Modular Concrete Type-inference for Statically Typed Object-oriented Programming Languages*

Ramkrishna Chatterjee   Barbara G. Ryder

Department of Computer Science, Rutgers University, Piscataway, NJ 08855 USA,
Fax: 732 445 0537, {ramkrish,ryder}@cs.rutgers.edu

Abstract. The problem of concrete type-inference for statically typed object-oriented programming languages (e.g., Java, C^{++}) determines at each program point, those objects to which a reference may refer or a pointer may point during execution. We present a new technique called *analysis-using-abstract-values* which performs modular and demand-driven concrete type-inference of a robust subset of Java without threads and exceptions and C^{++} without exceptions. Our algorithm is provably precise on programs with only single-level types[2] and without dynamic dispatch, and has the worst-case complexity of $O(n^4)$ which is an improvement over the $O(n^7)$ worst-case bound achievable by applying previous approaches of [RHS95] and [LR91] to this case. For general programs, the algorithm is polynomial-time and computes a safe solution.

1 Introduction

The problem of concrete type-inference for statically typed object-oriented programming languages (e.g., Java, C^{++}) is to determine at each program point, those objects to which a reference may refer or a pointer may point during execution. This information has many uses including static resolution of dynamically dispatched calls, side-effect analysis, testing, program slicing and aggressive compiler optimization.

The problem of concrete type-inference is both intraprocedurally and interprocedurally flow-sensitive. However, there are approaches with various degrees of flow-sensitivity for this problem. Although some of these have been used for pointer analysis of C, they can be adapted for the type-inference problem for Java without exceptions and threads or C^{++} without exceptions. There are intraprocedurally and interprocedurally flow-insensitive approaches [BCCH94, Wei80, Ste96b, Ste96a, SH97, ZRL96, And94], which are the

* The research reported here was supported, in part, by NSF grant GER-9023628 and the Hewlett-Packard Corporation.

[2] types with data members only of primitive type

least expensive, but also the most imprecise. In contrast, there are intraprocedurally and interprocedurally flow-sensitive approaches [Deu94, LR92, EGH94, WL95, CBC93, MLR+93, PC94, PS91, Ruf95, GDDC97], which are the most precise, but also the most expensive.

An intraprocedurally flow-insensitive algorithm does not distinguish between program points within a method, and hence it reports the same solution for all program points. In contrast, an intraprocedurally flow-sensitive algorithm computes different (sometimes approximate) solutions for distinct program points.

An interprocedurally flow-sensitive (i.e. context-sensitive) algorithm considers (sometimes approximately) only interprocedurally *realizable paths* [SP81, RHS95, LR91]: paths along which calls and returns are properly matched, while an interprocedurally flow-insensitive (i.e. context-insensitive) algorithm does not make this distinction. The example in Appendix E illustrates this difference. For the rest of this paper, we will use the term *flow-sensitive* to refer to a intra- and interprocedurally flow-sensitive analysis.

The precision of the type-inference information obtained directly affects its utility for the applications mentioned above. Thus, in this paper, we are interested in a flow-sensitive algorithm for concrete type-inference. However, flow-sensitive (sometimes even flow-insensitive) algorithms are generally memory intensive and frequently run out of memory while analyzing large programs. This motivates *modular* analysis. The goals of a *modular* algorithm are two-fold:

- analysis of a complete program by keeping only a part of the program in memory at a time, and
- analysis of partial programs, such as libraries.

We introduce a new technique called *analysis-using-abstract-values* which achieves these goals. To the best of our knowledge, this is the first *modular* flow-sensitive algorithm for concrete type-inference of a subset of Java without exceptions[3] and threads and C++ without exceptions.

The main results in this paper are:

- (Sections 3, 4.1 and 5) a new technique called *analysis-using-abstract-values* which facilitates *modular* flow-sensitive concrete type-inference of Java and C++[4];
- (Sections 4.3 and 4.4) a proof that this technique computes the precise solution for programs with only single-level types and without dynamic dispatch, and has the worst-case complexity of $O(n^4)$ which improves upon the $O(n^7)$

[3] In [CRL97], we show how to do concrete type-inference in the presence of exceptions.

[4] In this paper, we present our algorithm only for Java.

worst-case bound achievable by applying previous techniques of [RHS95] and [LR91] to this case;
- (Section 6) a new method for demand-driven computation using *analysis-using-abstract-values*; and
- (Section 7) preliminary empirical evidence for effectiveness of *analysis-using-abstract-values* for *modular* analysis.

The rest of this paper is organized as follows. We start with some definitions. Next, we introduce *analysis-using-abstract-values* by considering programs with only single-level types and without dynamic-dispatch. We prove that this technique computes the precise solution in this case and its worst-case complexity is $O(n^4)$. After this, we present *analysis-using-abstract-values* for programs with general types and dynamic-dispatch. Next, we show how by using *analysis-using-abstract-values*, a part of the solution can be computed on demand. Finally, we present some preliminary empirical evidence for effectiveness of *analysis-using-abstract-values* for modular analysis.

2 Basic definitions

2.1 Precise solution for concrete type-inference

A *reference variable* is one of the following:

- a static variable (class variable) of reference [5] type;
- a local variable of reference type;
- Av, where Av is $V[t_1]...[t_d]$, and
  - V is a static/local variable or V is an array obj_n allocated at program point n, such that V is either a d-dimensional array of reference type or an array of any type having more than d dimensions and
  - each t_i is a non-negative integer; or
- $V.s_1...s_k$, where
  - V is either a static/local variable of reference type or V is Av or V is object obj_n created at program point n,
  - for $1 \leq i \leq k$, each $V.s_1...s_{i-1}$ (V for $i = 1$) has the type of a reference to a class T_i and each s_i is a field of reference type of T_i or $s_i = f_i[t_{i_1}]...[t_{i_{r_i}}]$ and f_i is a field of T_i and f_i is an array having at least r_i dimensions and each t_{i_j} is a non-negative integer, and
  - $V.s_1...s_k$ is of reference type.

[5] may refer to an instance of a class or an array

Using these definitions, the *precise solution*, up to symbolic execution, for concrete type-inference can be defined as follows: given a *reference variable* RV and an object obj identified by its creation site, $\langle RV, obj \rangle$ belongs to the *precise* solution at a program point n if and only if RV is visible at n and there exists an execution path from the *start-node* of the program to n (under the usual assumption of data-flow analysis[Bar78]: the result of a test is independent of previous tests and all the branches are possible), such that if this path is followed, RV points to obj at n (i.e. at the top of n).

2.2 Program representation

Our algorithm operates on an interprocedural control flow graph or ICFG [LR91]. An ICFG contains a control flow graph (CFG) for each method in the program. Each statement in a method is represented by a node in the method's CFG. Each call site is represented using a pair of nodes: a call-node and a return-node. Information flows from a call-node to the entry-node of a target method and comes back from the exit-node of the target method to the return-node corresponding to the call-node.

We will denote the entry-node of *main* by *start-node* in the rest of this paper.

2.3 Subset of Java

We essentially consider a subset of Java that excludes threads and exceptions, but in some cases we may need to exclude three other features: *finalize* methods, static initializations and dynamically defined classes.

- If a *finalize* method does not modify any variable of reference, it can be ignored during analysis (and no solution will be reported for it). Otherwise, since *finalize* methods are non-deterministically called (during garbage collection or unloading classes), such programs (which are extremely rare) cannot be handled by our algorithm.
- Static initializations complicate analysis due to dynamic loading of classes. If static initializations can be done by going through them in program order, our algorithm can handle them. Otherwise, if they depend upon dynamic loading (extremely rare), our algorithm cannot handle them.
- Java allows classes to be defined on the fly and dynamically loaded. Our algorithm cannot handle such classes which are not known statically.

2.4 Single-level type

Definition 1 *A single-level type is one of the following:*

– *A primitive type defined in [GJS96]. For example, int, float etc.*
– *A class whose all non-static data-members have primitive types. For example, class A { int i,j; }.*
– *An array of a primitive type.*

2.5 Safe solution

Definition 2 *An algorithm is said to compute a safe solution for concrete type-inference if and only if at each program point, the solution computed by the algorithm is a superset of the precise solution.*

3 Analysis-Using-Abstract-Values

In this section, we introduce the basic idea behind *analysis-using-abstract-values* with an example. Consider the following example program.

```
class A {};
class test {
 public static A p, q;
 public static void proc0( ) {
    n0: p = q;
    exit_proc0:
 };
 public static void proc1( ) {
    l1: q = new A;
    c1: proc0();
    n1:
 };
 ....
 public static void procn( ) {
    ln: q = new A;
    cn: proc0();
    nn:
 };
}
```

At *exit_proc0*, p may point to any *object_li*, $i \in \{1, ..., n\}$. *analysis-using-abstract-values* stores this information abstractly as $\langle p,\ q\_init \rangle$ at *exit_proc0*. This means p points to whatever q pointed to at the entry of *proc0*, and it summarizes the effect of *proc0* on variables of reference type. This single data-flow element is then passed to the return node of each *ci*, where it is instantiated with the value of q before the call. This is the essence of *analysis-using-abstract-values*. We will see in the following sections that this use of abstract values increases efficiency and precision and facilitates *modular* analysis.

