{th} analysis on any graph replacement is sound given the current dataflow approximation. Doing a meet of these recursive results, each of which is sound, provides the most optimistic inference that can be soundly drawn.

This last term in the definition of res_i is important for two reasons. First, it allows analyses to communicate implicitly through graph replacements. If one analysis makes a transformation, then the chosen graph replacement will immediately be seen by other analyses. Second, it ensures the precision result which we cover in section 6.4. Because all potential graph replacements are recursively analyzed, and the returned value is the most optimistic inference that can be drawn from these recursive results, we are guaranteed to get results which are at least as good as any interleaving of the individual analyses. Although analyzing all potential graph replacements is theoretically required to ensure this precision result, an implementation could choose to recursively analyze only a subset of the potential graph replacements. The Vortex and Whirlwind implementations in fact only analyze those graph replacements selected by the $PICK$ function defined below.

Composed replacement function. The definition of R relies on a cost function $PICK : 2^{Graph} \rightarrow Graph \cup \{\epsilon\}$ to select which graph replacement to apply if more than one analysis selects a transformation. Although the composed flow function recursively analyses all graph replacements, only one of these graphs can actually be applied once the analysis has reached fixed point. The $PICK$ function is used to make this decision: given a set of graphs, $PICK$ selects at most one of them to apply, which means that if $PICK(gs) = g$, then either $g = \epsilon$, or $g \in gs$. R can now be defined as follows:

$$R(n, ds) = \begin{cases} \epsilon & \text{if } PICK(gs) = \epsilon \\ SolveSubGraph_R(PICK(gs), ds) & \text{otherwise} \end{cases}$$

$$\text{where } gs = Graph \cap \bigcup_{j \in [1..k]} \{FR_j(n, c2s(j, ds))\}$$

$$SolveSubGraph_R(g, ds) = T(R, g, S_{\mathcal{A}}(g, ds))$$

This definition of R is very similar to the one for an integrated analysis from section 5.1, except that here the $PICK$ function selects which graph to apply from the set (gs) of all potential replacement graphs. If $PICK$ selects no transformation, then R does the same. If, however, $PICK$ chooses a replacement graph, then this replacement graph is returned after transformations have been applied to it recursively.

6.2 Soundness

A composed analysis $\mathcal{CA}$ is sound if the associated AT-analysis $\mathcal{AT}_{\mathcal{CA}}$ is sound. We say that a composed analysis $\mathcal{CA} = (\mathcal{IA}_1, \mathcal{IA}_2, \ldots, \mathcal{IA}_k)$ is *locally sound* if each integrated analysis $\mathcal{IA}_i$ is *locally sound* (according to definition 5).

Theorem 4. *If a composed analysis $\mathcal{CA}$ is locally sound, then the associated AT-analysis $\mathcal{AT}_{\mathcal{CA}}$ is sound, and therefore $\mathcal{CA}$ is sound.*

Theorem 4 says that if each integrated analysis has been shown to be sound (by showing that each one is locally sound), then the composed analysis is sound. Proving Theorem 4 involves showing that if each FR_i satisfies properties (3) and (4), then F and R as defined in section 6.1 satisfy properties (1) and (2) respectively. A proof is given in an accompanying technical report [22].

6.3 Termination

Termination is handled in a similar way to the case of integrated analyses from section 5.3. The only difference is that the analysis designer must now show that the *composed analysis* does not cause endless recursive graph replacements. Even if each integrated analysis by itself does not cause infinite recursive analysis, the interaction between two analyses can. For example, two analyses can oscillate back and forth, the first one optimizing a statement that the second one reverts back to the original form. However, as long as the lattice has finite height, our framework does guarantee that non-termination will never be caused by infinite traversal of the lattice.

6.4 Precision of Composed Analyses

The soundness result from section 6.2 guarantees that the information computed by the composed analysis correctly approximates the actual behavior of the program. It does not however say anything about the precision of the computed information. After all, if the composed flow function always returned $\top$, it would still be sound (and in fact monotonic). In this section we show that in addition to being sound, the composed analysis is at least as precise as any iterated sequence of the individual analyses.

