Efficient Computation of Fixpoints that Arise in Complex Program Analysis*

Li-Ling Chen
University of Illinois

Williams Ludwell Harrison III†
University of Illinois

Kwangkeun Yi‡
AT&T Bell Laboratories

Abstract

This paper proposes a model for studying the computation of fixpoints that arise in complex program analysis based on *abstract interpretation*, and presents an efficient algorithm for computing fixpoints based on the model. Abstract interpretation provides a unified framework for handling interprocedural analysis of programs with unrestricted pointer manipulation, higher-order functions, and continuations. In the general case, the structure of the fixpoint computation is not known before analysis; that is, the flow graph cannot be determined from the syntax of a program. Abstract interpretation handles this situation easily.

In our algorithm, the *entailment graph*, representing the structure of the fixpoint computation, is developed during analysis. The strategies for determining the evaluation order, which underlie the algorithm, are described. Based on the strategies, local knowledge of the *entailment graph* at each node is exploited to determine dynamically an effective order of evaluation. The algorithm behaves like those based upon interval analysis of a flow graph, but without requiring the interprocedural control flow graph to be given a priori. The algorithm has been implemented, and experiments have been conducted to compare it to other iterative algorithms for solving such problems. The results show that the algorithm is flexible, efficient, and consistently better than the others.

Index Terms: fixpoint computation, abstract interpretation, interprocedural analysis, efficient dataflow analysis, and dynamic flow graph.

To appear in The Journal of Programming Languages, Chapman & Hall, U.K.

*This project is supported by the U.S. Department of Energy under contact DE-FG02-85ER25001 and the IBM Corporation.

†llchen@csrd.uiuc.edu, Center for Supercomputing Research and Development, Urbana, IL 61801, U.S.A.

‡kwang@research.att.com, AT&T Bell Laboratories, Murray Hills, NJ 07974, U.S.A.

1 Introduction

Abstract interpretation [1, 2, 3] is a powerful and promising framework for constructing program analyses, especially in the presence of difficult language constructs, such as pointers [4], higher-order functions [5, 6], and continuations; an example analysis is given in Section 3.3. In contrast, conventional dataflow analysis methods have not adapted well to these language constructs. It is unclear how conventional dataflow analysis handles higher-order functions and continuations. As to pointers, conventional dataflow analysis is often restricted to limited pointer manipulation (e.g., no arithmetic operations on pointers). Unfortunately, abstract interpretation often gives rise to complex functionals whose least fixpoints (solutions to analysis problems), if approached naively, are very expensive to compute. To be practical, it is necessary to find efficient algorithms for computing these least fixpoints.

There are several methods for solving for a related but simpler class of fixpoint problems efficiently in conventional dataflow analysis. In these problems, monotonic functions are associated with the edges of a control flow graph [7]. These functions are initially unknown; they are the variables to be solved for in the dataflow problem. They are given an initial value and are computed iteratively [8, 9, 10] or by elimination [11, 12, 13] until they are stable (that is, until a fixpoint is reached). Because the relation between the unknowns is given as an input to the problem in the form of the control flow graph, it is possible to solve for the unknowns efficiently. This is the purpose of interval analysis [14]: to consider the unknowns of the dataflow problem in an order that will allow their final values to be determined most quickly; the order can be obtained by examination of the control flow graph, which is therefore a prerequisite for conventional dataflow analysis methods.

In the general case of abstract interpretation, we do not know the structure of the fixpoint computation (the relation among the unknowns) a priori. For programs with higher-order functions (e.g., pointers to functions in C and lambda expressions in Lisp), it is not always possible to determine the flow graph by examining the syntax of a program. Rather, the structure emerges only as the fixpoint itself is computed; that is, the flow graph is determined while analysis proceeds. In this setting, the methods obtained for dataflow analysis which require the control flow graph as a prerequisite are not applicable. In [16], it is pointed out that higher-order functions are the major obstacle for program analysis of functional programming languages because the flow graphs for performing dataflow analysis depend on the results of dataflow analysis.

For instance, the examples in Figures 1 and 2 have flow graphs (like the interprocedural control flow graphs of [17]) that cannot be precisely determined before analysis. Figure 1 is an example that we may imagine as the invocation of virtual functions in C++; the control flow graph of this program, which depends on the flow analysis information of x and i, is not evident from its syntax. In the example of Figure 2, X and Y are procedure-valued variables, which are passed by reference as actual arguments and modified in the procedure f; therefore, the procedures they refer to are not obvious without some extent of analysis.

$A[1] := \&f_1$
$\cdots$
$A[n] := \&f_n$
$\cdots$
$p := A$
call $A[x-7]$
call $*(p+i)$

Figure 1: Function pointers

procedure $f(x, y)$ $*x := y$;
procedure $g(z)$ call $z(\ )$;
main( )
Procedure Variable X, Y
call $f(\&X, \&f)$;
call $X(\&Y, \&g)$;
call $Y(\&g)$;
end

Figure 2: Procedure-valued variables

Our work was motivated by the abstract interpretation used in MIPRAC [18] and the analyzer generator Z1 [19]. MIPRAC analyzes and compiles an intermediate language, MIL, which includes first-class functions, dynamic allocation, and unrestricted pointer manipulation. Fortran, C, and Scheme programs can be transformed into MIL. The compiler has successfully performed complex analyses, for example, side effects and object life time. However, its efficiency is poor because of the time spent computing the least fixpoint. Z1 [19], an analyzer generator based on MIPRAC, provides a specification language for users to describe an analysis, and automates the generation and management of semantics-based, interprocedural program analysis. The method for fixpoint computation presented in this paper is applicable to any abstract interpretation over domains of finite height. This includes complex analyses like those of MIPRAC, as well as those written using Z1 (the method is being used in Z1), and analyses like those found in [3, 20]

In this paper we first propose an entailment model for solving fixpoint problems that arise in abstract interpretation in the presence of unlimited pointer manipulation, dynamic allocation, and higher-order functions, and then present an algorithm to efficiently compute the fixpoints based on the entailment model. Our algorithm makes use of the *entailment relation* of a program as guidance for dynamically ordering the evaluations as analysis proceeds. We call the strategies for determining an effective order, *waiting for all successors, leading node first, suspended evaluation, and back edge first.* It takes advantage of the principles of *minimal function graph* presented in [22] and dynamic ordering based on the local knowledge of the *entailment graph* to minimize redundant work. Our algorithm behaves like those based upon interval analysis of a flow graph when applied to a problem in the domain of interval analysis, but without requiring the control flow graph to be given a priori. Experiments have been conducted to demonstrate that the guided-entailment algorithm is efficient, flexible, and consistently better than the others that are applicable to the same problem.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 introduces abstract interpretation and least fixpoints, and gives an example analysis in the presence of pointers to locations and pointers to procedures, based on abstract interpretation. Section 4 defines *entailment* relation and gives a basic algorithm for computing least fixpoints based on the entailment relation. Section 5 presents the strategies for determining an effective order and the guided-entailment

model for efficient fixpoint computation. The guided-entailment algorithm based on these strategies for dynamic ordering is given in Section 6. Section 7 describes and discusses the experimental results. Section 8 presents the conclusions.

2 Related Work

Control flow graphs are commonly used in dataflow analysis [23, 7], but are inefficient in the computation of analyses. To alleviate the inefficiency, def-use chains, which allow to avoid propagating information through irrelevant nodes, were proposed as a means to improve the performance of program analysis [7]. However, def-use chains are limited to analyses of relatively simple variable references. For example, the static single assignment (SSA) form [24] is applicable primarily to scalar variables with transparent reference properties (i.e., no aliasing or pointers). The presence of pointers and higher-order functions greatly complicates the obtaining of def-use chains. We cannot use the def-use information to accelerate the analysis, because for the simple reason that the def-use information is precisely what we want to obtain as an output from the analysis.

Many representations have been proposed to cope with the inefficiency of control flow graphs. A program dependence graph [25, 15, 26] represents both data and control dependences. A dependence flow graph [27], combining the virtues of many representations, provides a representation for rapid traversal to gather the dependence information and itself is a program that can be executed. A sparse evaluation graph [28] directly connects the nodes that generate and use information; it is designed for forward and backward dataflow problems with the advantages of the SSA form, without using bit vectors. Basically, these representations incorporate the def-use information and propagate information through only relevant nodes. Therefore they have the same limitations as def-use chains. It is not obvious how to represent a general class of programs with complicated constructs for accessing memory, by using these forms.

Several methods have been proposed for computing the fixpoints that arise in abstract interpretation. In particular, the frontiers algorithm presented in [29, 30, 31] is an efficient means of computing analysis problems over binary lattices, for example, strictness analysis [32]. However, this algorithm is suitable only for the binary lattice; for general lattices, it is unclear how to define frontiers. In addition, the minimal function graph [22] was proposed to handle the class of applicative programs where the information of interest is limited to the set of values a function may be called with in the program being analyzed (as opposed to the complete set of possible input values). To our knowledge, how to improve the efficiency of abstract interpretation in general cases in practice has not yet been addressed.

3 Program Analysis Framework

This section introduces abstract interpretation and least fixpoints, and gives an example to illustrate the idea of abstract interpretation in the presence of pointers to locations and pointers to locations.

3.1 Abstract Interpretation

Abstract interpretation is a semantics-based approach to static program analysis [1, 2, 3]. The basic idea is as follows. Every programming language has a concrete interpretation (semantics) that gives a meaning to a program and defines the correctness of subsequent abstract semantics. Using the concrete semantics, it is able to answer certain questions of interest about programs correctly, for example, constants, side effects, and object lifetimes; but the concrete semantics is not computable at compile time. From the concrete semantics, we form an abstract interpretation of the language which answers the questions approximately, but which is appropriate for compile-time analysis. That is, for a certain property we are interested in, we can find information without having the exact answer or doing the whole calculation (of the concrete semantics).

In short, abstract interpretation "simulates" program execution over abstract domains, representing information of interest at compile time. This gives a general framework for handling interprocedural, complicated analysis in a simple manner because procedure boundaries are eliminated. Many techniques based on control-flow and data-flow analysis can be expressed as abstract interpretations. In addition, we can tune the precision and complexity of a program analysis based on abstract interpretation by controlling the abstraction maps it uses. Abstract interpretation also provides a basis for semantically correct transformation [33].

Under abstract interpretation, an analysis problem is associated with a monotonic [1] function (an abstract interpreter) $\overline{I}$ that gives an abstract semantics to the expressions (statements or basic blocks) in a program. We begin with $\overline{I}$, expressed as a recursive function. To solve the analysis problem is to compute an equivalent non-recursive form of $\overline{I}$. In general, an abstract interpreter is designed as a general template for the whole language to be analyzed. For a particular input program, the interpreter will be instantiated to give a particular interpretation function.

Let Σ be the set of expressions in a program. $\overline{X}$ and $\overline{Y}$ denote prestates and poststates respectively. To ensure the termination of an analysis, we assume $\overline{X}$ and $\overline{Y}$ are domains of finite height.

$$\overline{I} : \Sigma \rightarrow \overline{X} \rightarrow \overline{Y}$$

That is, $\overline{I}$ maps an expression and a prestate to a poststate. For example, $\overline{X}$ and $\overline{Y}$ might simply be abstract stores (abstracted from the store in concrete interpretation), and the semantic function $\overline{I}$ transforms one store to another store according to the meaning of each program expression.

[1] A function $f : A \rightarrow B$ is monotonic iff $\forall x, y \in A,\ x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$.