4 *analysis-using-abstract-values* for programs with only single-level types and without dynamic dispatch

analysis-using-abstract-values is an iterative data-flow analysis algorithm [MR90], but unlike ordinary data-flow analysis, data-flow elements are propagated in three distinct phases as explained below. The lattice for data-flow analysis is a subset lattice with union as the *meet* operator and sets of *points-to*s as lattice elements. For programs with only single-level types, a *points-to* has the form $\langle var, object \rangle$, where *var* is one of the following:

- a static variable of reference type or
- a local variable of reference type,

and *object* is one of the following:

- *object_n* - object (or array) created at program point *n* or
- *variable_init* - (unknown) *abstract* initial value of a global (static) variable or parameter *variable* at the entry-node of a method.

4.1 Algorithm

The algorithm consists of the following steps:

1. The directed acyclic graph of strongly connected components of the call graph, SCCG, is constructed.
2. **Phase I:** The nodes of SCCG are traversed in a reverse topological order (bottom-up) and for each node *SCC* the following iterative algorithm - whose pseudo-code is given in Figures 1, 2, 3, 4, 5 and 6 - is executed:
   - Each method is analyzed assuming that the parameters and global (static) variables have some unknown *abstract* initial values.
   - When a points-to involving a global variable is passed from the exit-node of a method to a return-node of a call-node for this method, if the value of the variable is an initial value of one of the global variables or one of the parameters, it is replaced by the values of that global variable or parameter at the corresponding call site. Points-tos at the exit-node of a method represent the effect of the method on variables of reference type in terms of the initial values of global variables and parameters. These represent the **transfer function** of the method for variables of reference type. Since there is a cyclic dependence between the transfer functions of methods belonging to the same node of SCCG, these functions need to be computed simultaneously and iteratively until a fixed point is reached.

Transfer functions for methods belonging to different nodes of SCCG have a hierarchical dependence, so they can be computed by a post-order traversal of SCCG.

3. **Phase II:** The nodes of SCCG are traversed in a topological order (top-down) to propagate initial values for global variables and parameters to entry-nodes. For methods belonging to the same node of SCCG, values are propagated iteratively until a fixed point is reached. Note that only entry-nodes, exit-nodes and call-nodes are explored in this phase.
4. **Phase III:** At each non-entry node, the value of a variable of reference type is computed by replacing the initial values of global variables and parameters by sets of their values computed in the previous step.

Appendix C contains an extensive example which illustrates the above algorithm.

4.2 Modular analysis

Each node of SCCG - and hence each method - needs to be in memory only thrice: first during bottom-up traversal, second during top-down propagation phase and finally when solutions for non-entry nodes are computed. The rest of the time, only the solution at the exit-node of a method (which represents its transfer function during Phase I) or the solution at the entry-node of a method (during Phase II) needs to be in memory. Hence this is a *modular* approach and requires less memory than whole-program analysis techniques. In other techniques, until the final solution is computed, a procedure cannot be moved out of memory without the possibility of its being needed again. As a result, if the whole program is not kept in memory, there is no *a priori* (constant) bound on the number of times a procedure needs to be moved into or out of memory.

Although, in this paper, we present *analysis-using-abstract-values* as a flow-sensitive algorithm, the technique is more general and can also be used for *modular* analysis that is context-sensitive but intraprocedurally flow-insensitive.

The maximum number of exit/entry-nodes which need to be in memory simultaneously, in addition to the node of SCCG being analyzed, depends upon the order of traversal of SCCG. At present we use a simple depth-first ordering. We plan to further investigate the complexity of finding the best possible topological order for steps 2 and 3 of the algorithm and find out whether this is solvable in polynomial time.

4.3 Proof of Precision

In this section, we present an outline of a proof which shows that the algorithm described in Section 4.1 computes a precise solution for programs with only

```
initialize-worklist( SCC ) {
 for ( each entry-node ent of each method in SCC ) {
   for ( each parameter and global variable v visible at ent ) {
     wl-node = new WorkListNode( ent, <v,v_init> );
     add wl_node to worklist;
   }
 }

 for ( each method M in SCC ) {
   for ( each node N in M, which allocates an object ) {
     // N is var = new type;
     for ( each successor succ of N ) {
       add-to-solution-and-worklist-if-needed( <var,object_N>, succ );
     }
   }

   for ( each call-node c-node in M ) {
     if ( c-node calls a method m not in SCC ) {
       // r-node is the return-node of c-node
       for ( each df-elm in the solution at the exit-node of m
             such that df-elm.var is a global variable and
             df-elm.object is not an initial value ) {
         add-to-solution-and-worklist-if-needed( df-elm, r-node );
       }
     }
   }
 }
}
```

Fig. 1. *initialize-worklist*

single-level types and without dynamically dispatched calls. The overall structure of the proof is as follows. First, we show that phase I is precise. Next, we prove that phase II is precise. Finally, assuming the first two phases are precise, we show that the whole algorithm is precise.

In this proof, we assume that each node is reachable from the *start-node*. This can be easily checked by a separate algorithm which iteratively propagates a single data-flow element *reachable* to those nodes which are reachable from the *start-node*.

To prove that phase I is precise, we need to define what we mean by *precision* for this phase because it computes in terms of abstract values. Let n be a program point in a method P. Further, let v be a global variable (for simplicity, we consider only global variables in this proof; the argument for parameters is similar), and x_init be the initial value of a global variable x, (which may be same as v) at the entry of P. If there exits a balanced [6] execution path from the entry of P

[6] i.e. along which each entry-node has a matching exit-node except possibly for the

```
initialize-worklist( SCC );
while ( worklist is not empty ) {
  delete wl_node from worklist;

  node = worklist.node;
  ndf-elm = worklist.data-flow-element;

  if ( node is not a call-node or the exit-node of a method ) {
     // compute the effect of the statement associated with node
     // on ndf-elm.
     generated-data-flow-elements = apply( node, ndf-elm );

     for ( each successor succ of node ) {
        for ( each df-elm in generated-data-flow-elements ) {
           add-to-solution-and-worklist-if-needed( df-elm, succ );
        }
     }
  }

  if ( node is a call-node ) {
     process-call-node( ndf-elm, node );
  }

  if ( node is an exit-node ) {
     process-exit-node( ndf-elm, node );
  }

}
```

Fig. 2. main loop of phase I

to n, such that if we follow this path, v will have the value x_init at n assuming x has the value x_init at the entry of P, then $\langle v,\ x\_init\ \rangle$ belongs to the precise solution at n. Now let *val_conc* be a *concrete value* (i.e. it is not an unknown initial value). Again suppose there exists a balanced execution path from entry of P to n such that if we follow this path then v has value *val_conc* at n. Then $\langle v,$ $val\_conc\ \rangle$ also belongs to the precise solution at n. A points-to belongs to the precise solution at n if and only if it satisfies one of the above two conditions.

It is easy to show that the solution computed by phase I is a subset of the precise solution as defined above using induction on the number of iterations needed to compute a points-to. Similarly, using induction on the length of a balanced path associated with a points-to in the precise solution, we can show that the precise solution is a subset of the solution computed by phase I. Hence phase I computes the precise solution.

A precise solution for phase II is defined as follows. Let P be a method, x be

initial entry-node of P; this corresponds to the notion of balanced path in [RHS95].

```
process-call-node( rdf-elm, c-node ) {
  // c-node is a call-node which invokes method M

  if ( rdf-elm.var is an actual for parameter param ) {
    for ( each df-elm at the exit-node of M such that df-elm.object is
          param_init and df-elm.var is a global variable or
          return-variable of M ) {
       // each method has a return-variable to which the
       // value returned by the method is assigned

       if ( df-elm.var is a global variable )
         // r-node is the return-node for c-node
         add-to-solution-and-worklist-if-needed(
             <df-elm.var,rdf-elm.object>, r-node );
       else
         // at c-node, result-var stores the value returned by M
         add-to-solution-and-worklist-if-needed(
             <c-node.result-var,rdf-elm.object>, r-node );
     }
  }

  if ( rdf-elm.var is a global variable v ) {
     for ( each df-elm at the exit-node of M such that df-elm.object is
           v_init and df-elm.var is a global variable or return-variable
           of M ) {

       if ( df-elm.var is a global variable )
         add-to-solution-and-worklist-if-needed(
             <df-elm.var,rdf-elm.object>, r-node );
       else
         add-to-solution-and-worklist-if-needed(
             <c-node.result-var,rdf-elm.object>, r-node );
     }
  }

  if ( df-elm.var is a local variable ) {
     add-to-solution-and-worklist-if-needed( df-elm, r-node );
  }
}
```