Consider running a set of analyses in sequence without repeating any analysis. This sequence generates for each edge in the original graph one dataflow value per analysis.[8] The composition of these analyses also computes one dataflow value per analysis per edge, except that the method for computing the dataflow values is different. Our precision result states that if the composed flow function is monotonic, then for any edge, the dataflow values computed by the composed analysis are at least as precise (in a lattice theoretic sense) as the dataflow values computed by the analyses running in sequence. This guarantees that the composition cannot do worse than running the analyses in sequence. In practice, however, the composition often does better, and an example of this was shown in section 2.2. Once the precision result for analyses without iteration is proved, it is easy to generalize it for arbitrarily iterated sequences of analyses. We refer the reader to an accompanying technical report [22] for a formal statement and a proof of the precision theorem.

In order to guarantee the precision result, the analysis writer must show that the composed flow function is monotonic. Even if each integrated analysis in the composition is monotonic in isolation, interaction through graph replacements can lead to a non-monotonic composed flow function. In particular, the graph replacement that one analysis chooses may produce non-monotonic results for another analysis. One can prove that the composed flow function is monotonic by establishing a partial order on all the possible replacement graphs for a given node, and showing that smaller inputs to an integrated analysis produce smaller sub-graphs, and that smaller sub-graphs lead to smaller computed values when the combined analysis is recursively solved. This is usually not difficult because for any one given node, there are only a few types of replacement graphs. For example, in the case of virtual function calls, there is only one replacement graph (the one that changes the virtual call to a static call), and in the case of assignment statements, there are only a handful of replacement graphs (such as the empty sub-graph, the sub-graph generated by constant folding, and the sub-graph generated by CSE).

7. EXPERIMENTAL RESULTS

In this section we provide experimental results showing that our approach for communication between analyses is useful in practice. We have collected performance numbers for the Vortex compiler [13] using several Cecil [5] benchmarks. The individual analyses under consideration are: class analysis [7], splitting [7], inlining, constant propagation and folding, common sub-expression elimination, removal of redundant loads and stores, and symbolic assertion propagation.

We used four different configurations of the compiler:

- The *monolithic* configuration uses a manually written monolithic analysis that incorporates all the optimizations of the individual analyses. This analysis was written before our framework was implemented, and it acts as the "gold standard" against which other configurations are measured.
- The *composed* configuration automatically composes the analyses using our framework.
- The *modular-iterated* configuration runs the analyses in sequence, iterating until no more transformations are performed.
- The *modular-once* configuration runs the analyses in sequence once.

Figure 1 shows for each benchmark the runtime of the generated code (the first row of numbers for the benchmark), and the runtime of the compiler (the second row of numbers for the benchmark), all normalized to the monolithic configuration. Due to run-to-run variations, differences of a few percent are not significant. Smaller numbers are better since they indicate faster runtime.

The important facts to note are the following:

- The modular-once configuration generates code that runs 3-13 times slower than the monolithic configuration. This indicates that the analyses exhibit non-

benchmark (num lines[9])	monolithic	composed	modular-iterated	modular-once
queens	1.00	1.02	1.25	13.14
(50)	1.00	1.17	6.17	0.84
life	1.00	1.00	1.09	7.39
(80)	1.00	1.17	5.72	0.83
msort	1.00	0.99	1.01	6.28
(110)	1.00	1.11	6.04	0.20
fft	1.00	1.00	0.98	3.00
(150)	1.00	1.00	6.06	0.71
richards	1.00	1.00	1.07	13.66
(400)	1.00	1.18	6.52	1.03
deltablue	1.00	1.03	0.94	12.89
(650)	1.00	1.20	6.54	0.66
instr-sched	1.00	1.00	1.01	3.78
(2,400)	1.00	1.18	5.80	1.02
typechecker	1.00	1.01	1.01	5.30
(20,000)	1.00	1.18	6.55	0.94
new-tc	1.00	1.05	1.03	4.57
(23,500)	1.00	1.17	6.09	1.16
compiler	1.00	1.02	1.00	4.05
(50,000)	1.00	1.15	7.46	1.22

Figure 1: Performance numbers for the Vortex compiler. For each benchmark, the first row of numbers shows the runtime of the generated code, and the second row shows compile-time, all normalized to the monolithic configuration.

[8]Edges are never removed by the transformation function T. When a node is replaced by an empty subgraph, the adjacent edges are disconnected from the node, but remain in the graph. As a result edges in the original graph are guaranteed to exist, even by the time the last analysis runs, although they may be completely disconnected by then.