The equivalent non-recursive form (the direct form, e.g., a table of inputs and outputs) of the semantic function $\overline{I}$ is undefined (unknown) before analysis; that is, $\lambda n.\lambda x.\perp_{\overline{Y}}$. As the analysis proceeds, information is collected and the direct form of $\overline{I}$ is defined incrementally from the undefined function to its final value; that is, the semantic function is approximated by successive functions until stable. Therefore, a semantic functional $\mathcal{F}$ that maps one approximation in $\Sigma \to \overline{X} \to \overline{Y}$ to another is employed to compute the direct form of $\overline{I}$, which is the least fixpoint of the functional $\mathcal{F} : (\Sigma \to \overline{X} \to \overline{Y}) \to (\Sigma \to \overline{X} \to \overline{Y})$. $\mathcal{F}$ might be written as

$$
\begin{array}{l}
\mathcal{F} = \lambda I.\lambda\sigma.\lambda x.\ \text{case } \sigma \text{ of} \\
\qquad [\![call]\!]\colon\ g_1(I,\sigma,x) \\
\qquad [\![if]\!]\colon\ g_2(I,\sigma,x) \\
\qquad [\![+]\!]\colon\ g_3(I,\sigma,x) \\
\qquad\quad \cdots \\
\quad \text{endcase}
\end{array}
$$

The definitions of the g_i's depend upon the semantics of the language we analyze and the abstract domains. Each g_i is abstracted from the corresponding concrete interpretation function for each type of expression. All g_i's are monotonic. In general, the current approximation I is recursively called in g_i which computes a new postcontext for node σ with prestate x (i.e., $g_i(\sigma, x) = \cdots I(\sigma', x') \cdots$). For examples of g_i's, see the example in Section 3.3.

3.2 Least Fixpoints

Typically, the least fixpoint of a functional $\mathcal{F} : (X \to Y) \to (X \to Y)$ is computed iteratively, where X and Y are domains of bounded height and $\mathcal{F}$ is monotonic. A domain is a complete partial ordering and the function space $X \to Y$ is the set of all continuous functions that map X to Y. It can be shown that the least fixpoint of the functional $\mathcal{F}$ exists [34]. We begin with an initial approximation $f_0 = \lambda x.\perp_Y$ where $\perp_Y$ is the least element of Y. $\mathcal{F}$ is repeatedly applied to the current approximation until it produces an approximation the same as the previous one; that is,

$$
\begin{array}{l}
\mathcal{F}^0\perp = f_0 = \lambda x.\perp_Y \\
\mathcal{F}^1\perp = f_1 = \mathcal{F}(f_0) \\
\quad \cdots \\
\mathcal{F}^i\perp = f_i = \mathcal{F}(f_{i-1}) \\
\quad \cdots \\
\mathcal{F}^n\perp = f_n = \mathcal{F}^{n-1}\perp
\end{array}
$$

The approximations $\mathcal{F}^i\perp$'s form an Ascending Kleene Chain [34]; that is, $\forall i \ \ \mathcal{F}^i\perp \sqsubseteq \mathcal{F}^{i+1}\perp$. Formally, the least fixpoint, $fix\ \mathcal{F}$, is defined as $\bigsqcup\{\mathcal{F}^i\perp | i \geq 0\}$ where $\bigsqcup$ is the least upper bound operation on the set of functions in $X \to Y$. If $f_n = f_{n+1}$, then the least fixpoint $fix\ \mathcal{F} = f_n$.

Example 1 To illustrate how to compute the least fixpoint, we give a simple example. $f : \{a, b, c\} \to 2^{\{a,b,c\}}$ is a recursive function. To compute the final values for its possible

arguments, a functional $\mathcal{F}$ is used.

$$
\begin{array}{ll}
f = \lambda x.\ \text{case } x \text{ of} & \mathcal{F} \equiv \lambda f'.\lambda x.\ \text{case } x \text{ of} \\
\quad a : \{a\} \cup f(b) & \quad a : \{a\} \cup f'(b) \\
\quad b : \{b\} \cup f(c) \quad \Longrightarrow & \quad b : \{b\} \cup f'(c) \\
\quad c : \{c\} \cup f(c) & \quad c : \{c\} \cup f'(c) \\
\quad \text{endcase} & \quad \text{endcase}
\end{array}
$$

The computation of the least fixpoint of $\mathcal{F}$ is as follows. $\emptyset$ is the least element of $2^{\{a,b,c\}}$.

	f_0	f_1	f_2	f_3	f_4
a	$\emptyset$	$\{a\}$	$\{a,b\}$	$\{a,b,c\}$	$\{a,b,c\}$
b	$\emptyset$	$\{b\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$
c	$\emptyset$	$\{c\}$	$\{c\}$	$\{c\}$	$\{c\}$

$f_3 = f_4$; that is, the least fixpoint $fix\ \mathcal{F} = f_3$. □

Example 2 This example demonstrates the computation of the least fixpoint of an analysis problem. Consider the definedness problem for the following program segment; that is, we want to know which variables may be defined after a statement (e.g., procedure call) is executed. To simplify the computation, suppose there are no local variables and no parameters for the procedures P, Q, and R, and there are only two global variables, a and b.

```
int a, b
procedure P()
    begin
    if (···) then                (n0)
        call Q();                (n1)
    else call R();               (n2)
    end
procedure Q() begin b := a; end
procedure R() begin a := b + 1; end
```

The abstract semantic function $\overline{I}$ associated with this problem will map a set ($\rho \in 2^V$) of defined variables to another set in 2^V for each statement; that is, $\overline{I} : N \to 2^V \to 2^V$, where V is the set of variables (here is $\{a, b\}$) and N is the set of statements in a program.

$\mathcal{F} = \lambda I.\lambda n.\lambda \rho.$ case n of
  $[\![v := const]\!]$: return $\{v\} \cup \rho$
  $[\![v := w]\!]$:
    if $(Vars(w) \subseteq \rho)$ then return $\{v\} \cup \rho$ else return $\rho - \{v\}$
  $[\![if\ e_1\ then\ e_2\ else\ e_3]\!]$:
    $\rho_1 \leftarrow I([\![e_1]\!],\rho)$
    $\rho_2 \leftarrow I([\![e_2]\!],\rho)$
    return $\rho_1 \cup \rho_2$
  $[\![$`begin` $n_1, \cdots, n_k$ `end`$]\!]$:
    $\rho_0 \leftarrow \rho$
    $\rho_i \leftarrow I(n_i, \rho_{i-1})\ for\ \ i = 1\ to\ k$
  $[\![$`call` $g()]\!]$: $I([\![Bodyof(g)]\!],\rho)$
  ...
endcase
where $Vars(w)$ returns the set of variables appearing in the expression w.

Suppose we want to know the set of defined variables for the procedure P in a direct form. For this particular problem, the abstract semantic function is defined as follows.

$\mathcal{F} = \lambda I.\lambda n.\lambda\rho.$ case n of
  $[\![$`call` $P()]\!]$: /* that is, evaluate n_0 */
    $\rho_1 \leftarrow I([\![$`call` $Q()]\!],\rho)$
    $\rho_2 \leftarrow I([\![$`call` $R()]\!],\rho)$
    return $\rho_1 \cup \rho_2$
  $[\![$`call` $Q()]\!]$: return $I([\![b := a]\!],\rho)$ /* n_1 */
  $[\![$`call` $R()]\!]$: return $I([\![a := b + 1]\!],\rho)$ /* n_2 */
endcase

The least fixpoint is computed using the typical iterative method shown previously. We will show the approximate functions for n_0, n_1, and n_2. Initially, at each statement, the semantic function $\overline{I}_0$ maps every input to $\emptyset$, the least element of 2^V. Next, apply $\mathcal{F}$ to the initial approximation $\overline{I}_0$ to obtain the second approximation $\overline{I}_1$:

n_0: $\{\emptyset \mapsto \emptyset,\ \{a\} \mapsto \emptyset,\ \{b\} \mapsto \emptyset\ ,\{a,b\} \mapsto \emptyset\}$
n_1: $\{\emptyset \mapsto \emptyset,\ \{a\} \mapsto \{a,b\},\ \{b\} \mapsto \emptyset, \{a,b\} \mapsto \{a,b\}\}$
n_2: $\{\emptyset \mapsto \emptyset\ ,\ \{a\} \mapsto \emptyset,\ \{b\} \mapsto \{a,b\}, \{a,b\} \mapsto \{a,b\}\}$

Then, apply $\mathcal{F}$ to the second approximation $\overline{I}_1$. The results for n_1, n_2 remain the same in the third approximation $\overline{I}_2$; for n_0, we have

n_0: $\{\emptyset \mapsto \emptyset,\ \{a\} \mapsto \{a,b\},\ \{b\} \mapsto \{a,b\}, \{a,b\} \mapsto \{a,b\}\}$.

Application of $\mathcal{F}$ to $\overline{I}_2$ will not raise its value, so the least fixpoint is reached. □

To turn the fixpoint computation into an algorithm for program analysis based on abstract interpretation, we use two tables, $T_{\overline{X}} : \Sigma \to \overline{X}$ and $T_{\overline{Y}} : \Sigma \to \overline{Y}$, to represent the prestate and poststate for each expression respectively. In a monotonic analysis framework, we need keep only the highest prestate and poststate for each expression. The two tables together represent the current approximation $I : \Sigma \to \overline{X} \to \overline{Y}$ of $\overline{I}$. Given a semantic functional $\mathcal{F}$ of an analysis problem, the least fixpoint (i.e., the solution to the analysis problem) can be computed in this way:

$FIXPOINT$::
$\forall\ i \in \Sigma \quad T_{\overline{X}}(i) = \bot_{\overline{X}} \quad T_{\overline{Y}}(i) = \bot_{\overline{Y}}$
repeat
  $OldT_{\overline{X}} \leftarrow T_{\overline{X}}$
  $OldT_{\overline{Y}} \leftarrow T_{\overline{Y}}$
  $\langle T_{\overline{X}},\ T_{\overline{Y}}\rangle \leftarrow$ *Apply* $\mathcal{F}$ $\langle OldT_{\overline{X}},\ OldT_{\overline{Y}}\rangle$
until ($\langle OldT_{\overline{X}},\ OldT_{\overline{Y}}\rangle = \langle T_{\overline{X}},\ T_{\overline{Y}}\rangle$)

Whenever a prestate or a poststate is improved, the semantic functional $\overline{I}$ will be applied to the current approximation again until none of the prestates and poststates are changed.

```
main( )
Global Variable int c, *u
Procedure Variable a, b, arr[4]
    begin
        n1:    u = &c;
        n2:    arr[0] = P;   arr[1] = Q;
        n3:    arr[2] = R;   arr[3] = T;
        n4:    *u = 0;
        n5:    if (c < 0) then  c := 0 else c := c + 1;
        n6:    call P(&a, arr[2]);
        n7:    call Q(a);
        n8:    call P(&b, P) ;
        n9:    call b(&a, *(arr + c)) ;
        n10:   call a(T) ;
    end

procedure P(x, y)
    begin   *x := y;   end
procedure Q(z)
    begin
    n11:    ··· := c;
    n12:    call z( );
    n13:    ··· := c;
    end
procedure R( )
    begin
    n14 :   c := 1;
    end
procedure T( )
    begin
    n15:    c := 2;
    end
```

Figure 3: An example program of pointers and higher-order procedures.

3.3 Example Analysis based on Abstract Interpretation

We will illustrate the idea of abstract interpretation in the presence of pointers and higher-order procedures using the example program in Figure 3, which is similar to that defined in [35, 36]. The program is written in a simple language that permits pointers to variables and pointers to procedures, but little else. In particular, for simplicity, we assume that the language does not support recursion or structures, and that every variable has a distinct identifier. In other words, the memory locations known to the program are finite and each is accessed by an index. For more complex examples, see [19, 4, 20, 3]. Suppose the anlysis we want to perform is *constant propagation*; that is, we want to know whether a variable (c, in the example of Figure 3) is constant at a program point. The concrete interpretation and abstract interpretation for this analysis problem are described next.

3.3.1 Concrete Interpretation

The concrete semantic function I associated with this problem maps a store $s \in S$ to another s' for each expression in the program; that is, $I : N \rightarrow S \rightarrow S$. A store maps a location $l \in Loc$ to a denotable value $v \in Val$, where Loc is the set of locations in a store, which can be simply integers as indices to the store. The denotable values are Loc, Z, and $Proc$; Z represents integers and $Proc$ is the set of defined procedures (like closures, but in the case of this simple language, closures without environments). In order to treat pointers, Loc is also a denotable value in the store. In fact, there is an environment that maps an identifier to a location (i.e., $Id \rightarrow Loc$) where Id is the set of identifiers in a program. An environment can be very complicated, depending on the language and the desired analysis. Here we simply use $addr$ to return the address of a variable (i.e., a location in the store).

$$
\begin{array}{rcll}
I & : & N \times S \rightarrow S & /* \text{ interpretation function } */ \\
S & = & Loc \rightarrow Val & /* \text{ interpretation store } */ \\
Val & = & Loc + Z + Proc & /* \text{ denotable values in } S \ */
\end{array}
$$

The concrete interpreter I for the language is defined as follows.