Fig. 3. process-call-node for single-level types and without dynamic dispatch

a global variable, and *val_conc* be a *concrete value*. If there exists an execution path from the *start-node* of the program to the entry-node of P, such that if we follow this path then x has the value *val_conc* at the entry of P, then $\langle x, val\_conc \rangle$ belongs to the precise solution at the entry of P. Again using induction on the number of iterations and the length of a path associated with a points-to, and the fact that we have already proved phase I is precise, we can show that phase II is precise according to the preceding definition.

```
process-exit-node( rdf-elm, ex-node )
// ex-node is the exit-node of method M
{
  if ( rdf-elm.var is a global variable or the return-variable of M ) {
    for ( each call-node c-node of M in the current SCC ) {
      if ( rdf-elm.object is an initial value ) {
        for ( each value val of rdf-elm.object at c-node ) {
          if ( rdf-elm.var is a global variable ) {
            // r-node is the return-node of c-node
            add-to-solution-and-worklist-if-needed(
               <rdf-elm.var,val>, r-node );
          }
          else {
            add-to-solution-and-worklist-if-needed(
               <c-node.result-var,val>, r-node );
          }
        }
      }
      else {
        if ( rdf-elm.var is a global variable ) {
          add-to-solution-and-worklist-if-needed( rdf-elm, r-node );
        }
        else {
          add-to-solution-and-worklist-if-needed(
              <c-node.result-var,rdf-elm.object>, r-node );
        }
      }
    }
  } // else rdf-elm.var is a local variable
}
```

Fig. 4. process-exit-node for single-level types and without dynamic dispatch

Now assuming that we have proved that the first two phases are precise, we show that the whole algorithm is precise. Let n be a program point in a method P, v be a global variable, and *val_conc* be a *concrete value*. Suppose there exists an execution path, S, from the *start-node* of the program to n, such that if we follow this path then v has the value *val_conc* at n. So $\langle v,\ val\_conc\ \rangle$ belongs to the precise solution at n. There are two cases:

1. Let *ent* be the last occurrence of the entry-node of P along S. In this case we assume that there exists a program point k after *ent* on s, such that *val_conc* is stored in a variable at k and then passed to v through a series of assignment statements. But this means $\langle v,\ val\_conc\ \rangle$ belongs to the precise solution of phase I. Hence it is in the solution computed by the algorithm.
2. In this case we assume that a global x (which may be same as v) has the value *val_conc* at *ent*, and through a series (possibly none) of copy statements

```
set of points-tos apply( node, rdf-elm )
{ set of points-tos retVal;
  if ( node assigns to a variable of reference type ) {
     // node represents lhs-var = rhs-var or lhs-var = new type
     if ( rdf-elm.var == rhs-var of node ) {
         add <lhs-var,rdf-elm.object> to retVal;
     }

     if ( rdf-elm.var != lhs-var )
       // preserved
       add rdf-elm to retVal;
     // else killed
  }
  else {
     add ref-elm to retVal;
  }

  return retVal;
}
```

Fig. 5. apply for single-level types

```
add-to-solution-and-worklist-if-needed( rdf-elm, node ) {
 // This function checks if rdf-elm is present in the solution computed
 // at node so far. If not, it adds rdf-elm to the solution set of node
 // and puts node along with rdf-elm on the worklist.
}
```

Fig. 6. add-to-solution-and-worklist-if-needed

(after *ent*) this value is passed to v. This means $\langle x,\ val\_conc\ \rangle$ belongs to the precise solution of phase II, and $\langle v,\ x\_init\ \rangle$ belongs to the precise solution of phase I. Hence after the expansion phase $\langle v,\ val\_conc\ \rangle$ will be in the solution computed by the algorithm.

Hence the precise solution is a subset of the solution computed by the single-level algorithm.

Now suppose $\langle v,\ val\_conc\ \rangle$ is computed by the algorithm at program point n. Again there are two cases:

1. Suppose $\langle v,\ val\_conc\ \rangle$ was computed by phase I. Then there exists a balanced execution path from the entry-node of P to n, such that if we follow this path then v has the value *val_conc* at n. Assuming that the entry-node of P is reachable from the *start-node* of the program, there exists an execution

path from the *start-node* to n, such that if we follow this path then v has the value *val_conc* at n. Hence $\langle v,\ val\_conc \rangle$ belongs to the precise solution.

2. Suppose $\langle v,\ x\_init \rangle$ was computed by phase I, and then *x_init* was replaced by *val_conc* in the expansion phase. It means there exists an execution path from the entry-node of P to n, such that along this path the initial value of x at the entry-node of P flows to v. Also since $\langle x,\ val\_conc \rangle$ was computed by phase II, it means there exists an execution path from *start-node* to the entry-node of P, such that if we follow this path then x has the value *val_conc* at the entry of P. Combining these two paths, we get an execution path from *start-node* to n, such that if we follow this path then v has the value *val_conc* at n. Hence $\langle v,\ val\_conc \rangle$ belongs to the precise solution.

Thus, the precise solution is a superset of the solution computed by the single-level algorithm. Hence the algorithm computes the precise solution.

4.4 Complexity

In Appendix A, we show that the worst-case complexity of *analysis-using-abstract-values* for programs with only single-level types and without dynamically dispatched calls is $O(n^4)$, where n is approximately the number of statements in the program.

5 *analysis-using-abstract-values* in the presence of general types and dynamic dispatch

In this section, we present *analysis-using-abstract-values* for the subset of Java defined in section 2.3. The overall structure of this algorithm is same as that of the single-level algorithm described in Section 4 and it is an extension of the single-level algorithm. In the following paragraphs, we describe these extensions. First we consider non-recursive general types. Later in Section 5.11, we will show how to handle recursive types.

5.1 Call graph decomposition

The initial call graph used for obtaining strongly connected components is constructed by resolving dynamically dispatched calls using class hierarchy analysis [DMM96] and by considering only instantiated types [BS96]. Given a dynamically dispatched call site, the class hierarchy is scanned to find all the methods which override the method which would have been called in the case of static dispatch. This set is then refined by excluding those types which are not instantiated in the program as in [BS96]. The resulting set is then used as the set of methods callable from this site. The initial call graph need not be precise; we only need an overestimate in order to ensure the safety of the final solution.

5.2 Basic scheme for handling general types

Consider the following example:

```
class A {
  public  C member1;
  public  C member2;
};
class test {
  public static void copy( A param1, A param2 ) {
   C p;
   l1: param1.member1 = param2.member2 ;
   l2: p = param2.member1;
  }
}
```

Let the initial values of *param1* and *param2* be *param1_init* and *param2_init* respectively. At *l1*, the value in *member2* of *param2_init* is copied to *member1* of *param1_init*. However, *param1_init* could be same as *param2_init*. Thus, this statement can potentially modify *member1* of *param2_init* in addition to *member1* of *param1_init*. This is the basic problem in the presence of general types: potential modification of a field of an initial value when the same field of another initial value is modified.

The above problem is solved as follows. The potential modification to *param2_init.member1* is recorded conditioned upon the equality of *param1_init* and *param2_init* as shown below:

```
void copy( A param1, A param2 ) {
   C p;
   l1: param1.member1 = param2.member2 ;
   l2: p = param2.member1;
   exit_copy:
   /* <param1_init.member1,param2_init.member2_init>,
      <(param1_init == param2_init),<param2_init.member1,
       param2_init.member2_init>>,
      <(param1_init != param2_init),param2_init.member1,
       param2_init.member1_init>,
      <(param1_init == param2_init), p, param2_init.member2_init>,
      <(param1_init != param2_init), p, param2_init.member1_init> */
}
```

The data-flow element $\langle$*(param1_init == param2_init)*, $\langle$*param2_init.member1*, *param2_init.member2_init* $\rangle\rangle$ is relevant to a call site of *copy* if and only if *param1* and *param2* have the same value at the call site. In general, a data-flow element at the exit-node of a method is relevant to a call site of the method if and only if the condition associated with the data-flow element holds at the call site.

Thus, the data-flow elements have one of the following two forms:

1. $\langle$*var, object*$\rangle$, where
   - *var* is
     - a global (static) or local variable of reference type,
     - *object_n.f*, field *f* of reference type of object created at program point n,
     - *init_val.f*, field *f* of reference type of an abstract initial value,
     - *object_n*, array created at program point n (all locations in an array are represented by a single object), or
     - *init_val*, an abstract initial value of array type; and
   - *object* is
     - *object_n*, an object (array) created at program point n or
     - *init_val*, an abstract initial value of class or array type.
2. $\langle$*condition,* $\langle$*var, object*$\rangle\rangle$, where
   - $\langle$*var, object*$\rangle$ is as defined above; and
   - *condition* is *(object1 op object2)*, where
     - *op* is == or *!=*,
     - *object1* is *object_n* (an object or array created at program point n) or is an abstract initial value of class or array type, and
     - *object2* is an abstract initial value of class or array type.

