Equivalence Analysis: A General Technique to Improve the Efficiency of Data-flow Analyses in the Presence of Pointers

Donglin Liang and Mary Jean Harrold
{dliang,harrold}@cis.ohio-state.edu
The Ohio State University

Abstract

Existing methods to handle pointer variables in data-flow analyses can make data-flow analyses inefficient both in time and space, because the data-flow analyses must store and propagate the large amount of data facts introduced by dereferences of pointer variable. This paper presents equivalence analysis, a general technique to improve the efficiency of data-flow analyses in the presence of pointers. The technique explores equivalence relations among the memory locations accessed by a procedure and ensures that two equivalent memory locations share the same set of data facts in a procedure and the procedures called by that procedure. Thus, a data-flow analysis needs to compute the data-flow information only for a representative memory location in an equivalence class. The data-flow information for other memory locations in the equivalence class can be derived from that of the representative memory location.

Keywords: Alias analysis, data-flow analyses.

1 Introduction

Data-flow analyses, such as the computations of reaching definitions and of slices, have been widely used to provide information required for compiler optimization and for program understanding, debugging, and testing. Many of these analyses, however, become difficult when attempting to handle programs written in languages that allow pointer variables because of the aliasing that such varibles induce.[1] Thus, if data-flow analyses are to provide safe information, aliasing must be resolved,

Existing alias-analysis algorithms can be classified in two ways: *flow-sensitive* algorithms [3, 5, 10] provide precise aliasing information at each program point but are expensive; *flow-insensitive* algorithms [1, 8, 9] sacrifice precision for efficiency, and quickly compute less precise aliasing information that holds throughout a procedure or the entire program. Many researchers report ways to use aliasing information in data-flow analyses to ensure safety [2, 6, 7]. The method used by all these techniques to handle pointer variables in a program can be described as follows: when the data facts are computed for a statement, a dereference of pointer variables is replaced with the set of memory locations accessed by the dereference of the pointer variable. For example, if `p` can access `x`, `y`, and `z`, then statement "`*p=3;`" will create definitions for `x`, `y`, and `z`, and these techniques will essentially replace the definition of `*p` with definitions of `x`, `y`, and `z`.

Researchers have studied how the aliasing information computed by different aliasing analysis algorithms, with varying levels of precision, can affect the precision and efficiency of data-flow analyses [6, 7, 11]. The empirical results of these studies show that, through dereferences of pointer variables, the side-effects of a statement or a procedure can be greatly increased when the size of the subject programs is large. For example, Zhang et al. [11] report that, when Landi and Ryder's algorithm [5] is used to resolve the dereference of pointers, a statement in a program that contains approximately 13,000 lines of code can, on average, modify 20 memory locations through dereference of pointer variables. These empirical results also show that the side-effects of a statement or a procedure can be further increased if less precise aliasing information is used. For example, Landi et al. [6] report that, when aliasing information provided by a variant of Steensgaard's algorithm [9] is used to analyze the modification side-effects, a procedure in a program with 28,735 lines of code can, on average, modify 115 memory locations; if aliasing information provided by Landi and Ryder's algorithm is used, however, a procedure in this program modifies, on average, about 13 memory locations.

These empirical results imply that, without better techniques to handle pointer variables, many data-flow anal-

[1] *Aliasing* occurs when two different names access the same memory location at a point in a program

yses may be too inefficient for use in analyzing programs that use pointer variables, because these techniques may require storage and propatation of large sets of data-flow information. These empirical results also imply that, without better techniques to handle pointer variables, the time saved by using fast flow-insensitive aliasing analyses can be mitigated by the extra time to perform data-flow analyses with the less precise aliasing information.