$I = \lambda n.\lambda s.$ case n of
$[\![v := w]\!]$: if $(v \in Id)$ then $s(addr(v)) \leftarrow Eval(w, s)$
$[\![*v := w]\!]$: if $(v \in Id)$ and $(Eval(v, s) \in Loc)$ then $s(Eval(v, s)) \leftarrow Eval(w, s)$
$[\![A[e] := w]\!]$: if $(A \in Id)$ and $(Eval(A + e, s) \in Loc)$ then $s(Eval(A + e, s)) \leftarrow Eval(w, s)$
$[\![begin\ \ n_1, \cdots, n_k\ \ end]\!]$:
$s_0 \leftarrow s$
$s_i \leftarrow I(n_i, s_{i-1})\ for\ \ i = 1\ to\ k$
$[\![if\ \ a\ \ then\ \ b\ \ else\ \ c)]\!]$:
if $(EvBool(a))$ then $I(b, I(a, s))$ else $I(c, I(a, s))$
$[\![\texttt{call}\ x(a_1, \cdots, a_k)]\!]$:
if $(x \in Id)$ and $(s(addr(x)) \in Proc)$ then
$I([\![\texttt{call}\ s(addr(x))(a_1, \cdots, a_k)]\!], s)$ /* x is a procedure variable */
$[\![\texttt{call}\ g(a_1, \cdots, a_k)]\!]$:
$\langle l_1, \cdots, l_k \rangle \leftarrow Formals(g)$
$s_0 \leftarrow s$
$s_i \leftarrow I([\![l_i := a_i]\!], s_{i-1})\ for\ i = 1$ to k /* a_i is assigned to l_i using I */
$s' \leftarrow I(BodyOf(g), s_k)$ /* the body of the function is interpreted */
$s' \leftarrow s'[\perp/l_1, \cdots, \perp/l_k]$ /* formal parameters are removed from the store*/
...
endcase

$Eval = \lambda\mu.\lambda s.$ case μ of
$[\![t]\!]$: if $(t \in Z + Proc)$ then t
else if $(t \in Id)$ then $s(addr(t))$
$[\![*t]\!]$: if $(t \in Id)$ and $(s(addr(t)) \in Loc)$ then $s(s(addr(t)))$
$[\![\&t]\!]$: if $(t \in Id)$ then $addr(t)$
$[\![A[e]]\!]$: if $(A \in Id)$ then $s(s(addr(A)) + Eval(e, s))$
$[\![*(A + e)]\!]$: if $(A \in Id)$ then $s(Eval(A + e, s))$
$[\![e_1 + e_2]\!]$: $Eval(e_1, s) + Eval(e_2, s)$
$[\![e_1 - e_2]\!]$: $Eval(e_1, s) - Eval(e_2, s)$
...
endcase

The right hand side of an assignment statement will be evaluated first using $Eval$, and its value is assigned to the l-value of the left hand side. $Eval$ also evaluates arithmetic expressions. $*v$ means the content of the location which is pointed by v. $A[e]$ denotes the content of the location pointed by A with offset e; in fact, it is equivalent to $*(A + e)$. "begin" expression is a block-like structure that causes its subexpressions to be evaluated from left to right. When a procedure g is called with a store s, a formal parameter l_i is assigned a_i using I; then the body of g is evaluated. After g is returned, those formal parameters of g will not be in the store any more. *Formals(g)* is an ordered set of the formal parameters of g. $EvBool$ evaluates a boolean expression.

3.3.2 Abstract Interpretation

Now, abstract interpretation is simply an approximation of the concrete interpretation. The abstract interpreter transforms an abstract store to another abstract store, mapping Loc to $\hat{Val}$. The abstract

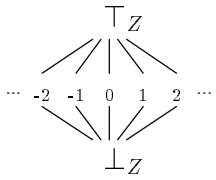

Figure 4: Abstract domain L_Z for integers.

denotable values are $\hat{Loc}$, $\hat{Proc}$, and $\hat{Z}$.

$$
\begin{array}{rcll}
\hat{I} & : & N \times \hat{S} \to \hat{S} & \text{/* abstract interpretation function */} \\
\hat{S} & = & Loc \to \hat{Val} & \text{/* abstract store */} \\
\hat{Val} & = & \hat{Loc} + \hat{Z} + \hat{Proc} & \text{/* abstract denotable value */} \\
\hat{Z} & = & L_Z & \text{/* abstract integer */} \\
\hat{Loc} & = & 2^{Loc} & \text{/* abstract location */} \\
\hat{Proc} & = & 2^{Proc} & \text{/* abstract procedure */}
\end{array}
$$

2^{Loc} is the power set of Loc, in which the top element is Loc and the least element is $\emptyset$; similarly for 2^{Proc}. "$\sqcup$" for them is set union. $\hat{Z}$ is a flat domain whose least element is $\bot_Z$ and whose top element is $\top_Z$ (see Figure 4). $\bot_Z$ means unknown and $\top_Z$ means not a constant. Initially, all variables are assigned $\bot_Z$; therefore, the analysis of constant propagation here is an optimistic approach.

The abstract interpreter is similar to the concrete one except that an abstract denotable value is a set of locations or procedures, or an element from a flat domain. For example, if a procedure-valued variable is called, all procedures it refers to are called and the resulting stores are joined. If a pointer is used, we need to deal with all the possible locations it may reference. In this example, the index of an array (e.g., 2 for $arr[2]$ at n_6 and c for $*(arr + c)$ at n_9) can be determined at compile time (taking advantage of constant propagation); therefore, only one location is treated (since "arr" points to only one array in this case). If the index cannot be determined during analysis (i.e., $\top_Z$), all the locations of an array referred to by the pointer are considered. Details of alias anlysis are beyond the scope of this paper.

The principal change to the concrete interpreter is shown in Figure 5.

$Range(loc)$ returns the size of the object starting at loc. For the example in Figure 3, A is an array of size 4; $Range(s(addr(A)))$ returns 4. "$\hat{+}$" is abstracted from "+," in which $\top_Z \hat{+} x = \top_Z$ and $\bot_Z \hat{+} x = \bot_Z$; similarly for $\hat{EvBool}$.

$$
\begin{array}{l}
\hat{Eval} = \lambda\mu.\lambda s.\ \text{case } \mu \text{ of} \\
\quad [\![*t]\!]\text{: if } (t \in Id) \text{ and } (s(addr(t)) \in \hat{Loc}) \text{ then} \\
\qquad\qquad \bigsqcup_{d_j \in s(addr(t))} s[d_j] \\
\quad [\![e_1 + e_2]\!]\text{: } \hat{Eval}(e_1, s) \hat{+} \hat{Eval}(e_2, s) \\
\qquad \cdots \\
\text{endcase}
\end{array}
$$

Figure 6 lists the abstract execution of the example program step by step using $\hat{I}$; abstract interpretation basically " executes" a program over abstract domains and gathers information as analysis proceeds. The store after a statement is executed during analysis is shown. Initially, the store is unknown, that

$\hat{I} = \lambda n.\lambda s.$ case n of

$[\![$`call` $x(a_1, \cdots, a_k)]\!]$:

if $(x \in Id)$ and $(s(addr(x)) \in \hat{Proc})$ then /* x is procedure-valued variable */

$\bigsqcup_{f_j \in s(addr(x))} \hat{I}(f_j, s)$

$[\![*v := w]\!]$:

if $(v \in Id)$ and $(s(addr(v)) \in \hat{Loc})$ then /* v is a pointer */

$\bigsqcup_{d_j \in s(addr(v))} s[\hat{Eval}(w)/d_j]$

$[\![*(p+e) := w]\!]$: /* equivalent to $p[e] := w$ */

case $\hat{Eval}(e)$ of /* evaluate e first */

$\perp_Z : \perp$ /* if unknown, return unknown */

$const : \bigsqcup_{d_j \in s(addr(p))} s[\hat{Eval}(w)/(d_j + \hat{Eval}(e))]$ /* if constant, the offset is fixed */

$\top_Z : \bigsqcup_{d_j \in s(addr(p))} \bigsqcup_{i < Range(d_j)} s[\hat{Eval}(w)/(d_j + i)]$

endcase

$[\![$ *if a then b else c*$)]\!]$:

case $\hat{EvBool}(a)$ of /* the condition of "*if*" is evaluated first */

$\perp : \perp$ /* if unknown, return unknown */

$true : \hat{I}(b, \hat{I}(a, s))$ /* if true, take the "true" branch */

$false : \hat{I}(c, \hat{I}(a, s))$ /* if false, take the "false" branch */

$\top : \hat{I}(b, \hat{I}(a, s)) \sqcup \hat{I}(c, \hat{I}(a, s))$ /* otherwise, join the results from two branches */

endcase

$\cdots$

endcase

Figure 5: Abstract semantic function

is, $\lambda l.\perp$. "enter" and "return" are added to show a procedure call. The formal parameters are assigned actual arguments when "enter" is listed, and removed from the abstract store when "return" is listed. If a procedure has no arguments, we simply show the evaluation of its body; that is, "enter" and "return" are not listed. We use P to represent the procedure name and the procedure itself. Since the array "arr" is not modified after initialization, we simply use arr_i to represent the location of $arr[i]$; the locations for the array pointer by arr are not listed in the store.

$T_{\overline{Y}}$ is used to record the analysis information for each node. Originally, each node is associated with a store, mapping Loc to $\hat{Val}$, and when a node n is evaluated with store s, $T_{\overline{Y}}(n) \leftarrow T_{\overline{Y}}(n) \sqcup \mathcal{F}I(n, s)$. To show the desired results more clearly, we combine s and env together and let $T_{\overline{Y}}(n) : Id \rightarrow \hat{Val}$, which gives the abstract denotable value of an identifier at node $n \in N$. Suppose we are interested primarily in c and z, of which the analysis information is shown at the end of the table in Figure 6. From the listed stores, it is easy to conclude that c at n_{11} (marked with $\circ$) is a constant, but c at n_{13} (marked with $\bullet$) is not a constant because the values for c are possibly 1 or 2 (i.e., $\top_Z$). We can also conclude that there are edges from Q to R and to T in the call graph from the information of z at node n_{12} (marked with $\star$).