The data-flow elements of the first form have an *empty* condition associated with them, meaning that they hold in all contexts.

Since the conditions are inferred in a bottom-up manner rather than being propagated in a top-down manner, only those conditions which are *relevant* are inferred, instead of all possible conditions which may exist at the entry-node of a method. For example, the relationship between *param1_init.member1_init* and *param1_init.member2_init* is not inferred as it is not relevant, although they may be the same at a call site of *copy*. Hence, the number of such conditions is determined (in the worst case) by the statements reachable from a method rather than the size of the whole program, which makes this approach *scalable.* Moreover, this bottom-up inference facilitates *modular analysis* by allowing the analysis of complete programs by keeping only a part of the program in memory at a time and analysis of partial programs, such as libraries.

If a conjunction of conditions is associated with a points-to, any fixed-size subset of these conditions may be stored without affecting safety. At present, we store only one condition with a points-to.

5.3 Dynamically dispatched calls

Dynamically dispatched calls can be handled as shown below:

```
class D {};

class A {
  public static D x;
  public void foo( ) { l: x = new D; };
};

class B extends A {
  public void foo( ) { };
};

class C extends A{
};

class test {
   public static void proc0( A param )
      // <A.x,x_init>
   {
      l1: param.foo();

      exit_proc0: // <(param_init,A.foo),<A.x,object_l>>,
                  // <(param_init,B.foo),<A.x,x_init>>.
   }
}
```

In the above example, at the exit of *test.proc0*, *A.x* points to *object_l* if the run-time type of *param_init* is such that *A.foo* gets called at *l1*, while its value remains unchanged if the run-time type is such that *B.foo* gets called. Hence, the values of *A.x* at the exit-node of *test.proc0* are inferred conditioned on the type of *param_init*. Since the same method may be called for many different types (e.g., *A.foo* gets called for both A and C), the conditions encode the relationship between *param_init* and the actual methods rather than just the type of *param_init*. In C^{++}, calls through function pointers can be handled similarly.

5.4 *apply*

Figure 7 contains a high-level description of *apply* for the general case. Contrast it with the definition of *apply* for the single-level case given in Figure 5. As stated before, *apply* computes the effect of a (assignment) statement on an incoming data-flow element. For example, suppose node l represents the statement $p.f1 = q$, *ndf_elm* (defined in Figure 2) is $\langle (z), \langle p,\ object\_s \rangle \rangle$, $\langle (u), \langle q,\ object\_n \rangle \rangle$ is present in the solution computed at l so far and the conditions z and u are *compatible*. Two conditions are *compatible* if and only if they do not imply different values for the same abstract value. For example, ($x\_init == y\_init$) and (x_init != y_init) are incompatible. *apply* generates $\langle$(z,u), $\langle object\_s.f1,\ object\_n \rangle \rangle$, which means $\langle object\_s.f1,\ object\_n \rangle$ holds at the bottom of l under the condition that both z and u hold at the entry-node of the method containing l. However, as we stated earlier, any subset of fixed size of the set of conditions associated with a data-flow element may be chosen without affecting safety. At present we choose a subset of size 1, so assuming u is chosen, $\langle u, \langle$ *object_s.f1, object_n* $\rangle\rangle$ will be generated.

Whenever a field f of an abstract initial value av is modified by a statement, the field f of all the abstract initial values of the same type as av, or of a type that is a subtype or a supertype of the type of av, and the field f of all the *concrete* objects of the same type as av or of a type that is a subtype of the type av are also conditionally modified under the conditions that av is same as these abstract initial values or *concrete* objects, and appropriate data-flow elements are generated to represent these modifications. Similarly, if a field f of a *concrete* object obj is modified by a statement, the field f of all the abstract initial values of the same type as obj or of a type that is a supertype of the type of obj are also conditionally modified. For instance, in the example in Section 5.2, at *l1*, *param2_init.field1* is conditionally modified when *param1_init.field1* is modified. The fields of abstract initial values considered for conditional modification are those that are used in the method containing the assignment statement, or in one of the methods callable from this method through actual formal bindings, as shown in Section 5.6.

Note that we can always choose the maximum number of field selectors appearing in an operand in a statement to be a fixed constant, say c, (even 1) because using temporary variables, any statement with more field selectors can be broken into a sequence of statements, each having at most c field selectors. This will increase the size of the program by at most a factor of the maximum number of field selectors appearing in the original program, and might degrade the precision a little without compromising safety.

5.5 Repropagation

Consider the following program fragment:

```
class B {};
class C extends B {};
class A {
 public B field;
};
class test {
 public static void method( A param ) {
   l1: param.field = new B;
   if ( _ )
     param = new A;
   l2: param.field = new C;
 };
};
```

```
set of data-flow-elements apply( node, rDFE )
{
 set of data-flow-elements retVal;
 if ( node does not assign to any pointer ) {
   add rDFE to retVal;
 }
 else {
  lhs_set = combine those data-flow elements in the solution set
  computed at node so far (including rDFE) whose conditions are
  compatible to generate the set of objects represented by the
  left-hand-side.
  // Note: each element in lhs_set has a set (i.e. conjunction)
  // of conditions associated with it

  similarly compute rhs_set for the right-hand-side.

  retVal = combine compatible elements from lhs_set and rhs_set.
  // only one of the conditions associated with a
  // resulting points-to is chosen as its condition

  if ( rDFE cannot be killed by this node )
    add rDFE to retVal;
  else
     // following function is defined in Section 5.5
     check-if-repropagation-is-needed( rdf-elm, node );
 }

 return retVal;
}
```

Fig. 7. *apply* for the general case

First suppose that the *if* statement between l1 and l2 is absent. Under this assumption, assignment to *param_init.field* at *l2* *kills* the previous assignment to it at *l1*. However, if the *if* statement is present, assignment to *param_init.field* at *l1* is not killed by the assignment at *l2* because there is a path to *l2* for which *param* does not point to *param_init*. Since until a fixed point is reached, only a partial solution is available at a node, the decision for killing (sometimes) cannot be made as soon a data-flow element reaches a node. Thus, when a data-flow element can be killed by a node according to the solution computed so far, but may not be killed according to the (larger) fixed-point solution, the algorithm does not propagate this data-flow element to the successors of this node, instead it marks the node and data-flow element. After the worklist becomes empty, the algorithm revisits the marked nodes to check if the marked data-flow elements are killed according to the current solution. If not, it unmarks these points-tos and nodes, and restarts iteration to propagate these points-tos. Propagation stops when none of the marked nodes require repropagation (i.e. they kill the marked points-tos according to the fixed-point solution).

A previous assignment to the field of a *concrete* object cannot be killed by a subsequent assignment because the previous assignment may have been made to a different run-time object than the one to which the subsequent assignment is made. However, assignments to the fields of abstract values can be killed because for a given call to a method, an abstract value represents the same run-time object at all points in the method. This improves precision over other techniques in many situations [CR97].

5.6 Demand-driven generation of initial (abstract) values

In Phase I of the single-level algorithm, the worklist is initialized with the initial values of all parameters and globals at all the entry-nodes of all the methods contained in a SCC. However, for the general case, this approach is not feasible because there are too many initial values of fields to consider (an infinite number for recursive types). Hence a different approach is taken. The worklist is initialized only with the initial values of global variables and parameters that are explicitly used in a method.

The algorithm generates other initial values in a *demand-driven* manner. When a field f of an initial value iv is accessed for the first time during data-flow analysis, $\langle iv.f,\ iv.f\_init \rangle$ is propagated from the entry-node of the method containing the (call or assignment) statement where the field is accessed. If access to *iv.f* implies accesses to new fields of initial values of any caller of this method, then these fields are similarly propagated from the entry-node of the caller. This is done iteratively using the worklist. A similar approach is used for global variables accessed by methods called from a method. As a result, a field, a global variable or a parameter that is not accessed in a method (or any method callable from it) does not appear in the solution computed by phase I. For instance, in the example in Section 5.2, *param1.member2* does not appear in the solution for *copy*.