To reduce this high cost of many data-flow analyses in the presence of pointer variables, we have developed a new technique, *equivalence analysis*. Equivalence analysis is based on the observation that memory locations pointed to by a pointer variable often behave in the same way for the data-flow analyses of a procedure. Equivalence analysis explores this equivalence relation among the memory locations for each procedure and partitions the memory locations referenced in the procedure into equivalence classes. A data-flow analysis must compute data-flow information for only one memory location that represents each equivalence class. Data-flow information about other memory locations in an equivalence class can be derived easily from the data-flow information computed for the representive memory location. Therefore, equivalence analysis can be used to improve the efficiency of data-flow analyses.

In this paper, we present our algorithm to perform interprocedural equivalence analysis. Our experiments with an implementation of our algorithm show that equivalence analysis can greatly speed-up several data-flow analyses in the presence of pointer variables. There are two major contributions of this paper: (1) a generic technique to improve the efficiency of data-flow analyses in the presence of pointers; (2) a practical way to use flow-insensitive aliasing information with data-flow analyses in large programs.

2 Equivalence Analysis

In this section, we first define the equivalence relation among memory locations in a program and discuss applications of the equivalence relations. Then, we present our algorithm to compute the equivalence classes of memory locations for each procedure.

2.1 Equivalence of memory locations and its applications

A memory location is accessed through an *object name* [5], which consists of a variable and a sequence of dereference operators and field access operators. We further classify object names and accesses. An object name is *indirect* if the object name consists of at least one dereference operator; otherwise, the object name is *direct*. A memory location l is *indirectly accessed* if l is accessed through an indirect object name.

```
int g;
f1(int *p) {
  *p = g+1;
  if( g>0 )
    p = &g;
  f2(p);
  *p++;
}
f2(int *q) {
  *q++;
  g++;
}
main() {
  int x,y,z;
  f1(&x);
  f1(&y);
  f2(&z);
}
```

Figure 1: Example Program

The mode of access of two memory locations can induce an equivalence relation between the locations. Location l_1 is *equivalent* to location l_2 in procedure P if, at any statement in P or in any procedure directly or indirectly called by P, (1) l_1 and l_2 are not accessed directly, and (2) whenever l_1 (l_2) is accessed through an indirect object name obj, l_2 (l_1) is also accessed through obj. For example, in procedure `f2()` of the program in Figure 1, variables `x, y,` and `z` are equivalent. Note that, under this definition, if location l is accessed directly in P or in any procedure directly or indirectly called by P, then l is equivalent only to itself. In some case, we might be interested in equivalence only when memory locations are being modified. Location l_1 is *mod equivalent* to location l_2 in a procedure P if, at any statement in P or procedures directly or indirectly called by P, whenever l_1 (l_2) is modified by the statement through an indirect object name, l_2 (l_1) is also modified by the statement. For example, in procedure `f1()` in Figure 1, `x` and `y` are mod equivalent. Like other equivalence relations, the memory-location equivalence we define here is reflexive, symmetric, and transitive.

Equivalence of memory locations can be used to improve the performance of many data-flow analyses: the set of memory locations accessed in a procedure can be partitioned into equivalence classes according to the equivalence relation; members of an equivalence class then share the same data facts at each statement. Because members of the same equivalence class share the same data facts at each statement, an intraprocedural data-flow analysis would compute the same data-flow information for each memory location in the equivalence class. The data-flow analysis can be improved by only computing the data-flow information for a representative memory location in an equivalence class. Using the information computed for the representative memory location, the data-flow analysis can then derive data-flow information about other memory locations in the equivalence class, For example, locations referenced by `f2()` in Figure 1) can be divided into two equivalence classes, { `x, y, z`} and { `g` }. To compute reaching definitions for `f2()`, the analysis only computes the reaching definitions for `x` and `g`. In `f2()`, `x` is defined at statement 9, and this definition can reach statements 10 and 11; thus, the analysis concludes that `x`, `y` and `z` are defined at statement 9 and these definitions can reach statement 10 and 11.