	Abstract Execution	Abstract Store							Analysis Information
		a	b	c	u	x	y	z	
	begin	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	/* initial store */
	$n_1 : u = \&c$	$\perp$	$\perp$	$\perp$	$\&c$	$\perp$	$\perp$	$\perp$	/* u points to c */
	$n_2 : arr[0] = P; \quad arr[1] = Q$								/* $s(arr_0) \leftarrow P$, $s(arr_1) \leftarrow Q$ */
	$n_3 : arr[2] = R; \quad arr[3] = T$								/* $s(arr_2) \leftarrow R$, $s(arr_3) \leftarrow T$ */
	$n_4 : *u = 0$	$\perp$	$\perp$	0	$\&c$	$\perp$	$\perp$	$\perp$	/* $*u$ is assigned 0 */
	$n_5 : if\ (c < 0)\ then\ \ c := 0\ else\ c := c + 1$	$\perp$	$\perp$	1	$\&c$	$\perp$	$\perp$	$\perp$	/* the branch $c := c + 1$ is taken */
	$n_6 :$ call $P(\&a, arr[2])$								
	enter	$\perp$	$\perp$	1	$\&c$	$\&a$	R	$\perp$	/* $s(arr_2) = \{R\}$ */
	$*x := y;$	R	$\perp$	1	$\&c$	$\&a$	R	$\perp$	
	return	R	$\perp$	1	$\&c$	$\perp$	$\perp$	$\perp$	
	$n_7 :$ call $Q(a) \equiv$ call $Q(R)$								
	enter	R	$\perp$	1	$\&c$	$\perp$	$\perp$	R	
$\circ$	n_{11}: **use of** c	R	$\perp$	**1**	$\&c$	$\perp$	$\perp$	R	$T_{\overline{Y}}(n_{11}, c) = 1$
$\bullet$	n_{12} : **call** $z \equiv$ **call** R	R	$\perp$	1	$\&c$	$\perp$	$\perp$	R	$T_{\overline{Y}}(n_{12}, z) = \{R\}$
	n_{14} : $c := 1;$	R	$\perp$	1	$\&c$	$\perp$	$\perp$	R	/* c is assigned 1 */
$\star$	n_{13} : **use of** c	R	$\perp$	**1**	$\&c$	$\perp$	$\perp$	R	$T_{\overline{Y}}(n_{13}, c) = 1$
	return	R	$\perp$	1	$\&c$	$\perp$	$\perp$	$\perp$	
	$n_8 :$ call $P(\&b, P)$								
	enter	R	$\perp$	1	$\&c$	$\&b$	P	$\perp$	
	$*x := y;$	R	P	1	$\&c$	$\&b$	P	$\perp$	
	return	R	P	1	$\&c$	$\perp$	$\perp$	$\perp$	
	$n_9 :$ call $b(\&a, *(arr + 2))$								
	enter	R	P	1	$\&c$	$\&a$	Q	$\perp$	/* $*(arr + 2) = \{Q\}$ */
	$*x := y;$	Q	P	1	$\&c$	$\&a$	Q	$\perp$	
	return	Q	P	1	$\&c$	$\perp$	$\perp$	$\perp$	
	n_{10} :call $a(T) \equiv$ call $Q(T)$								
	enter	Q	P	1	$\&c$	$\perp$	$\perp$	T	
$\circ$	n_{11} : **use of** c	Q	P	**1**	$\&c$	$\perp$	$\perp$	T	$T_{\overline{Y}}(n_{11}, c) = 1 \sqcup 1 = 1$
$\bullet$	n_{12} : **call** $z \equiv$ **call** T	Q	P	1	$\&c$	$\perp$	$\perp$	T	$T_{\overline{Y}}(n_{12}, z) = \{R, T\}$
	n_{15} : $c := 2;$	Q	P	**2**	$\&c$	$\perp$	$\perp$	T	
$\star$	n_{13} : **use of** c	Q	P	**2**	$\&c$	$\perp$	$\perp$	T	$T_{\overline{Y}}(n_{13}, c) = 1 \sqcup 2 = \top_Z$
	return	Q	P	2	$\&c$	$\perp$	$\perp$	$\perp$	

Figure 6: A list of abstract interpretation of the program.

3.3.3 Discussion

Since abstract interpretation mimics program execution using abstract domains, the interprocedural constant propagation with conditional branches presented in [37] is accomplished in an elegant way. For example, the abstract interpreter will evaluate the condition of "if" expression first ; if the condition can be determined at compile time, only one branch will be taken. In addition, if the index to an array can be determined by constant propagation, only one location is treated (if the array name points to one array); this has been discussed previously. The presence of loops and procedure calls can also be solved by the idea of simulating program execution. Constant propagation using abstract interpretation shown in the section can find as many constants as the SCC algorithm for those examples shown in [37], but of course the solution by abstract interpretation applies to a much larger class of programming languages.

If there is recursion, for instance, recursive calls, some abstraction mechanism must be used to assure the finiteness of an abstract interpretation. For example, the Z1 system [19] employs abstract procedure strings to accommodate interprocedural analysis (see [19, 4]). The precision of an analysis depends critically on abstraction maps. In general, the more precise information we want, the more space and analysis time are required. However, the design of abstraction interpretation is not our main concern. The method presented later can be applied to any abstract interpretation of domains of finite height. In fact, our algorithm has been applied to Z1 in practice.

4 Entailment Model

In this section, we will define the *entailment relation*, propose an entailment model for computing the least fixpoints that arise in abstract interpretation, and present a basic algorithm based on the model.

4.1 Entailment Relation

The naive approach to computing least fixpoints in abstract interpretation constructs the graph of $fix\ \mathcal{F}$ in its entirety. This approach assumes every prestate is possible for every node, and the corresponding poststate is computed as in Examples 1 and 2. Usually, however, only a small fraction of the possible prestates will arise in a program. It is not always possible nor practical to explore the entire argument space of a semantic functional. In Example 2, suppose the result we want is the definedness of variables after procedure P is called *given that the initial input is* $\{a\}$. So we start with $\mathcal{F}I(n_0, \{a\})$. To compute it, we first compute $\mathcal{F}I(n_1, \{a\})$ and $\mathcal{F}I(n_2, \{a\})$. The only input argument to n_1, n_2, and n_0 is $\{a\}$. By contrast, there are 2^k possible prestates for each statement where k is the number of variables in the program.

Let $X = \Sigma \rightarrow \overline{X}$ and $Y = \overline{Y}$, so that $\mathcal{F}$ has type $(X \rightarrow Y) \rightarrow (X \rightarrow Y)$. Obviously, there is a relationship of dependence between the elements in X, induced by the functional $\mathcal{F}$. In the function f of Example 1, $f(a)$ depends on $f(b)$, $f(b)$ depends on $f(c)$, and $f(c)$ depends on itself. The dependence

relation can be used to order the evaluations efficiently. Suppose the function value we want to know is $f(a)$ (e.g., a is the start node). Only four, instead of twelve in Example 1, evaluations of $\mathcal{F}f$ are enough, in the order of $\mathcal{F}f(c)$, $\mathcal{F}f(c)$, $\mathcal{F}f(b)$, and $\mathcal{F}f(a)$. In Example 2, $I(n_0, \{a\})$ depends on $I(n_1, \{a\})$ and $I(n_2, \{a\})$. Only those nodes related, directly or indirectly, to the start node, $x_0 \in X$, need to be represented in the resulting fixpoint, if we are interested only in the states that can arise from a particular (abstract) starting condition. This is the concept of a minimal function graph [22]. We will define the relationship, which we call *entailment*, and the least fixpoint based on this relation.

Definition 1 (Entailment relation) *Let* $\mathcal{F} : (X \to Y) \to (X \to Y)$, $f : X \to Y$, *and* $x, x' \in X$. *We say that* x *entails* x' *under* $\mathcal{F}$ *and* f *(written* $x \overset{\mathcal{F},f}{\rightsquigarrow} x'$*) iff* $\exists y \sqsupset fx'$, $\mathcal{F}(f[y/x'])x \sqsupset \mathcal{F}fx$ □

The function $f[y/x]$ is the same as f, except that x is mapped to y. The intuition of Definition 1 is that if $\mathcal{F}(f[y/x'])x$ is higher than $\mathcal{F}fx$ for some y higher than fx', it can only be that $\mathcal{F}fx$ depends upon the value of fx'; that is, to raise the value of fx' will raise the value of $\mathcal{F}fx$. If x depends upon x' in this way, we say that x entails x'. On the other hand, $\forall x'$ *if* $x \not\rightsquigarrow x'$, $\mathcal{F}(f[y/x'])x = \mathcal{F}fx$ even if $y \sqsupset fx'$,

The entailment relation can be used to enormous practical advantage in computing least fixpoints. Let $x_0 \in X$ denote the initial node in which a computation begins. We write $x \overset{\mathcal{F}}{\rightsquigarrow} x'$ if $x \overset{\mathcal{F},f_i}{\rightsquigarrow} x'$ for some f_i that is an approximation of $fix\ \mathcal{F}$ during fixpoint computation, and we write $X^* = \{x' \in X | x_0 \overset{\mathcal{F}^*}{\rightsquigarrow} x'\}$. In other words, X^* is the set of nodes that are reached directly or indirectly from x_0; that is, $\forall x \in X^*$, $x \rightsquigarrow x' \Rightarrow x' \in X^*$ (for simplicity, we use $\rightsquigarrow$ to denote $\overset{\mathcal{F}}{\rightsquigarrow}$).

We will base the computation of least fixpoints on the entailment relation. $f|_Z$ denotes the function f restricted to the set $Z \subseteq X$. The initial approximation f_0 is the bottom function restricted to $\emptyset$ $(E1)$, and the entailed set X_0 consists of x_0 only $(E2)$. Application of $\mathcal{F}$ to the current approximation restricted to the elements in the entailed set (i.e., $f_i|_{X_i}$) will yield a new approximation $(E3)$ and entail some new nodes that are therefore added to the entailed set $(E4)$. The procedure is repeated until the approximation converges, that is, $f_{i+1} = f_i$ and $X_{i+1} = X_i$. Thus, the least fixpoint of $\mathcal{F}$ restricted to X^* (the nodes entailed, directly or indirectly, by x_0), denoted by $(fix\ \mathcal{F}|_{X^*})$, is reached. Now consider the sequence:

$(E1)$ $f_0 = \lambda x.\perp|_\emptyset$

$(E2)$ $X_0 = \{x_0\}$

$(E3)$ $f_{i+1} = \{x \mapsto \mathcal{F}(f_i|_{X_i})x \mid x \in X_i\}$

$(E4)$ $X_{i+1} = X_i \bigcup \{x' | x \overset{\mathcal{F}, f_i|_{X_i}}{\rightsquigarrow} x' \ \forall x \in X_i\}$

Let n be the least integer satisfying $f_{n+1} = f_n$. It can be shown that $f_n = (fix\ \mathcal{F}|_{X^*})$, and that $X_n = X^*$.

First we need to prove that $X_n = X^*$.

Lemma 2 *Let $G = (V, E)$ is the entailment graph where $V = X^*$ and $\langle x, x' \rangle \in E$ if $x \rightsquigarrow x'$. If the length of the shortest path from x_0 to x is i, then x is in X_i.*

Proof: Let $len(x)$ be the length of the shortest path from x_0 to x. It is easy to prove by induction on i.

1. Basic step: $len(x_0) = 0$ and $x_0 \in X_0$.

2. Induction step:

   Suppose that $\forall x \ \ if \ \ len(x) = i \ \ then \ \ x \in X_i$ holds true.

   If $len(x) = i + 1$, there exists z such that $z \rightsquigarrow x$ and $len(z) = i$. We know $z \in X_i$.

   Application of $(E4)$ $X_{i+1} = X_i \bigcup \{x' | x \overset{\mathcal{F}, f_i|_{X_i}}{\rightsquigarrow} x', \forall x \in X_i\}$ will add x into X_{i+1}.

Therefore we can conclude that $if\ len(x) = i \ \ then \ \ x \in X_i$. □

Theorem 1 $X_n = X^*$.

Proof: From Lemma 2, all the elements entailed by x_0, directly or indirectly, are included in X_l where

$$l = max\{len(x) | x \in V(G)\}, l \leq n$$

Hence, $X_l = X_{l+1} \cdots = X_n = X^*$. □

Theorem 2 $f_E = (fix\ \mathcal{F}|_{X^*})$ *where f_E is the fixpoint computed by the procedure of $E1$ to $E4$.*

Proof: To prove that $f_E = (fix\ \mathcal{F}|_{X^*})$, we need to prove that $f_E \sqsupseteq (fix\ \mathcal{F}|_{X^*})$ and $f_E \sqsubseteq (fix\ \mathcal{F}|_{X^*})$.

1. First, we prove $f_E \sqsupseteq (fix\ \mathcal{F}|_{X^*})$.

   $fix\mathcal{F}|_{X^*}$ is the least fixpoint of $\mathcal{F}|_{X^*}$; that is, if $\mathcal{F}|_{X^*} f = f$, then $f \sqsupseteq fix\mathcal{F}|_{X^*}$. When the procedure terminates, f_E is a fixpoint of $\mathcal{F}|_{X_n}$; in other words, $(\mathcal{F}|_{X_n}) f_E = f_E$. We know $X_n = X^*$. Thus, f_E is a fixpoint of $\mathcal{F}|_{X^*}$. Therefore, $f_E \sqsupseteq fix\ \mathcal{F}|_{X^*}$.

2. Second, we prove $f_E \sqsubseteq (fix\ \mathcal{F}|_{X^*})$.

   That is to prove that $(f_i : X_i \rightarrow Y) \sqsubseteq fix\mathcal{F}|_{X_i}$ holds true for all i. It is proved by induction on i.