5.7 *process_call_node*

Figure 8 contains a high-level description of *process-call-node* for dynamically dispatched call sites. Call sites of static methods are handled similarly. Conditions associated with a method callable from a dynamically dispatched call site specify conditions on the types of initial values as shown in Section 5.3. For simplicity, we have ignored the case when *df-elm.var* is the *return-variable* of *M*. This case is handled in a way similar to the way it is handled in Figure 3.

```
process-call-node( rdf-elm, c-node ) {

 bindings = compute-initial-value-actual-value-bindings( rdf-elm, c-node );

 for ( each receiver object recv at c-node ) {
   for ( each method M callable using recv ) {
     Let cond be the condition on recv under which M is invoked;
     if ( M is found to be callable from c-node for the first time ) {
       propagate-dfelms-at-exit-node( M, c-node, cond );
     }
     else {
       if ( cond is a new condition for M at c-node ) {
         propagate-dfelms-with-empty-conds( M, c-node, cond );
       }
       for ( each bind in bindings ) {
         for ( each df-elm at the exit-node of M such that df-elm contains
               bind.initial-value and df-elm.var is not a local variable of
               M ) {
           // r-node is the return-node for c-node
           gen-points-tos =
                instantiate-abstract-values( df-elm, bind, cond );
           for ( each gdf-elm in gen-points-tos ) {
             add-to-solution-and-worklist-if-needed( gdf-elm, r-node );
           }
         }
       }
     }
   }
 }

  if (rdf-elm.var is a local variable) {
     // cannot be killed by the call
     add-to-solution-and-worklist-if-needed( rdf-elm, r-node );
  }
  else
     check-if-repropagation-is-needed( rdf-elm, c-node );
}
```

Fig. 8. *process-call-node* for the general case

compute-initial-value-actual-value-bindings checks if *rdf-elm* determines the value of any (unknown) initial value used by a method callable from *c-node*. If so, it stores the binding between *rdf-elm.object* and that initial value in a table called the *BindingTable*. The condition under which the binding holds is also stored with the binding. If this binding implies other bindings, then those are also stored in the *BindingTable*. *compute-initial-value-actual-value-bindings* returns the set of newly created bindings.

propagate-dfelms-at-exit-node propagates data-flow elements that do not represent values of local variables from the exit-node of *M* to *r-node*. It instantiates the initial values appearing in these data-flow elements using the bindings stored in the *BindingTable*. If M is a static method not contained in the current SCC, the propagation of data-flow elements that do not contain any initial values and do not represent values of local variables from the exit-node of M to *r-node* is done during the initialization of the worklist for phase I, as in Figure 1.

propagate-dfelms-with-empty-conds propagates data-flow elements that have *empty* condition associated with them and that do not represent values of local variables from the exit-node of M to *r-node*. The initial values appearing in these data-flow elements are instantiated using the bindings stored in the *BindingTable*. *cond* is used as the condition for the generated data-flow elements.

instantiate-abstract-values replaces, if possible, all the abstract values appearing in *df-elm* with their values computed so far at *c-node*. It combines (through conjunction) *cond*, the conditions of *bind* and *df-elm*, and the conditions associated with the bindings between abstract values and their values at *c-node* (as stored in *BindingTable*) to produce the complete condition of a generated data-flow element. If after instantiation, the complete condition associated with a generated data-flow element turns out to be false (e.g, due to incompatible conjuncts or false conjuncts), the data-flow element is discarded. Otherwise, one conjunct is chosen from the complete condition as the condition for the generated data-flow-element. If all the abstract values in *df-elm* cannot be instantiated with the values computed at *c-node* so far, *gen-points-to* is empty.

When a data-flow element at the exit-node of a method depends upon the simultaneous occurrence of more than one data-flow element at the entry-node of the method, in many situations, the use of abstract values improves precision over other techniques like [PR96, LR92] (which represent context using *concrete* values) by reducing the number of data-flow elements propagated along *unrealizable* paths. Details of this are given in [CR97].

5.8 *process_exit_node*

Figure 9 contains a high-level description of *process-exit-node* for the general case. *instantiate* is similar to *instantiate-abstract-values* in Figure 8. If *rdf-elm* represents the value returned by M, *instantiate* uses the *result-variable* of *c-node* as the *var* for the generated points-tos. While computing conditions associated with the elements of *generated-points-tos*, *instantiate* considers each *cond* under which M could be called from *c-node*; if there is no such *cond* yet, *c-node* is skipped. *instantiate* substitutes the abstract values in *rdf-elm* with their values computed at *c-node* so far. As before, data-flow elements are generated if and only if values for all the abstract values in *rdf-elm* are available, and data-flow elements whose conditions are false are discarded.

```
process-exit-node( rdf-elm, exit-node )
// Let exit-node be the exit-node of method M
{
 if ( rdf-elm.var is not a local variable ) {
   for ( each possible call-node c-node of M in the
         current SCC ) {
     // r-node is the return-node of c-node
     generated-points-tos = instantiate( rdf-elm, c-node );
     for ( each gdf-elm in generated-points-tos ) {
       add-to-solution-and-worklist-if-needed( gdf-elm, r-node );
     }
   }
 }
}
```

Fig. 9. *process-exit-node* for the general case

5.9 Phase II

During Phase II, for each initial value *concrete-val* of an abstract value *av* computed at the entry-node of a method *M*, the algorithm given in Figure 10 is executed. For each dynamically dispatched call site in *M*, *eval_conds_assoc_with_dyn_dispatched_calls* evaluates the conditions (see Section 5.3) associated with methods callable from this site.

propagate-init-values instantiates all the bindings containing initial values of *M*, which are stored in the *BindingTable* at *c-node*, using the values of initial values computed at the entry-node of *M*. The values of initial values of *m* determined by those bindings whose conditions are true are propagated to the entry-node of *m*.

For libraries, an initial value at the entry-node of a method that could be invoked from outside the library is called an *interface initial value* or *IIV*. During Phase II, an *IIV* is treated like a *concrete value* and propagated to the entry-nodes of other methods. Conservative worst-case assumptions are made in evaluating conditions involving an *IIV*. Presently, *analysis-using-abstract-values* needs to know the complete class hierarchy for analyzing libraries; but, this technique could be extended to avoid this in some situations.

5.10 Phase III

The algorithm for phase III is shown in Figure 11. *instantiate* substitutes each abstract value appearing in *df-elm* with their concrete values computed at the entry-node of *M* in phase II. A concrete points-to is generated if and only if the condition associated with *df-elm* (if any) evaluates to true.

```
propagate-initial-value( av, concrete-val ) {
 eval_conds_assoc_with_dyn_dispatched_calls( av, concrete-val );

 for ( each call-node c-node in M ) {

   for ( each method m callable from c-node such that at least one of the
         conditions under which m could be called has evaluated to true ) {

     for ( each initial value iv (at the entry-node of m) and actual value
           act-val binding involving av stored in BindingTable at c-node ) {

       evaluate the condition associated with the binding and possible
         values of act-val using initial values computed at the entry-node
         of M so far and concrete-val as the value for av;

       if ( the condition associated with the binding evaluates to true ) {
          propagate the values of act-val computed above to the entry-node
            of m as initial values of iv;
       }
     }  // end of each binding
   }    // end of each method callable from c-node
 }      // end of each call-node c-node in M
}

eval_conds_assoc_with_dyn_dispatched_calls( av, concrete-val ) {
 for ( each call-node c-node in M ) {

  for ( each method m callable from c-node ) {

   for ( each condition cond under which m is callable from c-node ) {

     if ( cond contains av ) {
       evaluate cond using concrete-val as value for av;
       if ( cond evaluates to true ) {
         mark cond as true;
         if ( for m, cond is the first condition which has
              evaluated to true ) {
           propagate-init-values( c-node, m );
         }
       }
     }
   } // end of each condition
  }  // end of each method m callable from c-node
 }   // end of each call-node c-node in M
}
```

Fig. 10. Phase II for the general case

Suppose *N* is a call-node. If *conc-df-elm.var* is a global variable or a field of an object or an array object that is not used[7] by at least one of the methods callable from *N* or it is a field of an *IIV* or an *IIV* of array type, then the tuple (*conc-df-elm*,*N*) is stored in a global table called *DemandTable*. The information stored in the *DemandTable* is used for computing a part of the solution on demand as shown later in Section 6.

```
for ( each method M ) {
 for ( each node N in M ) {
  for ( each df-elm in the solution computed at N ) {
     concrete-df-elms = instantiate( df-elm, M );
     for ( each conc-df-elm in concrete-df-elms ) {
       add conc-df-elm to final solution set of N;
       if ( N is a call-node ) {
          add-to-DemandTable-if-needed( conc-df-elm, N );
       }
     }
  }
 }
}
```

Fig. 11. Phase III for the general case

5.11 Recursive Types

The initial values defined with respect to the initial value of a parameter or a global variable comprise the *space of initial values* associated with it. For example:

```
class B {          class D {};  class A {
 public D fld1;                  public B member1;
 public D fld2;                 };
};
proc( A param ) {
/* The space of initial values associated with
   param_init consists of param_init, param_init.member1_init,
   param_init.member1_init.fld1_init and
   param_init.member1_init.fld2_init */
}
```

In the presence of recursive types, this *space* could be potentially infinite as shown in the following example.