Interprocedural data-flow analyses also can be improved by only computing and storing information for a representative memory location in each equivalence class. Many interprocedural data-flow analyses summarize the result computed by an intraprocedural phase for nonlocal memory locations accessed by a procedure P. In such interprocedural analyses, when a callsite to P is processed, the summary information can be used (or computed on-demand) to compute the effect of P on the callsite. Because memory locations in an equivalence class share the same data facts in P, they also share the same summary information. Thus, this kind of data-flow analyses can be improved by computing and storing summary information for a representative memory location of each equivalence class. At the callsite to P, to find the summary information for a memory location l, we only need to determine the equivalence class that contains l in P, and then find the summary information for that equivalence class. For example, to represent summary information of `f2()` at statement 6 in the construction of the system dependence graph [4], we need to create only one actual-in parameter vertex and one actual-out parameter vertex to represent both `x` and `y`, which are in one equivalence class in `f2()`.

2.2 Equivalence analysis algorithm

Our algorithm computes a set of equivalence classes for each procedure in a program. Like the algorithm that computes GMOD/GREF sets for these procedures [], our algorithm has an intraprocedural phase and an interprocedural phase.

In the intraprocedural phase, the algorithm processes each statement in each procedure P and collects the direct object names and the indirect object names that are referenced at the statement. Each indirect object name classifies memory locations referenced in P into three categories: non-accessed, everywhere-accessed, and may-be-accessed. A location l is *non-accessed* by indirect object name *obj* in P if, at any statement that references obj, l is not aliased to obj. A location l is *everywhere-accessed* by indirect object name *obj* if, at any statement that references obj, l is aliased to obj. A location l is *may-be-accessed* by indirect name *obj* if l is neither non-accessed nor everywhere-accessed by obj. For example, in `f1()` (Figure 1), if we use Landi and Ryder's alias analysis algorithm [5] to resolve pointer dereference, then `x` and `y` are everywhere-accessed by `*p`, and `g` is may-be-accessed by `*p`.

The equivalence of memory locations is then computed by two rules:

1. a memory location that is accessed by a direct object name or that may-be-accessed by any indirect object name is not equivalent to any other memory location;
2. location l_1 is equivalent to location l_2 if, for each indirect object name *obj* in P, both l_1 and l_2 are everywhere-accessed by *obj* or both l_1 and l_2 are non-accessed by obj.

The algorithm computes a set of initial equivalence classes for P using these rules.

In the interprocedural phase, the algorithm processes the call statements in each procedure P and propagates information from the called procedures to P. To process a call statement C that calls P', the algorithm first identifies the set, S, of nonlocal memory locations accessed by P'. S is partitioned into a set of equivalence classes according to the equivalence relation computed for P'. The algorithm then combines S into the set of equivalence classes computed for P using the following rules:

1. if l_1 and l_2 are not present in P's equivalence classes so far and l_1 and l_2 are equivalent in S, then l_1 and l_2 are in one equivalence class in P;
2. if l_1 and l_2 are present in one equivalence class of P and l_1 and l_2 are equivalent in S', then l_1 and l_2 are in one equivalence class in P;
3. if l_1 and l_2 are present in one equivalence class of P and l_1 and l_2 are not in S', then l_1 and l_2 are in one equivalence class;
4. in other cases, l_1 and l_2 can not be in one equivalence class.

The algorithm iterates over the procedures until the equivalence classes for each procedure become stabilized.

Fig 2 shows the algorithm `EquivAnalysis` that computes equivalence classes for each procedure in a program. `EquivAnalysis` builds a virtual matrix for each procedure and uses the matrix to compute the equivalence classes. In the matrix, the rows represent memory locations and columns represent object names that appear in the procedure or equivalence classes of memory locations. The elements of the matrix are marks: O_mark, X_mark, and unmarked. O_mark means that (1) the direct object name represented by the column accesses the location represented by the row; (2) the equivalence class represented by the column contains the location represented by the row; or (3) the indirect object name represented by the column fully-access the location represented by the row. X_mark means that the indirect object name represented by the column might-be-access the location represented by the row. If neither

O_mark nor X_mark applies, the element remains unmarked. Given such a matrix, locations l_1 and l_2 are equivalent if and only if 1) there is no X_mark on the two rows that represent l_1 and l_2; and 2) the two rows are marked with O_mark at the same positions.