   - Basic step:

     $X_0 = \{x_0\}$ and $f_0 = \lambda x.\perp|_\emptyset$

     Hence, $(f_0 : X_0 \rightarrow Y) \sqsubseteq fix\ \mathcal{F}|_{X_0}$ holds true.

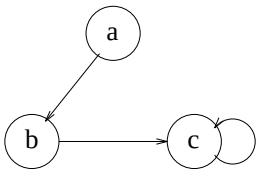

Figure 7: The entailment graph of Example 1.

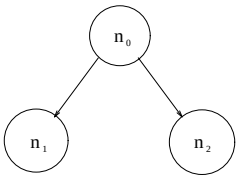

Figure 8: The entailment relation of Example 2.

- Induction step:

  Suppose $(f_i : X_i \rightarrow Y) \sqsubseteq fix\ \mathcal{F}|_{X_i}$ holds true for some i.

  To compute a node $x \in X_i$ to obtain $f_{i+1}(x)$, consider x' such that $x \overset{\mathcal{F},f_i}{\rightsquigarrow} x'$:

  $$if\ x' \in X_i,\quad f_i(x') \sqsubseteq fix\ \mathcal{F}|_{X_i}(x'), \text{ or}$$

  $$if\ x' \notin X_i,\quad f_i(x') = \bot$$

  Hence, $f_{i+1}(x) = \mathcal{F}f_i(x) \sqsubseteq fix\ \mathcal{F}|_{X_{i+1}}(x)$ since for all x' such that $x \overset{\mathcal{F},f_i}{\rightsquigarrow} x'$, $f_i(x') \sqsubseteq fix\ \mathcal{F}|_{X_i}(x')$.

Therefore, $(f_i : X_i \rightarrow Y) \sqsubseteq fix\mathcal{F}|_{X_i}$ holds true for all i. That is, $f_E \sqsubseteq fix\ \mathcal{F}|_{X^*}$.

From 1 and 2, we can conclude that $f_E = (fix\ \mathcal{F}|_{X^*})$. □

Thus, f_E is the least fixpoint if $\mathcal{F}$ is restricted to those nodes that are reachable from the initial node. Recall that X (i.e., $\Sigma \rightarrow \overline{X}$) represents the set of expressions and possible (pre-)states together, and let $x_0 \in X$ denote the start expression with the initial state in which a computation begins. In other words, X^* is the set of expressions and their resulting prestates that are reached directly or indirectly from x_0. Of course, we can envisage the entailment relation over prestates as a graph.

An entailment graph describes the relationship between the elements of X induced by $\mathcal{F}$; that is, $x \overset{\mathcal{F}}{\rightsquigarrow} x'$ means the computation of $(fix\ \mathcal{F})x$ depends on the computation of $(fix\ \mathcal{F})x'$, and in this case there is an edge $\langle x, x' \rangle$ in the entailment graph. The entailment graph will be constructed during the computation of the least fixpoint. The entailment relation for Examples 1 and 2 are given in Figures 7 and 8. It is no accident that the relationship of the elements in X resembles a flow graph. However, the entailment graph and control flow graph are quite different, being at most distantly related. Consider the expression $x + y + z$: there are three entailment edges, from the whole expression to the subexpressions x, y, and z, depending on the definition of the semantic functional $\mathcal{F}$. The idea of entailment relation is defined to match the dependences among evaluation steps during computation. It is general and applicable to more languages, for example, the functional language of which the control flow graphs are not evident.

4.2 Basic Entailment Algorithm

In this section, we will give an algorithm for computing the least fixpoint using the entailment relation. However, it is not efficient to compute all the elements in X_i to obtain f_{i+1}. Only those nodes that are newly entailed or by which the nodes entailed are improved. Therefore, W_i is used to contain those nodes that are actually needed to be computed. The following equations are used to efficiently compute the least fixpoint restricted to those nodes entailed by the start node.

$(A1)$ $f_0 = \lambda x.\bot|_\emptyset$

$(A2)$ $X_0 = \{x_0\},\ W_0 = \{x_0\}$

$(A3)$ $f_{i+1} = \{x \mapsto \mathcal{F}(f_i|_{X_i})x \mid x \in W_i\}$

$(A4)$ $W_{i+1} = \{x' | x \overset{\mathcal{F},f_i|_{X_i}}{\rightsquigarrow} x',\ x \in W_i\} \bigcup \{x | x \rightsquigarrow x' \ and\ \mathcal{F}(f_i|_{X_i})x' \sqsupset (f_i|_{X_i})x',\ x' \in W_i\}$

$(A5)$ $X_{i+1} = X_i \cup W_{i+1}$

Let n be the least integer satisfying $f_{n+1} = f_n$. It is easy to show that $f_n = (fix\ \mathcal{F}|_{X^*})$, and that $X_n = X^*$. The proof is similar to that of Theorem 2.

Given a node $x \in V(G)$, we need to evaluate the functional, $\mathcal{F} : X \rightarrow Y$, $|X| - 1$ times, to obtain the set of nodes that x entails, This sums up to $O(|X|^2)$ evaluations to compute X_{i+1}. It is not practical to compute the entailment relation exactly. Therefore, we employ an approximate method to find the entailment relation in our algorithm; if the value of $f(b)$ is retrieved to compute $\mathcal{F}f(a)$, an edge $\langle a, b\rangle$ is added to G. However, the entailment relation derived in this way is not exactly the same as we define (see Section 4.3).

Algorithm 1 in Figure 9 is a basic algorithm for computing the least fixpoint of $\mathcal{F}$ given that start node w_0 and the initial state ρ_0, using $A1 - A5$ and the approximate entailment relation. We will call the graph that Algorithm 1 produces the computed entailment graph $\overline{G}$ hereafter. $\overline{G}$ uses an approximation of the entailment graph from Definition 1, in which all prestates for a fixed $\sigma \in \Sigma$ are collapsed to a single node σ in $V(\overline{G})$. The algorithm keeps a prestate and a poststate for each expression in the table $T_{\overline{X}}$ and $T_{\overline{Y}}$ respectively. The two tables are initialized with bottom values. Prestates flow forward and poststates flow backward in $\overline{G}$.

Initially, the worklist W contains the start expression w_0, only w_0 is in $\overline{G}$, and $T_{\overline{X}}(w_0)$ is assigned ρ_0. We repeatedly choose an element w from W to evaluate with its current prestate $T_{\overline{X}}(w)$ until W is empty. If the poststate of w is improved, those nodes that entail (depend on) w will be put on W (i.e., they need to be re-evaluated), and the new poststate is recorded in $T_{\overline{Y}}$. If w entails w', $f(w', \rho')$ is invoked by $\mathcal{F}$ in the evaluation of $\mathcal{F}(f, w, T_{\overline{X}}(w))$, where ρ' is a prestate, prepared by w, for w'. In f, a node w' and an edge $\langle w, w'\rangle$ are added to $\overline{G}$, the new prestate ρ' to w' is recorded in $T_{\overline{X}}$, and the current value of $T_{\overline{Y}}(w')$ is returned. To start the computation of the least fixpoint, we call $Fixpoint(\mathcal{F}, w_0, \rho_0)$.

Algorithm 1 *The basic iterative algorithm using the entailment graph:*

function $Fixpoint\ (\mathcal{F} : (\Sigma \to \overline{X} \to \overline{Y}) \to (\Sigma \to \overline{X} \to \overline{Y}),\ w_0 : \Sigma,\ \rho_0 : \overline{X}) : \Sigma \to \overline{X} \to \overline{Y}$

```
  G : (V, E) = (Σ, Σ × Σ)                    /*the computed entailment graph*/
  W : 2^Σ                                    /*worklist*/
  T_X̄ : Σ → X̄                                /*prestate of a node*/
  T_Ȳ : Σ → Ȳ                                /*poststate of a node*/
  w : Σ                                      /*the current node to evaluate*/
  y : Ȳ                                      /*the computed value*/

  function f(w' : Σ, ρ : X̄) : Ȳ
    begin
      if (ρ ⋢ T_X̄(w')) then                  /*if the prestate is higher*/
        W ← W ∪ {w'}                         /*put the node that w entails on W*/
        T_X̄(w') ← T_X̄(w') ⊔ ρ                /*record the new prestate*/
      endif
      V(Ḡ) ← V(Ḡ) ∪ {w'}                     /*build the graph Ḡ*/
      E(Ḡ) ← E(Ḡ) ∪ {⟨w, w'⟩}
      return T_Ȳ(w')                         /*return the current poststate of w'*/
    end

  begin /*Fixpoint*/
    T_X̄(e) ← ⊥_X̄, T_Ȳ(e) ← ⊥_Ȳ   ∀e ∈ Σ      /*initialization*/
    Ḡ ← ⟨{w_0}, ∅⟩                           /*only w_0 is in Ḡ*/
    W ← {w_0}                                /*W contains w_0*/
    T_X̄(w_0) ← ρ_0                           /*initial prestate for the start expression*/
    while (W ≠ ∅) do                         /*iteratively until W is empty*/
      w ← retrieve an element from W         /*retrieve a node to evaluate*/
      y ← F(f, w, T_X̄(w))                    /**EVALUATION**/
      if (y ⋢ T_Ȳ(w)) then                   /*if its poststate is improved*/
        W ← W ∪ {w' | ⟨w', w⟩ ∈ E(Ḡ)}        /*put all the nodes that entail w on W for reevaluation*/
        T_Ȳ(w) ← y                           /*record the new poststate*/
      endif
    endwhile
    return (T_X̄, T_Ȳ)
  end /*Fixpoint*/
```

Figure 9: Basic algorithm based on entailment relation

4.3 Discussion

Note that the computed entailment graph $\overline{G}$ may have more edges than the (true) entailment relation derived from Definition 1. consider the example:

$$
\begin{array}{l}
f = \lambda x.\ \text{case } x \text{ of} \\
\qquad a : \top \cup f(b) \\
\qquad \cdots \\
\qquad \text{endcase}
\end{array}
$$

The edge $\langle a, b\rangle$ is included in $\overline{G}$, but a does not entail b because no matter what the value of $f(b)$ is, the value of $\mathcal{F}f(a)$ is $\top$. In addition, more edges may be included, like $\langle n_0, n_1\rangle$ caused by "$n_0 : \cdots f(n_1) - f(n_1)$" in the semantic functional. However, it is easy to show that superfluous edges in $\overline{G}$ do not affect the correctness of the fixpoint. Let f_A be the fixpoint computed by Algorithm 1 and X^+ be the set of entailed nodes using the approximate entailment. We know $X^* \subseteq X^+$.

$$\forall n \in X^*, \quad f_A(n) = f_E(n).$$
$$\forall n \in (X^+ - X^*), \quad f_A(n) \sqsupseteq f_E(n) = \bot$$

Consider a node $n \in (X^+ - X^*)$. If $f_A(n) = \bot$, it is the same as $f_E(n)$. If $f_A(n) \sqsupset \bot$, it will leads to a safe approximation of $f_E(n)$. Hence, f_A is a safe approximation of f_E; however, f_A restricted to X^* is the same as $(fix\ \mathcal{F}|_{X^*})$.

On the other hand, notice that by the entailment definition, $\langle n_2, \rho'\rangle$ entails $\langle n_1, \emptyset\rangle$ (where $\rho' = f(n_1, \rho)$), induced by $f(n_2, f(n_1, \rho))$ of the functional $\mathcal{F}$. $\overline{G}$, however, does not contain an edge $\langle n_2, n_1\rangle$. This is caused by the projection function Θ and the approximate method we use to compute the entailment relation. In fact, it is not necessary for $\overline{G}$ to contain such edges, because the dependence of n_2 upon n_1, is implicitly enforced in the functional $\mathcal{F}$. Besides, such dependences are "internal" to a single evaluation step of Algorithm 1 (see the line marked /**EVALUATION**/), whereas the purpose of $\overline{G}$ is to record the dependences between separate evaluations. It is easy to include those internal dependences in $\overline{G}$ if desired.