[9]The number of lines of code is approximate and does not include 11,000 lines of library code that gets compiled with the benchmarks.

trivial mutually beneficial interactions in these benchmarks. The key interaction here is between inlining and class analysis. Merging these two analyses leads to more precise class analysis information, and this is crucial for optimizing a pure object-oriented language like Cecil, since it allows inlining of message sends in critical loops.

- The iterated configuration generates code that runs nearly as fast as the monolithic version, but slows down compile-time by at least a factor of 5.
- The composed configuration (which uses our framework) generates code that runs as fast as the monolithic version, while incurring a compile-time cost of less than 20%. This shows that our technique for communication through graph transformations can capture (with low compile-time overhead) the cases needed in practice to exploit mutually beneficial interactions.

8. EXTENSIONS TO THE BASE FRAMEWORK

This section describes an extension to the base framework. Other extensions, including the interprocedural aspect of our framework, are described in two technical reports [6, 22].

In addition to supporting communication via graph transformations, our framework also supports communication via what we call *snooping*. Snooping allows the flow function of one analysis to look at the dataflow values being produced by other analyses running in parallel with the first. Snooping is used in our framework to allow analyses that make no transformations (which we call *pure analyses*) to communicate information to other analyses. Pointer analysis, for example, can be framed as a pure analysis, on which other optimizations can snoop. Snooping does not however make communication through graph replacements less useful: although snooping is used to communicate from pure analyses to other analyses, graph transformations are still used to communicate in the other direction, from other analyses to pure analyses. For instance, pointer analysis can produce better results when it is composed with other analyses such as constant propagation and inlining, because it will be exposed to the simplifying graph transformations of the other analyses.

Snooping violates the strict modularity of individual analyses presented so far, because the snooping analysis is aware of the possibility of being combined with other analyses, and knows how to interpret the information they are computing. However, the snooping analysis need not always be combined with the analyses on which it snoops, because default implementations of any missing analyses that simply set all snooped-on edges to $\top$ can be provided automatically by the framework, causing the snooping analysis to behave conservatively. The ability to reuse the snooping analysis in other analysis combinations is not hindered.

9. RELATED WORK

A number of analysis frameworks have been developed for making intra- and interprocedural analyses easier to write and reason about, including Sharlit [27], SPARE [28], FIAT [17], McCAT [19], System-Z [34], PAG [2], the k-tuple dataflow analysis framework [23], and Dwyer and Clarke's system [15]. However, none of these systems address integrating transformations with analyses, nor automatically combining analyses profitably.

Nelson and Oppen [24] describe how under certain conditions satisfiability programs for several theories can be combined into a satisfiability program for the combined theory. Click and Cooper [9] define formally the circumstances in which two dataflow analyses should be integrated to reach better fixed points than repeated sequences of the two analyses run separately. Cousot and Cousot [11] also point out that such interactions can arise. However, in all these cases, the composition needs to be done manually by defining special flow functions over the combined dataflow information.

Whitfield and Soffa [32, 33] have developed a framework for examining the interactions between different optimizations. By analyzing the pre- and post-conditions of optimizations, their framework can determine if one optimization helps or hinders another optimization. This information can then be used to select an order in which to run the optimizations. However, they do not provide a method for exploiting mutually beneficial transformations: when cyclic interactions are found between optimizations, a linear order is still chosen, based on experimental results or on the perceived importance of the optimizations.

Assmann [3, 4] has developed a technique for uniformly specifying analyses and transformations using graph rewrite rules which trigger based on pattern matching. An analysis is defined by rewrite rules that add edges to the graph, thus creating a relation which encodes the analysis results. Transformations are then specified using rewrite rules that trigger on patterns which can include edges added by the analysis, thus allowing transformations based on the analysis results. Assmann's work, however, is not motivated by the phase ordering problem. In fact, he argues that his system works better when the analyses are written individually and run in sequence, instead of having a large graph rewrite system that composes multiple analyses, because it becomes hard to reason about the termination of such large rewrite systems. His formalization of graph rewrite systems is also mainly concerned with termination, and he does not provide soundness or precision results, as we do. Finally, Assmann's framework cannot handle arbitrary abstract interpretations, whereas our framework can. On the other hand, his formulation does allow a richer set of transformations, because edges and nodes can be arbitrarily replaced. Clients of our framework would simply sequence analyses and transformations if non-local graph replacements are needed, as in all other frameworks, including Assmann's.