`EquivAnalysis` dynamically completes the matrix to efficiently compute the equivalence classes. `EquivAnalysis` encodes the matrix in a set of equivalence classes so that it does not need to explicitly store the matrix. `EquivAnalysis` calls `Update()` to add a new column to the matrix that is encoded in the equivalence classes. `Update()` checks the marks of the locations in each equivalence class that is *touched* by the marking, that is, at least one location in the equivalence class has been marked. If two locations l_1 and l_2 in an equivalence class E have not been marked or have been marked with O_mark in the new column, then l_1 and l_2 remain in the same equivalence class after adding a new column. Otherwise, l_1 and l_2 need to be in separate equivalence classes after adding the new column.

algorithm `EquivAnalysis`
input $\mathcal{P}$: program to be analyzed
output a set of equivalence classes for each procedure
declare P_i, P_j: procedures in $\mathcal{P}$
$\mathcal{E}_i$: a list of equivalence classes of locations for P_i
W : a list of procedures, sorted reverse topologically on the strongly-connected components of the call graph
begin `EquivAnalysis`
1. **foreach** procedure P_i in $\mathcal{P}$ **do**
2. put P_i in W
3. **foreach** statement in P_i **do**
4. collect the direct object names;
5. collect indirect object names;
6. **endfor**
7. **foreach** direct object name obj **do**
8. put $\{obj\}$ in $\mathcal{E}_i$;
9. **endfor**
10. **foreach** indirect object name obj **do**
11. mark the locations always-accessed by obj with O_mark;
12. mark the locations mightbe-accessed by obj with X_mark;
13. E_1 := {locations accessed by obj but not in any set on $\mathcal{E}_i$};
14. insert E_1 into $\mathcal{E}_i$;
15. `Update(`$\mathcal{E}_i$`);`
16. **endfor**
17. **endfor**
18. **while** $W \neq \phi$ **do**
19. P_i = remove procedure from head of W
20. **foreach** callsite C to P_j in P_i **do**
21. find S, the set of non-local locations in P_j's set of equivalence classes;
22. **foreach** equivalence class E in S **do**
23. mark locations in E with O_mark;
24. E_1 := {locations in E but not in any set on $\mathcal{E}_i$};
25. insert E_1 into $\mathcal{E}_i$;
26. `Update(`$\mathcal{E}_i$`);`
27. **if** $\mathcal{E}_i$ changes **then**
28. put P_i's callers in W;
29. **endif**
30. **endfor**
31. **endfor**
32. **endwhile**
end `EquivAnalysis`
procedure `Update(`$\mathcal{E}$`)`
input $\mathcal{E}$: a list of equivalence classes
output updated $\mathcal{E}$
begin `Update`
1. **foreach** location set E_c in $\mathcal{E}$ touched by marking **do**
2. E_2 := {locations in E_c without being marked};
3. E_3 := {locations in E_c with O_mark};
4. E_4 := {locations in E_c with X_mark};
5. reset marks for locations;
6. delete E_c;
7. put E_2, E_3 into $\mathcal{E}$;
8. **foreach** location l in E_4 **do**
9. put $\{l\}$ in $\mathcal{E}$;
10. **endfor**
11. **endfor**
end `Update`

Figure 2: `EquivAnalysis`: Equivalence analysis algorithm.

Lines 1–17 shows the intraprocedural phase. In this phase, `EquivAnalysis` collects direct object names and indirect object names referenced at each statement. For each direct object name, `EquivAnalysis` updates the equivalence class by creating a single item equivalence class with the memory location accessed by the object name and adding into the equivalence class set. For each indirect object name, first, `EquivAnalysis` adds new rows into the matrix to represent the memory locations that are not present in the matrix. Then, `EquivAnalysis` creates a new column to represent the indirect object name and calls `Update()` to add the column into the matrix.