5 Efficient Computation of Least Fixpoints

This section presents the strategies for determining an effective order of evaluations and then the guided-entailment model using these strategies is described and discussed.

5.1 Strategies for Improving the Basic Algorithm

As discussed previously, propagation using def-use chains or sparse representations is not straightforward for interprocedural analysis in the presence of unrestricted pointers. Furthermore, the control flow graph is not always available a priori; for example, those of programs with higher-order functions. Therefore, it is difficult to avoid propagating throughout the entire program for the complex analyses in which we

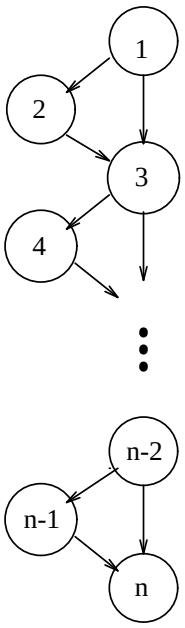

Figure 10: An example for describing the worst case

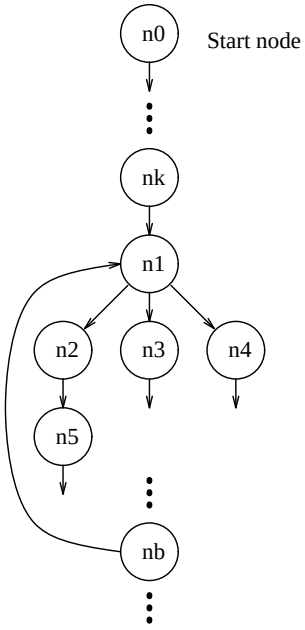

Figure 11: An example for explaining the strategies

are interested. However, we can exploit the entailment relation to dynamically order the evaluations that lead to the fixpoint more quickly.

In Algorithm 1, we did not specify how to retrieve the next element to evaluate. The order in which elements are chosen will significantly affect the performance of the iterative method. Consider the entailment graph in Figure 10. The basic algorithm may require $O(2^n)$ evaluations in the worst case even though $O(n)$ are apparently enough. The best sequence of evaluations is 1,2,3,...,n-1,n (for prestate forwarding), n-1,..,2,1 (for poststate propagation). However, the basic algorithm may evaluate 1 and put 2 and 3 on the worklist, then choose 2 and propagate the poststate of 2 immediately back to 1. Later it may choose 3 and propagate back to 1 first and then back to 2 and 1. That is, whenever the poststate of a node is improved, it may propagate in the worst possible way: propagating to node 1 before propagating to the other predecessors of n. the worst sequence is 1,[2,1], [3,1,2,1], [4,3,1,2,1], [5,3,1,2,1,4,3,1,2,1], ..., which sums to O(2^n).

In fact, it is possible to determine an effective order without much overhead. In the following, the strategies for determining the evaluation order that underlie our model will be described separately. Each of these strategies, independently, improves a basic iterative algorithm. The strategies, *waiting for all successors, leading node first, suspended evaluation, and back edge first*, need not be used independently; one strategy may benefit another. The *guided-entailment method* we propose integrates all the strategies in a unified way. We will use the example entailment graph in Figure 11 to illustrate these strategies.

5.1.1 Waiting for All Successors

When n_1 is evaluated (see Figure 11), its successors n_2, n_3, and n_4, are put on the worklist. If n_2 (or any of its successors) is evaluated and improved, n_1 is put back on the worklist for reevaluation. It may happen that n_1 is chosen earlier than n_3 and n_4. The sequence of evaluations will be $n_2, ..., n_1, ..., n_3, ..., n_1, ..., n_4, ..., n_1$. Thus n_1 is evaluated repeatedly, as each of its successors is improved. Instead, n_1 should wait until all its successors are evaluated. It is then evaluated only once, as in the sequence $n_2, ..., n_3, ..., n_4, ..., n_1$.

5.1.2 Leading Node First

In Algorithm 1, there are two kinds of nodes on the worklist. One is a *leading node*—added for computing a new poststate because its prestate has been improved. The other is a *retreating node*—added for reevaluation because the poststate of one of its successors has improved. In the example of Figure 11, n_1 depends on n_2, n_3, and n_4. When n_1 is evaluated, n_2, n_3, and n_4 will be put on the worklist. n_2, n_3, and n_4 are leading nodes at this time. Suppose n_2 is next evaluated. If its poststate is improved, n_1 will be put on the worklist for propagating the new poststate of n_2. At this moment, n_1 is a retreating node. Notice that n_5, entailed by n_2, is already on the worklist at this point.

It is obvious that a leading node should be preferred over a retreating node. Suppose we choose n_1 (a retreating node) first, and propagate back to the start node n_0. Later, n_5 (a leading node) will be evaluated. If its poststate is improved, n_2 will be reevaluated. If the poststate of n_2 is also improved, all the nodes on the path from n_1 back to the start node n_0 will be evaluated again. The resulting sequence will be $[n_1, n_k, ..., n_0], ..., n_5, ..., n_2, ..., [n_1, n_k, ..., n_0]$. The sequence in "[ ]" is redundant. Hence, choosing leading nodes first will lead to a shorter (or equal) sequence of evaluations reaching the least fixpoint .

5.1.3 Suspended Evaluation

In the basic algorithm, n_1 is evaluated, and its successors are put on the worklist as leading nodes. If the poststate of any of its successors is improved, n_1 will be reevaluated. It is more efficient to perform only as much of the evaluation of n_1 as is needed to generate prestates for its successors, put them on W, and then suspend the evaluation of n_1. After its successors are evaluated, the suspended evaluation of n_1 is resumed. If there exists an internal entailment between the successors of a node, the evaluation of this node can be partitioned into slices, in which each prepares a prestate for a successor, puts this successor on W, and then is suspended in the order given by the entailment relation.

If the entailment graph is acyclic, it is easy to suspend the evaluation of a node and evaluate its successors first; when there is a cycle, this will cause an infinite wait. The entailment graphs in program analysis are usually cyclic. Therefore, a mechanism must be designed to cope with circularity. We will return to this later.

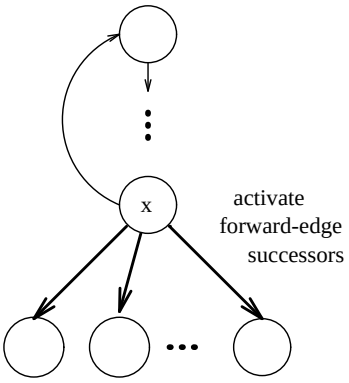

Figure 12: Activation of a node

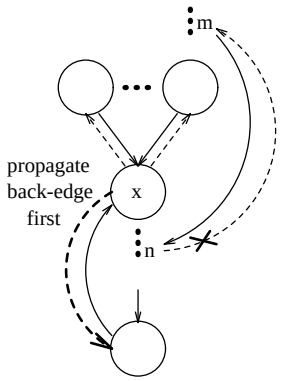

Figure 13: Propagation of a node (dotted line)

5.1.4 Back Edge First

If a node entails its ancestor, there is a circular entailment relation. In other words, a back edge exists in the entailment graph. When the poststate of a node is improved, the poststate will be propagated to its predecessors, which will be put on W for reevaluation. Consider n_1 in the example. One of its predecessors, n_b, is also its descendant. Suppose we propagate the poststate of n_1 to n_k, and afterwards, due to circularity, a change in one of its descendants n_b results in a new poststate for n_1, which will invalidate the earlier propagation to n_k. The sequence will be n_1, $[n_k, ..., n_0]$,$(n_b, ..., n_1)$,$[n_k, ..., n_0]$,$(n_b, ..., n_1)$,$[n_k, ..., n_1]$,n_b, ... where "(...)" denotes a loop. Whenever the poststate of n_1 is improved by the propagation from n_b, the previous evaluations of n_k and all nodes along the way back to the start node n_0 denoted by "[...]" will be invalid.

If a back edge exists, as $\langle n_b, n_1 \rangle$ in this example, the value of the header of a loop, such as n_1, may be improved later as a result of propagation along the back edge to the header itself. Thus, propagation to those predecessors not connected by back edges should be suspended. We can accomplish this by always propagating a new poststate along a back edge first, if any, until the loop is stable. The resulting sequence of this example will be $(n_1, n_b, ...), (n_1, n_b, ...), (...), ..., [n_k, ..., n_0]$. The idea is similar to the loop first strategy which is commonly used in dataflow analysis [38].

5.2 Guided-Entailment Algorithm

In this section, we will propose a model for efficiently computing the least fixpoint of a semantic functional $\mathcal{F} : (X \to Y) \to (X \to Y)$ in which the relationship of elements of X is not given a priori. The aforementioned strategies are embedded in the model. Based on the strategies, local knowledge of the entailment graph that is dynamically constructed is exploited to determine dynamically an effective order of evaluations.

There are two primary functions in this model: "$Activate(x)$" pursues a new value in Y for a node

$x \in X$, depicted in Figure 12, and "$Propagate(x)$" passes back an improved value of a node $x \in X$ to its predecessors, depicted in Figure 13. At the end of "$Activate(x)$," "$Propagate(x)$" will be called after the value of x is actually evaluated. Basically, "*Activate*" occurs along the edges of the entailment graph, while "*Propagate*" takes place in the reverse direction of the entailment relation. Each node (expression, statement, or block) in the program acts as an object. When it receives a message (activate or propagate), it will perform a corresponding action.

5.2.1 Activation

A node is activated, as a leading node, when it is actually needed (entailed) during the computation of the least fixpoint. A node will not be evaluated until all the values of its successors are available (stable so far; this is determined by local knowledge of the entailment relation). To avoid an infinite wait, when a node entails its ancestor, it will neither activate nor wait for this ancestor. To start the computation of the least fixpoint, the initial node $x_0 \in X$ is activated.

```
Activate(x : X)::
begin
    if (x entails a node x' and ⟨x, x'⟩ ∉ BackEdge(G)) then
        for each x' of those entailed non-ancestor nodes
            Activate(x') and
            wait for a notice of completion from x'
        endfor
    endif
    if any value of the successors is improved then
        y ← compute (Ff)x using the computed values of x's successors
        if (y is improved) then record the new value y
    endif
    Propagate(x)   /*will be defined later*/
end
```

In "*Activate*," the first three strategies are integrated.

- Waiting for all successors: when x is activated, its non-ancestor successors are activated. x will wait until all the successors complete their evaluations, and the value of x is computed once with all their new values. "Waiting for all successors" is therefore accomplished.

- Leading node first: a node will first activate its non-ancestor successors; that is, the model recursively calls "*Activate*" first for activating leading nodes, and "*Propagate*" is called later for propagating a computed value to its predecessors (as retreating nodes). "Leading node first" is obviously achieved.

- Suspended evaluation: the evaluation of x will not continue until all the activated successors return; then the value of x is actually computed. Thus "suspended evaluation" is integrated. It will be

clear how the evaluation of a node is suspended (if the node depends on other nodes) when we give a concrete algorithm for the functions g_i of $\mathcal{F}$, in Section 6.2.

We did not specify the order for activating the successors. If there is a dependence relation (internal entailment) between these successors in the semantic functional $\mathcal{F}$, the order for activating them should be the same as the entailment relation; otherwise, the order for activating the successors is not important.

5.2.2 Propagation

When the value of a node x is improved, it is necessary to propagate the new value to x's predecessors as retreating nodes. The "back edge first" strategy is applied in "$Propagate$". If a back edge $\langle m, x\rangle$ exists, propagation of the value of x to its non-back-edge predecessors should be suspended since the value of x may not be stable yet. The loop, caused by $\langle m, x\rangle$, is manipulated first until it is stable. Then the stable value of x is propagated back to its non-back-edge predecessors. Therefore " $Propagate$" is defined as:

```
Propagate(x ∈ X)::
begin
   for each m ∈ {x'|⟨x', x⟩ ∈ BackEdge(G)} do BackFirst(x, m)
   for each z ∈ {x'|⟨x', x⟩ ∈ E(G) − BackEdge(G)} do
       send a notice of completion to z and return
end
```

When a back edge exists, we call "$BackFirst$" to propagate along the back edge to the header repeatedly in the loop. $BackFirst(header, z)$ maintains a working set of nodes to be evaluated for propagation in a loop, caused by the back edge $\langle z, header\rangle$. Initially it contains only the back-edge predecessor z. We repeatedly retrieve a node to evaluate from the working set until the set is empty (the order is not important). If the value of a node other than the $header$ is improved, all its predecessors within the loop (i.e., they are not the ancestors of $header$) are put into the working set. If $header$ is improved, only the original back-edge node z that defines the loop is put in the set. If a back edge is found while propagating in a loop, we deal with the inner loop first, recursively. The detailed algorithm is included in Section 6.2.