[7] i.e. it is not the *var* field of any points-to computed by Phase I for at least one of the callable methods.

```
class B {                 class A {
 public A parent;          public B child;
};                        };

proc( A param ) {
/* The space of initial values associated with
   param_init consists of param_init,
   param_init.child_init, param_init.child_init.parent_init,
   param_init.child_init.parent_init.child_init and
   so on */
}
```

For *analysis-using-abstract-values* we need to represent any arbitrary point in a *space of initial values*; thus in the presence of recursive types, this requires a method for keeping any *space of initial values* finite. We have devised a way of dividing initial values into equivalence classes such that each class has a representative element, with the invariant that any points-to involving the representative of a class is taken to involve any member of that same class.

To intuitively describe our algorithm, suppose we have a variable name $param\_init.f_1\_init.f_2\_init \ldots f_k\_init$ in the space of initial values associated with the initial value *param_init* of a parameter *param*. To find its equivalence class and its representative name, we need to consider this variable name as a path through the infinite tree of names which could be possibly constructed from the type of the root name *param_init*. Each internal node has as its children, all the fields of its type. A path through this tree to a node is a variable name of the above form. Our algorithm essentially collapses specific paths in this tree which end in the same type to one path, thus handling any recursion in the type definition. There are many ways of performing this collapse; our choice is just one of them.

All members of an equivalence class are of the same type. Equivalence classes which contain elements of the same type are distinguished by the field selectors from the root type. The equivalence class representative can be characterized as the variable name in that class with no repeated types (i.e. f_i_init and *param_init* have different types); intuitively any "repeating" same sequences of certain types have been "collapsed". This ensures that the number of equivalence classes is finite (see Section 5.12).

As a rule of thumb, given a variable name $param\_init.f_1\_init.f_2\_init \ldots f_k\_init$, we know that in its equivalence class all variables are of type $type(f_k\_init)$, and all begin with field selector f_1. Looking at the tree in which this name is a path, we can select each field f_i in turn, for $i = 1..k$, as we traverse the path corresponding to the name in the tree starting at the root. We build the representative name during this tree traversal by discarding repetitive subpaths we explore. Given the selected field f_i, we look forward on the path until we find the last field (closest to the end of the path) with the same type as type(f_i), say it is f_s. Then we delete $f_{i+1}\_init, \ldots, f_s\_init$ in the name of the representative which we are building in this manner. This process continues until we reach the end of the name. Now we have the representative name for our original variable name.

Appendix B contains pseudo-code and an example for this scheme.

The following example shows how these equivalence classes are used to represent the transfer function of a method.

```
class A {
 public B fld1;
 public B fld2;
 public A( ) {
   fld1 = fld2 = null;
 };
};

class B {
 public C fld1;
 public A fld2;
 public B( ) {
   fld1 = fld2 = null;
 };
};

class C {
};

class test {
 static B x;
 static void proc0( ){
  A p;
  B q;
  l1: p = new A();
  l2: q = new B();
  p.fld1 = q;
  l3: p.fld1.fld2 = new A();
  proc1( p );
  /* [param_init] consists of obj_l1 and obj_l3. So
     [param_init].fld2 represents obj_l1.fld2 and
     obj_l3.fld2. Hence, both <obj_l1.fld2,x_init>
     and <obj_l3.fld2,x_init> belong to the solution
     at this point. */
 }
 static void proc1( A param ){
    A tmp;
    tmp = param;
    while( tmp != null ) {
      tmp.fld2 = x;
      if ( tmp.fld1 != null )
        tmp = tmp.fld1.fld2;
      else
        break;
    }
   // <[param_init].fld2,x_init>
 }
}
```

5.12 Safety and complexity of *analysis-using-abstract-values* in the general case

In this section, we briefly outline the reasons for the safety and polynomial-time complexity of *analysis-using-abstract-values* in the general case.

All approximations made by *analysis-using-abstract-values* are safe, which ensures that the computed solution is a superset of the precise solution. These approximations are:

- Using a subset of conjuncts associated with a data-flow element. This is safe because at a program point where this data-flow element is used, if all the conjuncts are true then any subset of the conjuncts is also true. However, this may cause overestimation of solution at program points where only a proper subset of the conjuncts is true.
- Using a data-flow element containing an equivalence class for all elements in the equivalence class.
- Whenever any data-flow element depends upon the simultaneous holding of two or more data-flow elements at a program point, assume they do. For example, in phase II, if a condition associated with a method callable from a call site evaluates to true, all instantiations of abstract values at the entry-node of the method containing the call site are propagated to the entry-node of the called method, irrespective of whether the propagated instantiation and the instantiation which makes the condition true hold simultaneously at the entry-node of the method containing the call site or not.
- Never killing assignments to the fields of heap objects because the previous assignment may have been made to a different run-time object than the one to which the current assignment is being made. However, note that the assignments to fields of initial values can be killed and this is an advantage of using abstract values.
- Representing all locations of an array by a single location and hence not killing assignments to any location of an array object.
- Not killing assignments to the field of an equivalence class because the equivalence class represents a set of objects and the previous assignment may have been made to the field of a different member than the one to whose field the current assignment is being made.
- *!=* evaluates to true if both operands are the same heap object because they could be different run-time objects. However, *!=* evaluates to false if both operands are the same abstract object.

Using Landi's result [Lan92] about undecidability of may alias analysis, it can be shown that concrete type-inference in the general case is undecidable. Thus, we can only hope for a safe algorithm for the general case.

Now consider the complexity of *analysis-using-abstract-values*. It is easy to see that if the total number of data-flow elements generated is polynomial in the size of the program, then *analysis-using-abstract-values* does polynomial (in the program size) amount of work for each generated data-flow element, and hence it takes polynomial amount of time in the worst case. But the algorithm as presented may generate an exponential number of abstract initial values. This can be easily avoided by enforcing a bound on the length of abstract initial values (analogous to *k-limiting* [JM82]). All initial values of the same type having lengths greater than this bound will be considered to be in the same equivalence class. This will ensure that the number of data-flow elements is polynomial in the size of the program.

6 Demand-driven concrete type-inference

analysis-using-abstract-values facilitates demand-driven computation in two ways: (1) phase III, the expansion phase, can be done on demand, and (2) the *DemandTable* may be used to compute a part of the solution on demand.

Demand-driven phase III: The phase III of *analysis-using-abstract-values* needs to be performed only at those program points where the solution of concrete type-inference is needed. For example, in order to statically resolve frequently executed dynamically dispatched calls, we need the solution only at those program points that are *hot-spots* and that contain dynamically dispatched calls that cannot be resolved using cheaper techniques like class hierarchy analysis.

Demand-driven computation using the DemandTable: The solution computed at a program point by *analysis-using-abstract-values* is sufficient to find the values of variables or fields used by a method directly or indirectly through a call. Many important applications like static resolution of dynamically dispatched calls and side-effect analysis need only this information. However, if the values of variables or fields not used by a method are needed (e.g., for detecting dangling pointers in C^{++}), information in the *DemandTable* can be used to compute these on demand. The following example illustrates this.

```
class A {
  public B field1;
  public B field2;
};
class test {
  public static void method( A param ) {
    B p;
    l5: p = param.field1;
  };
```

```
  public static void method1( A param ) {
    method( param );
  };
  public static void main( void ) {
    A local;
    l1: local = new A;
    l2: local.field1 = new B;
    l3: local.field2 = new B;
    l4: method1( local );
  };
};
```

Phase I computes $\langle p,\ param\_init.field1\_init\ \rangle$ at the exit-node of *method*, and *param_init.field2_init* does not appear in the solution computed for *method* or *method1*. As a result, during Phase II, the value of *param_init.field2_init* at the call site *l4*, which is *object_l3*, is not propagated to the entry-node of *method1*. Instead, the tuple $(\langle object\_l1.field2,\ object\_l3\ \rangle,\ l4)$ is stored in the *DemandTable*. If we want to know the value of *param_init.field2_init* at *l5*, then since the solution computed at *l5* tells us that *param_init* is *object_l1*, we lookup the *DemandTable* for the value of *object_l1.field2*. The result of this lookup tells us that that *object_l1.field2* points to *object_l3* at the call site *l4*. Now we check if the entry-node of *method* is reachable from *l4*. Since it is reachable, *object_l3* is a valid value for *param_init.field2_init* at *l4*. Further details of how a query is answered are given in Appendix D.

Call graph refinement Checking reachability in the absence of dynamically dispatched calls is easy; as it reduces to checking reachability on the call graph. However, in the presence of dynamically dispatched calls, checking reachability on the call graph gives an approximate, but safe solution. In order to improve precision, we use the following scheme. The DAG of strongly connected components of the call graph is traversed in a reverse topological order (bottom-up) and for each strongly connected component *SCC*, the algorithm shown in Figure 12 is executed. For each method, the algorithm computes the methods reachable from it and the conditions on the initial values at the entry-node of the method (or *empty* condition, meaning it is independent of initial values) under which these methods are reachable. When the set of methods reachable from a call-node C in a method M is needed, it is computed on demand using the solution sets (computed in Figure 12) of the methods callable from C. The conditions associated with the elements of these solution sets are translated into conditions on the initial values of M (or *empty*) using the bindings in the *BindingTable* at C and evaluated using the *concrete values* computed at the entry-node of M. The set of methods reachable from C is determined by those conditions which evaluate to true.