There is also a large body of literature on advanced and efficient program representations [16, 12, 8, 14, 21, 31, 1, 18, 20, 26]. The definition of our framework is independent of the specific program representation used, and thus our work should be applicable to a wide range of graph-based intermediate representations. In fact, our current Whirlwind implementation works over both control flow graphs and dataflow graphs.

10. CONCLUSION

We have presented a framework that allows modular analyses to be automatically composed and achieve mutually beneficial interactions through graph transformations. We have shown that the composed analysis terminates, is sound if the individual analyses are sound, and under a certain monotonicity condition is guaranteed to produce no worse results than running arbitrarily iterated sequences of the individual analyses (but often produces better results).

Our framework has been implemented and used successfully in the Vortex compiler, and more recently in the Whirlwind compiler. Our approach allowed us to regain modularity while still maintaining the benefits of mutually beneficial interactions. Manually simulating transformations while the analysis is running is tedious and error-prone. Manually composing analyses profitably is even harder. Using our framework, we were able to design, debug, and reason about analyses separately, while combining them profitably with little additional effort than the design of the individual parts.

Acknowledgments

This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software. We would like to thank Jeffrey Dean for his work on the implementation of the Vortex dataflow analysis engine, and Tapan Parikh for his work on an early formalization of our framework. We would also like to thank Manuvir Das, Vinod Grover, Todd Millstein and the anonymous reviewers for their useful suggestions on how to improve the paper.

11. REFERENCES

[1] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of the 7'th IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, CA, June 1992.

[2] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the Second International Static Analysis Symposium*, LNCS 983, pages 33–50, Glasgow, Scotland, September 1995. Springer-Verlag.

[3] Uwe Assmann. How to uniformly specify program analysis and transformations with graph rewrite systems. In *Proceedings of the CC'96. 6'th International Conference on Compiler Construction*, pages 121–135. Springer-Verlag, April 1996.

[4] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems*, 22(4):583–637, 2000.

[5] Craig Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993. Revised, March 1997.

[6] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report UW-CSE-96-11-02, University of Washington, November 1996.

[7] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[8] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

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

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[11] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, January 1979.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[13] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *OOPSLA'96 Conference Proceedings*, pages 83–100, October 1996.

[14] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. *SIGPLAN Notices*, 27(7):212–223, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[15] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *17th International Conference on Software Engineering*, pages 554–564, Berlin, Germany, March 1998.

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

[17] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. Fiat: A framework for interprocedural analysis and transformation. In *The Sixth Anunual Workshop on Parallel Languages and Compilers*, August 1993.

[18] Nevin Heintze. Set-based analysis of ml programs. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 306–317, Orlando, FL, June 1994.

[19] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural analysis framework for c compilers. Technical Report ACAPS Technical Memo 72, McGill University School of Computer Science, July 1993.

[20] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407, January 1995.

[21] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, June 1993.

[22] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. Technical Report UW-CSE-01-11-01, University of Washington, November 2001.

[23] Stephen P. Masticola, Thomas J. Marlowe, and Barbara G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.

[24] Greg Nelson and Derek C. Oppen. A simplifier based on efficient decision algorithms. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 141–150, January 1978.

[25] Anthony Pioli and Michael Hind. Combining interprocedural pointer analysis and conditional constant propagation. Technical Report 21532, IBM T.J. Watson Center, 1999.

[26] Vugranam C. Sreedhar, Guang R. Gao, and Yong fong Lee. A new framework for exhaustive and incremental data flow analysis using DJ graphs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 278–290, May 1996.

[27] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. *SIGPLAN Notices*, 27(7):82–

93, July 1992. Conference on Programming Language Design and Implementation.

[28] G. A. Venkatesh and Charles N. Fischer. SPARE: A development environment for program analysis algorithms. *IEEE Transactions on Software Engineering*, 18(4):304–318, April 1992.

[29] 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.

[30] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.

[31] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, January 1994.

[32] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), SIGPLAN Notices*, pages 137–146, March 1990.

[33] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[34] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

APPENDIX

A. DEFINITION OF THE SOLUTION FUNCTION S

Given an analysis $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$, a graph g and a tuple of dataflow values $\iota \in D^*$ for the input edges of g, $S_{\mathcal{A}}(g, \iota)$ is defined as follows.