During the loop propagation, if an edge $\langle m, n\rangle$ exists where n is within the loop and m is an ancestor of the header (see Figure 13), m will not be propagated because m, crossing the header, is not within the loop. Note that m is not completely evaluated yet, although m was already notified by n when n was first evaluated and propagated (at that time, it is unknown that n is in a loop); m is awaiting a completion notice passed from the loop header indirectly. Therefore, the ordering induced in our model is as efficient as those methods that examine the whole control flow graphs a priori, even though in our method only local knowledge of the graph is kept (back edges and non-back edges at each node), and the identification of loops occurs as the algorithm proceeds.

5.3 Summary of the Guided Entailment Model

If the entailment graph is acyclic, the order of evaluations derived in the guided-entailment model is depth-first. If there are back edges, the nodes in a connected component are evaluated repeatedly until the component is stable, and the components are visited in reverse topological order. The model takes strongly connected components into consideration and takes advantage of local knowledge of the entailment relation to exclude unnecessary evaluations. The model is conceptually simple but flexible.

In summary, for an acyclic entailment graph, each node will be activated for pursuing a new value (as a leading node) exactly once, and will be propagated and then evaluated at most once (as a retreating node); those nodes that do not depend on any other nodes are evaluated when they are activated, and there will not be any propagation from other nodes to them. Therefore, for an acyclic entailment graph of n nodes, the total number of evaluations is no more than $2n$. For an entailment graph with only self loops (i.e., single-node loops), the time complexity is $O(n + l)$ where l is the height of lattices for abstract domains. As to a general entailment graph, we have collected experimental data (Section 7.2) that suggests that the model is efficient in practice. Intuitively, the behavior and complexity of the guided-entailment algorithm should be no worse than those of interval analysis.

If the algorithm is applied to a conventional dataflow analysis problem, that is, the control flow graph is given a priori, its behavior is similar to interval anlysis [14]. Given a directed graph, the ordering derived by our algorithm is basically the same as that derived by the iterative algorithm in [10] except that the guided entailment model first traverses and activates the nodes, but the order of evaluations should be the same.

6 Guided-Entailment Algorithm

So far, we have described a model for computing least fixpoints efficiently without control flow graphs a priori, but we must turn it into a concrete algorithm. Section 6.1 describes the data structures. Section 6.2 presents the guided-entailment algorithm. Section 6.3 describes another data structure for a sequential implementation to further improve the efficiency.

6.1 Design and Implementation

This section describes two essential data structures, dependence counts and path strings, in the implementation of the guided entailment algorithm.

6.1.1 Dependence Count (DC)

To avoid infinite wait, a count of the number of successors for which a node is waiting is maintained. Whenever a node activates its successors (puts its successors on the leading worklist), its dependence

count will be incremented by one. If a back-edge successor exists, to avoid an infinite wait, this successor will not be activated; that is, the dependence count is not incremented by a back-edge successor. After a node is evaluated, it will notify all its dependents. This can be accomplished simply by decrementing the dependence counts of its dependents by one. Once the count of a node becomes zero, all successors for which it is waiting have reported back, and that node is ready to be evaluated (put on the retreating worklist).

6.1.2 Path String (PS)

If a circular entailment relation exists, this will cause an infinite wait since a node will not be evaluated until all its successors are evaluated. To avoid this, we keep track of back edges as the graph develops to identify loops. A path string is a mechanism designed for recording paths dynamically. For example, n_0 is the initial node whose path string is "0." If there is a entailment edge from n_0 to n_1, the path string of n_1 is "01," the path string of its predecessor concatenated with its identification.

Each node is associated with a set of path strings. It is easy to check if there is a back edge using path strings. If x entails z and the path strings of x contains "z", $\langle x, z \rangle$ is a back edge. Moreover, path strings can easily be implemented so that they share storage with one another, and consume no more space than the entailment graph itself. We will show below that for a sequential implementation, a single bit-vector of length n (the number of nodes in a program) is sufficient.

6.2 Guided-Entailment Algorithm

The guided-entailment algorithm is described in this section. In $BackFirst$, a local working set is needed for propagation in a loop. To simplify the implementation and yet use the whole structure of the basic algorithm, $BackFirst$ in Algorithm 3 makes use of the global W as the local working set; it saves the worklist W when it is called, and restores W before it returns.

If a node is found to depend on the poststates of other nodes that are yet to be evaluated, the computation of the semantic function g_i for the node need not be finished. In order to suspend the computation, we need to check the dependence count in the semantic function; if it is not zero, the computation, defined by each semantic function, is suspended (see ** in g_i). When the node is chosen to be evaluated again, it will resume at the point where it was suspended. Each function g_i in $\mathcal{F}$ is defined as,

Algorithm 2 *An iterative algorithm based on the guided-entailment model:*

```
function Fixpoint (F : (Σ → X̄ → Ȳ) → (Σ → X̄ → Ȳ), w_0 : Σ, ρ_0 : X̄) : Σ → X̄ → Ȳ
  G : (V, E) = (Σ, Σ × Σ)                          /*the entailment graph*/
  W : 2^Σ                                          /*global worklist*/
  T_X̄ : Σ → X̄                                      /*prestate of a node*/
  T_Ȳ : Σ → Ȳ                                      /*poststate of a node*/
  PS: array of 2^{V+}                              /*path strings for each node*/
  DC: array of integer                             /*dependence count for each node*/
  w: Σ                                             /*the current node to process*/
  y: Ȳ
  function f(w' : Σ, ρ : X̄) : Ȳ
    begin
      if ⟨w, w'⟩ ∉ E(G)                            /*if the edge is not constructed yet*/
        PS[w'] ← PS[w'] ∪ {s & w' | s ∈ PS(w)}      /* "&" means concatenation of strings*/
      V(G) ← V(G) ∪ {w'}                           /*build the entailment graph*/
      E(G) ← E(G) ∪ {⟨w, w'⟩}
      if (ρ ⋢ T_X̄(w')) then                        /*if the prestate is higher*/
        T_X̄(w') ← T_X̄(w') ⊔ ρ                      /*record the prestate*/
        if (⟨w, w'⟩ is not a back edge) then       /*if not a back edge*/
          W ← W ∪ {w'}                             /*"Activate" the successor, and*/
          DC[w] ← DC[w] + 1                        /*increment the dependence count of w*/
        endif
      endif
      return T_Ȳ(w')                              /*return the current poststate*/
    end
begin  /*Fixpoint*/
  for all e ∈ Σ do
    DC[e] ← 0, PS[e] ← ∅, T_X̄(e) ← ⊥_X̄, T_Ȳ(e) ← ⊥_Ȳ   /*initialization*/
  G ← ⟨{w_0}, {}⟩                                  /*only node w_0 is in G*/
  W ← {w_0}                                        /*the worklist contains w_0*/
  T_X̄(w_0) ← ρ_0                                   /*prestate for w_0*/
  PS[w_0] ← {0}                                    /*path string for w_0*/
  while (W ≠ ∅) do                                 /* iteratively until W is empty*/
    w ← retrieve an element from W                 /*retrieve an element to process*/
    y ← F(f, w, T_X̄(w))                            /**EVALUATION**/
    if (DC[w] = 0) and (y ⋢ T_Ȳ(w)) then           /*if w awaits no node and improved*/
      T_Ȳ(w) ← y                                   /*record the new poststate*/
      for each z ∈ {w'|⟨w', w⟩ ∈ BackEdge(G)} do
        BackFirst(w, z)                            /*"Propagate": to back-edge dependents first*/
      for each z ∈ {w'|⟨w', w⟩ ∈ E(G) − BackEdge(G)} do /*then to non-back-edge predecessors:*/
          DC[z] ← DC[z] − 1;                       /*decrement the dependence count of its dependents*/
          if (DC[z] = 0) then W ← W ∪ {z}          /*trigger a ready node as a retreating node*/
      endfor
    endif
  endwhile
  return (T_X̄, T_Ȳ)
end  /*Fixpoint*/
```

Figure 14: Guided-entailment Algorithm

Algorithm 3 *An iterative algorithm for back edge first propagation:*

```
function BackFirst(header, z ∈ Σ): void
  local NewW : 2^Σ
begin
  NewW ← W                                  /*save W, and use W as a local working set*/
  W ← {z}                                   /*initialized with the back-edge predecessor z*/
  while (W ≠ ∅ )                            /*repeat until it is empty*/
    w ← retrieve an element from W          /*choose an element to evaluate; use global w*/
    y ← F(f, w, T_X̄(w))                     /*evaluation of the current node*/
    if (y ⋢ T_Ȳ[w]) then                    /*if the poststate is improved*/
      T_Ȳ[w] ← y                            /*record the improved poststate*/
      if (w ≠ header) then                  /*if not the header, propagate*/
        for each v ∈ {x'|⟨x', w⟩ ∈ BackEdge(G) and x' is not an ancestor of header} do
          BackFirst(w, v)
        for each v ∈ {x'|⟨x', w⟩ ∈ E(G) − BackEdge(G) and x' is not an ancestor of header} do
          W ← W ∪ {v}                       /*put the non-back predecessors on W*/
      else W ← W ∪ {z}                      /* if w = header, only z is put on W*/
      endif
    endif
  endwhile
  W ← NewW                                  /*restore W*/
end
```

Figure 15: BackFirst algorithm

```
g_i(f, σ, ρ)
    begin
        for each j ∈ Successors(σ)
            if (j entails k) and (k has been activated) then suspended*
            ··· (prepare the prestate ρ_j)
            call f(j, ρ_j) to activate node j with prestate ρ_j
            ··· (post-processing)
        endfor
        if (DC[σ] > 0) then suspended**
        computation of Ff for σ using the current poststates of Successors(σ)
    end
```

In the "for" loop in g_i above, if a successor j depends on another successor k, the order for activating its successors (in the entailment graph) must preserve the dependence relation (internal entailment); note that the suspension marked with * in g_i. Otherwise, all the successors can be processed simultaneously or in any order.

6.3 Path Bit Vector

To detect the back edges in the graph G, Algorithm 2 records all the paths that lead to a node. However, this is very inefficient. As a matter of fact, for a sequential implementation of the guided entailment algorithm, the successors of a node must be handled one by one sequentially. We can activate only one

successor at a time; after its value is computed, the next successor is activated. Thus, the order derived for activating leading nodes is basically depth-first; that is, only one path leads to a node at a time during analysis. It is not necessary to keep all path strings during analysis, in a sequential implementation. We can therefore make use of this characteristic to simplify path strings.

A bit vector $Path$ is designated to denote the current path as the analysis proceeds. Initially no nodes are on the path. When a node x puts a successor on the worklist as a leading node, $Path[x]$ is set 1; that is, x is on the path. When a node z is put on the retreating worklist, z is removed from the path. In this way, it is easy to identify back edges. If x entails z and $Path[z] = 1$ (i.e., z is already on the path), $\langle x, z \rangle$ is a back edge. Our guided-entailment algorithm has been implemented using such a path bit vector in a sequential version.

7 Experiments and Discussion

We have conducted experiments to compare the performance of the guided-entailment method with that of other iterative algorithms that do not require the control flow graph a priori. Section 7.1 describes the program analyses conducted and the test programs. Section 7.2 shows the experimental results and discussion.

7.1 Program Analyses and Test Programs

The experiments are based on the abstract interpretation used in MIPRAC [18]. MIPRAC analyzes and compiles an intermediate language, called MIL, which includes higher-order procedures, dynamic allocation, and unrestricted pointer manipulation. Programs in many source languages, such as FORTRAN, C, and Scheme, can be transformed into MIL programs. The compiler performs whole-program analysis of the intermediate form by abstract interpretation.