The following terms have been used in Figure 12:

- Each *WorkListNode* contains a *caller* method, a *callee* method and a condition *cond* on an initial value of *caller* under which *caller* calls *callee*. Note that *cond* may be *empty*.
- *m.calls* contains the solution (for reachability) for method *m*. Each element *s* of the solution set is a tuple, where *s.callee* represents a method callable from *m* and *s.cond* represents the condition under which this method is callable.
- *c-node.method* represents the method containing the call-node *c-node*.
- *translate_conds(c-node, m, cond)* translates the condition *cond* into a set of conditions using the initial-value to actual-value bindings for *m* stored in the *BindingTable* at *c-node* i.e. it translates *cond* into conditions on the initial values at the entry-node of the method containing *c-node*. Here *m* is a method callable from *c-node*.

7 Implementation

At present we are implementing *analysis-using-abstract-values* for a Java-like subset of C^{++}. We have completed the implementation of the algorithm for single-level types and the implementation for the general types is in progress. Table 1 contains data about call graph decomposition of 9 C^{++}programs obtained from Hemant Pande [PR96], David Bacon [BS96] and Ben Zorn [CGZ95]. We used hierarchy analysis for constructing the initial call graphs. Table 1 shows that the call graphs of these programs have a large number of components and the cycles in these call graphs are of small size. This is encouraging evidence for effectiveness of *analysis-using-abstract-values* for modular analysis.

8 Related Work

There are two kinds of related work: (1) those dealing with pointer analysis and concrete type-inference, and (2) those dealing with demand-driven data-flow analysis.

Related work of the first kind has already been discussed in the introduction. Unlike other flow-sensitive techniques for pointer analysis/concrete type-inference, *analysis-using-abstract-values* is a modular and demand-driven analysis technique, and hence scalable. Moreover, *analysis-using-abstract-values* provides the best known worst-case bound for programs with only single-level types

```
// initialize worklist
for ( each method M in SCC ) {
  for ( each call-node c-node in M ) {
    for ( each method m callable from c-node ) {
      for ( each condition cond under which m is callable
            from c-node ) {
        wl-node = new WorkListNode( M, cond, m );
        add wl_node to worklist;
      }
      if ( m is not in SCC ) {
        for ( each s in m.calls ) {
          if ( s.cond is empty ) {
            for ( each condition cond under which m is callable
                  from c-node ) {
              add-to-solution-and-worklist-if-needed( M, cond, s.callee );
            }
          }
          else {
            conditions = translate_conds( c-node, m, s.cond );
            for ( each cond in conditions ) {
              add-to-solution-and-worklist-if-needed( M, cond, s.callee );
            }
          }
        }
      }
    }  // end of each method m callable from c-node
  }    // end of each call-node c-node in M
}      // end of each method M in SCC


while ( worklist is not empty ) {
 delete wl_node from worklist;

 for ( each call site c-node in SCC from which wl_node.caller
       is callable ) {
   if ( wl_node.cond is empty ) {
     for ( each condition cond under which wl_node.caller
           is callable from c-node ) {
       add-to-solution-and-worklist-if-needed( c-node.method, cond, wl_node.callee );
     }
   }
   else {
     conditions = translate_conds( c-node, wl_node.caller, wl_node.cond );
     for ( each cond in conditions ) {
       add-to-solution-and-worklist-if-needed( c-node.method, cond, wl_node.callee );
     }
   }
 }
}
```

Fig. 12. Algorithm for checking reachability on the call graph

programs	number of nodes	number of methods	number of SCCs	maximum size of a SCC
deriv1	316	41	37	3
employ	512	72	72	1
deriv2	574	44	40	3
chess	437	47	47	1
deltablue	1258	102	102	1
richards	836	83	83	1
FeynDiagram	3976	237	225	4
vvector	1780	128	128	1
vmatrix1	1223	100	100	1

Table 1. Call graph decomposition

and without dynamic dispatch (the only known natural polynomial-time solvable special case [LR91, PR96, CRL97], if exceptions and threads are excluded).

Related work of the second kind include [DGS95] and [HRS95]. In both of these, each data-flow element is treated uniformly and computed from scratch on demand, although later queries may be answered more efficiently because of caching. However, *analysis-using-abstract-values* divides data-flow elements into two classes: (1) those for the values of variables or fields used by a method, and (2) those for the values of variables or fields not used by a method. This distinction is important because many significant applications do not need data-flow elements of the second kind. Moreover, the data-flow elements of the second kind can be computed by checking reachability on the call graph rather than by performing data-flow analysis from scratch. Thus, *analysis-using-abstract-values* introduces a new paradigm for demand-driven computation: preprocessing followed by queries. Phase I, phase II, the construction of the *DemandTable* and the computation of methods reachable from a method can be seen as preprocessing, which helps in answering queries on demand quickly without data-flow analysis. In addition, *analysis-using-abstract-values* combines demand-driven computation with modular analysis. Nevertheless, [DGS95] and [HRS95] are demand-analysis frameworks for general data-flow problems, whereas we are restricting our attention to concrete type-inference. Other data-flow problems (e.g, def-use associations) can also be handled using this technique and we plan to investigate these in the future.

9 Conclusion and Future Work

We have presented a new technique called *analysis-using-abstract-values* for modular and demand-driven concrete type-inference of Java without threads and exceptions and C^{++}without exceptions. We have shown that this technique is a modular analysis technique, allowing analysis of complete programs while only a part of the program need be in memory at one time. The technique can also be applied to partial programs like libraries. We have shown how to compute a part of the solution on demand using this technique. Further, we have proved that *analysis-using-abstract-values* computes the precise solution for programs with only single-level types and without dynamic dispatch, and has the worst-case complexity of $O(n^4)$ which is an improvement over the $O(n^7)$ worst-case bound achievable by applying previous techniques of [RHS95] and [LR91] to this case. For general programs, the algorithm is polynomial-time and computes a safe solution.

We have presented empirical data about call graph decomposition of 9 C^{++}programs, which provides encouraging evidence for effectiveness of *analysis-using-abstract-values* as a modular analysis technique.

We plan to complete the implementation of this technique and test it on bigger programs, especially object-oriented libraries.

References

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994. Also available as DIKU report 94/19.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BCCH94] Michael Burke, Paul Carini, Jong-Doek Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, August 1994.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CGZ95] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioural differences between C and C^{++} programs. *Journal of Programming Languages*, 2:313–351, 1995.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT Press and McGraw-Hill Book Company, 1992.

[CR97] Ramkrishna Chatterjee and Barbara Ryder. Scalable, flow-sensitive type-inference for statically typed object-oriented programming languages. Technical Report DCS-TR-326, Dept of CS, Rutgers University, July 1997.

[CRL97] Ramkrishna Chatterjee, Barbara Ryder, and William Landi. Complexity of concrete type-inference in the presence of exceptions. Technical Report DCS-TR-341, Dept of CS, Rutgers University, September 1997.

[Deu94] A. Deutsch. Interprocedural may alias for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, January 1995.

[DMM96] Amer Diwan, J.Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1996.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 242–256, 1994.

[GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pages 108–124, October 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[JM82] N. D. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. 1982.

[Lan92] W. A. Landi. Undecidability of static analysis. *acm Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[LR91] W.A. Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1991.

[LR92] W.A. Landi and Barbara G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[PC94] J. Plevyak and A. Chien. Precise concrete type inference for object oriented languages. In *Proceeding of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '94)*, pages 324–340, October 1994.

[PR96] Hemant Pande and Barbara G. Ryder. Data-flow-based virtual function resolution. In *LNCS 1145, Proceedings of the Third International Symposium on Static Analysis*, 1996.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, pages 146–161, October 1991.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 13–22, June 1995.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the Sixth International Confernce on Compiler Construction*, pages 136–150, April 1996. Also available as LNCS 1060.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 1–12, 1995.

[ZRL96] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, October 1996.

A Complexity

Here we show that the worst-case complexity of *analysis-using-abstract-values* for programs with only single-level types and without dynamically dispatched calls is $O(n^4)$.

Let the total number of statements in the program be n_1, the sum of the numbers of arguments passed at call sites be n_2, n be $n_1 + n_2$, the maximum number of arguments passed at a call site be A, the total number of user defined variables of reference type be V, the total number of dynamically created objects (identified by their creation sites) be L, the total number of call sites be C and l be a program point in a method M.

The first step, the decomposition of the call graph into strongly connected components, is done in linear time using the algorithm given in [CLR92].

Now consider the complexity of phase I. There could be at most V user defined variables of reference type visible at l. Each of them may point to at most L different objects and V different (abstract) initial values at l. This implies that the solution set at l may contain up to $O(V(L+V))$ points-tos. Suppose l is neither an exit-node nor a call-node. For each points-to reaching l, *apply* takes constant time and a constant amount of work is done along an edge to a successor of l. Since there are $O(n)$ edges, the total amount of work done for such nodes is $O(nV(L+V))$.