First, we define the interpretation function $Int : E_g \times (E_g \to D) \to D$ as in Cousot and Cousot [10]: given an edge e and the current dataflow solution m, Int computes the dataflow value for e at the next iteration. Int is defined as:

$$Int(e, m) = \begin{cases} \iota[k] & \text{if } \exists k. e = InEdges_g[k] \\ F(n, \overrightarrow{m}(In_g(n)))[k] & \text{where } e = Out_g(n)[k] \end{cases}$$

The global flow function $FG : (E_g \to D) \to (E_g \to D)$ takes a map representing the current dataflow solution, and computes the dataflow solution at the next iteration. FG is defined as:

$$FG(m) = \lambda e. Int(e, m)$$

The global ascending flow function FGA is the same as FG, except that it joins the result of the next iteration with the current solution before returning. This ensures that the solution monotonically increases as iteration proceeds, even if F is not monotonic. FGA is defined as:

$$FGA(m) = FG(m) \mathbin{\widetilde{\sqcup}} m$$

Finally, the result of $S_{\mathcal{A}}$ is a fixed point of FGA (the least fixed point if F is monotonic):

$$S_{\mathcal{A}}(g, \iota) = \bigsqcup_{n=0}^{\infty} FGA^n(\widetilde{\bot})$$

where $\widetilde{\bot} \triangleq \lambda e.\bot$, $FGA^0 = \lambda x.x$ and $FGA^k = FGA \circ FGA^{k-1}$ for $k > 0$.

B. DEFINITION OF THE TRANSFORMATION FUNCTION T

Given a replacement function R, a graph g and some analysis results m, $T(R, g, m)$ is defined as follows. First, we introduce the update function $Update : Graph \times Node \times Graph \to Graph$, which is used to replace a single node in a graph. Given an original graph old, a node n and a replacement graph $repl$ for this node, $Update$ returns the result of replacing the node n with $repl$ in old. $Update$ is defined as follows:

$$Update(old, node, repl) = (N_{new}, E_{new}, In_{new}, Out_{new}, InEdges_{new}, OutEdges_{new})$$

where

$$\begin{aligned} N_{new} &= (N_{old} - \{n\}) \cup N_{repl} \\ E_{new} &= (E_{old} \cup E_{repl}) - \\ &\quad (Elmts(InEdges_{repl}) \cup Elmts(OutEdges_{repl})) \\ &\qquad \text{with } Elmts(tuple) = \{d \mid \exists i. tuple[i] = d\} \end{aligned}$$

$$InEdges_{new} = InEdges_{old}$$
$$OutEdges_{new} = OutEdges_{old}$$

$$In_{new}(s) = \begin{cases} In_{old}(s) & \text{if } s \in N_{old} - \{n\} \\ \overrightarrow{ReplIn}(In_{repl}(s)) & \text{if } s \in N_{repl} \end{cases}$$

$$Out_{new}(s) = \begin{cases} Out_{old}(s) & \text{if } s \in N_{old} - \{n\} \\ \overrightarrow{ReplOut}(Out_{repl}(s)) & \text{if } s \in N_{repl} \end{cases}$$

and

$$ReplIn(e) = \begin{cases} In_{old}(n)[k] & \text{if } \exists k. e = InEdges_{repl}[k] \\ e & \text{otherwise} \end{cases}$$

$$ReplOut(e) = \begin{cases} Out_{old}(n)[k] & \text{if } \exists k. e = OutEdges_{repl}[k] \\ e & \text{otherwise} \end{cases}$$

We now define $Update_\epsilon$, a simple extension to $Update$ that works correctly if the replacement graph is ϵ:

$$Update_\epsilon(g, n, r) = \begin{cases} g & \text{if } r = \epsilon \\ Update(g, n, r) & \text{otherwise} \end{cases}$$

The graph returned by $T(R, g, m)$ is then simply the iterated application of $Update_\epsilon$ on all the nodes of g. Thus, $T(R, g, m)$ is defined by:

$$T(R, g, m) = IT(R, g, m, N_g)$$

where IT (which stands for $IteratedT$) is:

$$IT(R, g, m, N) = \begin{cases} IT(R, g_{new}, m, N - \{n\}) & \text{if } \exists n \in N \\ g & \text{if } N = \emptyset \end{cases}$$

with

$$g_{new} = Update_\epsilon(g, n, R(n, \overrightarrow{m}(In_g(n))))$$