We use Z1, an analyzer generator based on MIPRAC, to generate various interprocedural analyses. The analyses we have experimented with are interprocedural alias analysis and def-use analysis. The analyses are described using the specification language provided by Z1 and Z1 automatically generates analyzers to perform those analyses.. It is important to realize, when considering the results below, that these analyses occur over abstract domains that represent function pointers, memory addresses, integers, etc. In other words, these are complex and thus very time consuming.

The test programs—"matmul" (matrix multiplication), "gauss" (Gauss elimination), "simplex", and "amoeba", are widely used numerical programs. The programs, gauss, simplex, and amoeba are examples from a book of numerical algorithms [39]. TIS is a program from the Perfect Benchmarks [40]. The "wator" program that simulates a ecological system of fish & shark uses a large number of arrays and is used for testing generated analyzers in Z1. The others are test programs of our own.

The algorithms that we compare are: the algorithm that chooses a node randomly from the worklist,

Test program	Simple worklist	Queue worklist	Depth first ordering	Leading nodes first	Least recently evaluated	Guided-entailment algorithm	
testme	640[2]	640	552	619	627	179	(89[3])
(7^4 95^5 21^6)	0.11 [7]	0.10	0.08	0.08	0.11	0.03	
tune	1212	1213	1254	1220	1196	356	(177)
(8 124 15)	0.18	0.18	0.20	0.18	0.19	0.06	
dep3	2382	2386	2050	2320	2399	1115	(550)
(9 143 13)	0.33	0.32	0.26	0.29	0.32	0.14	
t2	4114	4108	3873	4203	3993	1200	(629)
(14 357 33)	1.14	1.08	1.01	1.08	1.06	0.21	
matmul	7840	7817	10196	7647	7666	2736	(1357)
(18 513 58)	2.31	2.21	2.56	2.12	2.19	0.47	
gauss I	42311	41946	37039	41986	43377	12940	(6821)
(34 1863 81)	16.22	14.53	12.19	14.17	16.59	5.30	
simplex	26451	26367	42034	27984	26762	3374	(1766)
(58 4662 221)	37.21	36.05	46.52	55.30	37.27	22.46	
gauss II	19928	19928	20412	18808	19570	2478	(1260)
(54 4710 172)	36.39	36.29	38.04	36.17	36.34	24.56	
wator	351912	354167	194340	284196	336007	17446	(87014)
(45 3467 166)	196.19	194.35	93.27	196.37	252.01	68.08	
TIS	343743	342057	153688	308144	304660	163068	(81026)
(102 6028 410)	219.54	225.38	83.19	178.15	214.36	88.12	
amoeba	615713	614656	457552	579028	582641	530715	(271291)
(36 6062 221)	521.41	553.35	575.47	676.19	467.39	369.55	

1 number of total iterations.
2 number of incomplete iterations.
3 number of procedures in the program.
4 number of expressions in the program.
5 number of places in the program.
6 cpu time (sec).

Table 1: Alias analysis

the algorithm that uses a depth-first ordering, the "leading nodes first" algorithm, and the algorithm that chooses the node which has been least recently evaluated. The last algorithm was suggested to us by Professor Daniel Weise at Stanford University.

7.2 Results and Discussion

The experimental results of alias analysis plus constant propagation[8], and def-use chain analysis are shown in Table 1 and 2, respectively. There are two metrics: the total number of iterations (evaluations) and cpu execution time, shown in the upper and lower part respectively. The number of iterations is greatly reduced by our method. In addition, because the guided-entailment method uses the suspended evaluation strategy, some incomplete evaluations that simply set up prestates and put successors on the

[8] constant propagation is automatically accomplished in the abstract interpretation framework

Test program	Simple worklist	Queue worklist	Depth first ordering	Leading nodes first	Least recently evaluated	Guided-entailment algorithm	
testme	656	656	567	635	643	194	(96)
(7 95 21)	0.12	0.12	0.10	0.11	0.13	0.04	
tune	1328	1329	1448	1298	1272	356	(177)
(8 124 15)	2.47	2.49	3.12	2.41	2.47	0.54	
dep3	2398	2402	2066	2337	2415	1115	(550)
(9 143 13)	0.34	0.34	0.28	0.34	0.38	0.15	
t2	4595	4591	4388	4591	4543	1255	(629)
(14 357 33)	1.22	1.16	1.12	1.20	1.23	0.21	
matmul	8798	8775	11143	8553	8541	2736	(1357)
(18 513 58)	2.47	2.49	3.12	2.41	2.47	0.54	
gauss I	53278	50986	43139	52984	54249	14795	(7340)
(34 1863 81)	24.20	22.39	18.48	22.16	25.09	6.40	
simplex	33379	26367	48870	27984	33562	3439	(1766)
(58 4662 221)	51.31	36.05	48.39	36.18	51.40	28.27	
gauss II	20464	20464	20799	19277	20135	2600	(1320)
(54 4710 172)	41.48	41.10	39.41	39.41	41.25	28.02	
wator	647981	354167	203460	284196	535736	226806	(112612)
(45 3467 166)	460.42	194.35	107.44	196.37	532.07	98.44	
TIS	405156	342057	205965	308144	357899	192134	(94742)
(102 6028 410)	278.59	224.38	116.03	178.15	286.04	109.08	
amoeba	1208327	1230333	1029806	985281	1071368	1036347	(518305)
(36 6062 221)	1013.25	1079.15	958.09	939.19	1860.48	718.20	

Table 2: Def-use chain analysis

worklist add to the iteration count artificially. Therefore, execution time gives a more precise comparison. The numbers of procedures, expressions, and places in each program are listed in the tables.

The performance of our algorithm varies among examples. Generally speaking, for complex programs, the ordering derived by the guided entailment algorithm will lead to the fixpoint much more quickly because the ordering without guidance may cause many ineffective evaluations. Note that the complexity of a program lies mainly in the structure of the program (e.g., cyclic call graphs and dynamic allocations). For numerical programs like simplex and amoeba, although there are many nested loops and expressions, the control-flow is simple and well structured (i.e., no dynamic allocation and few procedure calls), and therefore the speedups are not so impressive. In contrast, for programs with many procedure calls and much dynamic allocation (e.g., t2), the speedup are very significant.

Judging from the experimental results, our algorithm consistently outperforms the others. On the average, our algorithm is more than two times faster than the other algorithms. The experimental results also indicate that the performance of our algorithm is always better than the others for these test programs and analyses. None of the other algorithms is consistently better than the others because they are somewhat inflexible; they perform well for some examples, while they are inefficient for other examples. In contrast, our algorithm seems to order the evaluations efficiently for all these test programs.

8 Conclusion

The paper presents a practical contribution to abstract interpretation which is an important compiler optimization technique. We propose the entailment model for computing the fixpoints that arise in complex program analysis based on abstract interpretation. Then a guided-entailment algorithm for efficient computation of these fixpoints is presented. This algorithm is applicable to any abstract interpretation over domains of finite height and can solve complex inter- and intra- procedure program analysis without requiring the control flow graph a priori.

The *entailment graph* is constructed during analysis; it is precise and thus results in an efficient analysis. The strategies that underlie the algorithm are *waiting for all successors, leading node first, suspended evaluation,* and *back edge first.* Some strategies are similar to those embodied by interval analysis algorithms. Based on the strategies, our algorithm exploits local knowledge of the entailment graph to determine dynamically an effective order of evaluations. An iterative algorithm based on the model is designed and implemented. Experiments have been conducted to show the model is flexible and the associated algorithm is efficient. Results indicate that, on the average, it is more than two times faster than the applicable algorithms to which we compared it. The algorithm can efficiently handle complicated program analyses in the presence of difficult language constructs in a unified approach. That is, it is possible to efficiently compute general program analysis based on abstract interpretation, approaching the efficiency of dataflow methods that use interval analysis.

Although the algorithm developed in the paper is sequential, it can be easily adapted for parallel computation; there is inherent parallelism in the model. In particular, all the ready nodes on the worklist can be evaluated in parallel. In addition, all the nodes in the working set during loop propagation can also evaluated simultaneously. The successors of a node which are independent of one another in the functional $\mathcal{F}$ can be activated simultaneously.

References

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of program by construction or approximation of fixpoints. In *Conference Record of the ACM 4th Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[2] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[3] Samson Abramsky and Chris Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Series in Computers and Their Applications, 1987.

[4] Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation: An International Journal*, 2(3/4):179–396, 1989.

[5] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. In *Programs as Data Objects, Lecture Notes in Computer Science 217*, pages 43–62. Springer-Verlag, 1986.

[6] Neil D. Jones. Flow analysis of lazy higher-order functional programs. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 5. Ellis Horwood Series in Computers and Their Applications, 1987.

[7] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison Wesley, Reading, MA, 1986.

[8] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of ACM*, 23(1):158–171, 1976.

[9] A. Aho and J. Ullman. Node listings for reducible flowgraphs. *Journal of Computing System Science*, 13:286–299, 1976.

[10] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 28:679–694, 1987.

[11] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 2(1):146–159, 1972.

[12] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *JACM*, 23(1):172–202, January 1976.

[13] M. S. Hecht and J. D. Ullman. A simple algorithm for global dataflow problmes. *SIAM Journal of Computing*, 4(4):519–532, December 1977.

[14] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of ACM*, 19(3):137–147, August 1976.

[15] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transaction on Programming Language and System*, 9(3):319–349, 1987.

[16] Olin Shivers. Control flow analysis in Scheme. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 164–174, 1988.

[17] W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem classification. In *ACM 18th Symposium on Principles of Programming Languages*, pages 93–103, 1990.

[18] Williams Ludwell Harrison III and Zahira Ammarguellat. A program's eye view of Miprac. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 339–354, 1992. YALEU/DCS/RR-915.

[19] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural data flow analysis. In *ACM 20th Symposium on Principles of Programming Languages*, 1993.

[20] Paul Hudak. A Semantic Model of Reference Counting and its Abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[21] Li-Ling Chen and Williams Ludwell Harrison III. Efficient computation of fixpoints that arise in complex program analysis based on abstract interpretation. Technical Report 1245, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1992.

[22] Neil Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: abridged version. In *ACM 13th Symposium on Principles of Programming Languages*, pages 296–306, 1986.

[23] F. E. Allen. Control flow analysis. In *Sigplan Notice*, July 1970.

[24] R. Cytron, J. Ferrante, and B. K. Rosen. An efficient method of computing static single assignment form. In *ACM 16th Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[25] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM 8th Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[26] R. A. Ballance, A. B. Maccabe, and K. J.Otternstein. The program dependence web: A representation supporting control-, data-, and demand-diven interpretation of imperative languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 257–271, 1990.

[27] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: an algebraic approach to program dependencies. In *ACM 17th Symposium on Principles of Programming Languages*, pages 67–78, 1990.

[28] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *ACM 18th Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[29] Chris Clack and Simon L. Peyton Jones. Strictness analysis - a practical approach. In *IFIP Symposium on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 35–49, 1985.

[30] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In *Proc. 3rd International Conf. on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 274*, pages 426–445. Springer-Verlag, 1987.

[31] Sebastian Hunt and Chris Hankin. Fixed points and frontiers: a new perspective. *Journal of Functional Programming*, 1(1):91–120, 1991.

[32] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[33] G. L. Burn. *Abstract Interpretation and the Parallel Evaluation of Functional Languages.* PhD thesis, University of London, 1987.

[34] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[35] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages*, pages 164–174, 1993.

[36] J.-D. Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive inerprocedural computation of pointer-induced aliases and side effects. In *ACM 20th Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[37] Mark N. Wegman and Frank K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transaction on Programming Language and System*, 13(2):181–210, April 1991.

[38] B. G. Ryder and M. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, 1986.

[39] W. T. Vertterling *et al. Numerical Recipes: Example Book (C).* Cambridge University Press, 1988.

[40] M. Berry *et al.* The Perfect Club Benchmarks: effective performance evaluation of supercomputers. *Int'l. Journal of Supercomputer Applications*, 3(3):5–40, August 1989.