Lines 18–32 shows the interprocedural phase. In this phase, `EquivAnalysis` iterates over the procedures. For each procedure P_i, `EquivAnalysis` checks each call C to procedure P_j. `EquivAnalysis` computes the nonlocal memory locations that are referenced by P_j at C and divides the set of memory locations into a set of equivalence classes according the the equivalence relations computed for P_j. `EquivAnalysis` then processes each of these equivalence classes in a similar way as to process an indirect object name. If the equivalence classes have changed, `EquivAnalysis` puts the procedures that calls P_i into the worklist for further updates.

2.3 Complexity

Theoretically, a program can have $O(2^n)$ distinguishable memory locations [9], where n is the size of the program. This argument can make our algorithm exponential in time and space to the size of the program. To be more practical, we assume that the number of memory locations in a program is $O(n)$.

`Update()` takes $O(n)$ because it is possible that each equivalence class in $\mathcal{E}$ is touched by marking. In practice, we expect that such a worst case would seldom happen, and we expect `Update()` takes constant time.

In the intraprocedural phase (lines 1-17), lines 4-5, line 8, and lines 11-15 can be executed $O(n)$ times. The total time of $O(n)$ executions of lines 4-5 and line 8 is also $O(n)$. For lines 11, 12, 13, and 15, the time to execute each of this line is $O(n)$. Therefore the intraprocedural phase has $O(n^2)$ time complexity.

In the interprocedural phase (lines 18-32), the information is propagated from the called procedure to the calling procedure. In the absence of recursion, information is propagated through a call only once, thus, $O(c)$ propagations are needed, where c is the number of calls in the program. In the presence of recursion, information is propagated through a call when there is a change in the set of equivalence classes of the called procedure. Because a change in the set of equivalence classes can add a memory location, or divide one equivalence class into two equivalence classes, only $O(n)$ of these kinds of changes can occur to the equivalence classes of a procedure. Thus, at most $O(n * c)$ propagations are needed in this case. Each propagation requires $O(n)$ calls to `Update()`. [2] Therefore, in the absence of recursion, the interprocedural phase has $O(c*n^2)$ time complexity; in the presence of recursion, the interprocedural phase has $O(c * n^3)$ time complexity.

The time complexity of our algorithm is $O(c * n^2)$ in the absence of recursion; the time complexity of our algorithm is $O(c * n^3)$ in the presence of recursion.

3 Conclusions

We presented a technique that can be used to improve many data-flow analyses in the presence of pointers. We are conducting experiments to evaluate the effectiveness of our technique.

References

[1] L.O. Andersen. Program analysis and specialization for the c programming language. Technical Report 94-19, University of Copenhagen, 1994.

[2] D. Atkinson and W. Griswold. Effective whole=program analysis in the presence of pointers. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundation of Software Engineering*, November 1998.

[3] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[5] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of 1992 ACM symposium on Programming Language Design and Implementation*, pages 235–248, June 1992.

[6] W. A. Landi, B. G. Ryder, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. Technical Report DCS-TR-336, Rutgers University, May 1998.

[7] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis 4th International Symposium, SAS '97, Lecture Notes in Computer Science Vol 1302*, September 1997.

[8] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the 24th ACM symposium on Principles of Programming Languages*, pages 1–14, 1997.

[9] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proc. of the Int. Conf. on Compiler Construction*, 1996.

[10] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[11] S. Zhang, B. G. Ryder, and W. Landi. Experiments with combined analysis for pointer aliasing. In *Program Analysis for Software Tools and Engineering '98*, pages 11–18, 1998.

[2] We can replace these calls with a procedure with $O(n * log(n))$ time complexity. But we found that using these calls are more efficient in practice because only a few groups are touched in each call.