Now suppose l is a call-node. A value of a variable at l, may determine up to $A+1$ initial values at the entry of the called method. So at most

$O(V(L+V)(A+1))$ work may be done at l in locating points-tos containing abstract values at the exit-node of the called method (the work done in expanding these points-tos will be counted later). In addition, $O(V(L+V))$ work may be done in passing points-tos representing values of local variables to the return-node of l. Thus, the total work done at call-nodes is $O(CV(L+V)(A+1))$.

Now consider an exit node *ex-node*. Let *r-node* be one of the return nodes corresponding to *ex-node*. For each points-to passed from *ex-node* to *r-node*, an initial value may expand into at most $O(V)$ distinct values at *r-node*. So for each points-to passed along an edge from an exit node to a return node, up to $O(V)$ amount of work may be done (note that a part of this work may be done at a call-node). So the total amount of work along such an edge is $O(V^2(L+V))$ as there could be $O(V(L+V))$ points-tos at an exit-node. There are C such edges. This means that the total amount of work done at exit-nodes is $O(CV^2(V+L))$.

Now each of C, V, A and L is at most $O(n)$; hence the the worst-case complexity of Phase I is $O(n^4)$.

Next consider the complexity of phase II. An initial value at an entry-node may expand into at most L different values. So the maximum size of the solution set at an entry-node is $O(VL)$. Consider a call site. At most V variables may point to an initial value (of a variable at the entry of the method containing the call site) at the call site. So for each instantiation of an initial value at an entry-node, up to $O(V)$ amount of work can be done at a call site. Since there could be at most $O(VL)$ instantiations at an entry-node, total amount of work done at a call site is $O(V^2L)$. In addition, at most $O(VL)$ work may be done at a call site in initializing the worklist for phase II. So the total work done is $O(CV^2L)$. Hence, the worst-case complexity is again $O(n^4)$.

Finally we analyze the complexity of phase III. Suppose l is a non-entry node. A variable may point to up to $O(V)$ initial values at l. Each of these initial values may represent up to $O(L)$ *concrete values*. So we need to take the union of these $O(V)$ subsets of the set of *concrete values*. Moreover we need to do this for each variable. So we have the following problem: Given a set S of size $O(n)$ and $O(n)$ subsets of S, how to find the union of any k of these subsets efficiently. We may do preprocessing to reduce the cost of unions because potentially we may have up to $O(n)$ queries for unions and we need to do the preprocessing only once. The naive algorithm which scans each element of each subset takes $O(k \star n)$ time to satisfy each query. Clever preprocessing may reduce the worst-case cost of answering a query from $O(n^2)$ to $O(n^2/\log n)$. But the lower bound for this problem is an open problem in data structure design. So we use the naive algorithm for getting the worst-case complexity. Using the naive algorithm, we need up to $O(VL)$ time for computing the *concrete values* of a variable. So maximum amount of work done at any such node is $O(V^2L)$. There could be up to $O(n)$ such nodes. So the worst-case complexity of phase III is $O(nV^2L)$, which is $O(n^4)$.

Thus, the complexity of each of the three phases is $O(n^4)$. Hence, the worst-case complexity of the whole algorithm is also $O(n^4)$. In [CR97], we present an example for which the algorithm attains this worst-case bound.

B Algorithm for computing equivalence classes

```
Let x_init be the initial value of a parameter or
global variable x, S be the space of initial values
associated with x_init and y be a point in S. In the
following code, for simplicity, we ignore array fields.
They can be easily accommodated using a simple extension
of the following algorithm.

equivalence_class get_equivalence_class( initial_value y )
{
   equivalence_class retVal;
   initial_value tmp;

   tmp = x_init;

   retVal = (equivalence_class)tmp;
   tmp = get_last_element( y, get_type(tmp) );

   while ( tmp != y ) {

      tmp = get_field_init_value( tmp, y );

      retVal = concat( retVal, ".", get_field( tmp, y), "_init" );

      tmp = get_last_element( y, get_type(tmp) );
   }

   // retVal contains the equivalence class of y
   return retVal;
}

/* The example values of y,t and z in the following three
   functions are taken from the next example program. */

initial_value get_last_element( initial_value y, type t )
{
 it returns the last initial value of type t in y.
 For example, if y is param_init.fld1_init.fld2_init.fld1_init
 and t is A, this function returns param_init.fld1_init.fld2_init.
}

type get_type( initial_value y )
{
 it returns the type of y. For example,
 if y is param_init.fld1_init.fld2_init.fld1_init,
```

```
 it returns B.
}

initial_value get_field_init_value( initial_value z,
                                    initial_value y)
{
 z must be a prefix of y. It returns the initial value of the
 field of z through which y is defined. For example, if
 y is param_init.fld1_init.fld2_init.fld1_init
 and z is param_init.fld1_init.fld2_init,
 it returns param_init.fld1_init.fld2_init.fld1_init.
}

string get_field( initial_value z, initial_value y )
{
 it is similar to the previous function except that it returns the
 name of the field of z through which y is defined instead of
 its initial_value. For example, if y is
 param_init.fld1_init.fld2_init.fld1_init
 and z is param_init.fld1_init.fld2_init, it returns "fld1".
}

string concat( string s1, ..., string sn )
{
  return concatenation of s1, ..., sn;
}
```

The following example illustrates the above algorithm.

```
class A {         class B {
 public B fld1;    public C fld1;
 public B fld2;    public A fld2;
};                };

class C {         class D {    class E {
 public B fld1;   };             public D fld1;
 public D fld2;                };
 public C fld3;
};

proc( A param, E formal )
{
  The space of initial values associated with param_init is
  partitioned into following equivalence classes which are
  represented by their representatives:

  1. [ param_init ] : it contains param_init,
     param_init.fld1_init.fld2_init,
     param_init.fld2_init.fld2_init and so on.
     All elements are of type A.


  2. [ param_init.fld1_init ]: it contains param_init.fld1_init,
```

```
    param_init.fld1_init.fld1_init.fld1_init and so on.
    All elements are of type B.

 3. [ param_init.fld2_init ]: it contains param_init.fld2_init,
    param_init.fld2_init.fld1_init.fld1_init and so on.
    All elements are of type B.

 4. [ param_init.fld1_init.fld1_init ]: it contains
    param_init.fld1_init.fld1_init,
    param_init.fld1_init.fld1_init.fld3_init and so on.
    All elements are of type C.

 5. [ param_init.fld2_init.fld1_init ]: it contains
    param_init.fld2_init.fld1_init,
    param_init.fld2_init.fld1_init.fld3_init and so on.
    All elements are of type C.

 6. [ param_init.fld1_init.fld1_init.fld2_init ]: it contains
    param_init.fld1_init.fld1_init.fld2_init,
    param_init.fld1_init.fld1_init.fld3_init.fld2_init and so on.
    All elements are of type D.

 7. [ param_init.fld2_init.fld1_init.fld2_init ]: it contains
    param_init.fld2_init.fld1_init.fld2_init,
    param_init.fld2_init.fld1_init.fld3_init.fld2_init and so on.
    All elements are of type D.

 Similarly, the space of initial values associated with
 formal_init is partitioned into following equivalence
 classes which are singletons because the space is finite:

 1. [ formal_init ] = { formal_init }.

 2. [ formal_init.fld1_init ] = { formal_init.fld1_init }.
}
```

C Example for *analysis-using-abstract-values* for single-level types

```
class A {};
class test {
 static A p1,p2,p3,p4;
 public static void main( ) {
   /* entry_main */

   n1: p1 = new A;
   n2: p3 = new A;

   n3: proc1();

   n4: proc4();
```

```
 static void proc1( ) {
   /* entry1 */
   n5: p2 = p1;

   if ( _ )
      n6: proc2();

   n7: p4 = p1;

   n8: p1 = p3;
```

```
 } /* exit_main  */
 static void proc2( ) {
  /* entry2 */
  n9: proc3();
  n10: proc1();
 } /* exit2  */
 static void proc4( ) {
  /* entry4 */
  n12: p4 = new A;
 } /* exit4  */
};
```

```
} /* exit1 */
static void proc3( ) {
  /* entry3 */
  n11: p3 = new A;
} /* exit3 */
```

SCCG of the above program is as follows.

```
The set of nodes of SCCG is {N1, N2, N3, N4 }, where
N1 = { main }
N2 = { proc1, proc2 }
N3 = { proc3 }
N4 = { proc4 }
and the set of edges is { (N1,N2), (N2,N3), (N1,N4) }.
```

A topological ordering of the nodes of SCCG is as follows.

```
N1, N2, N3, N4
```

The result of Phase I is as follows.

```
First N4 (i.e. proc4) is analyzed to give:
 Node           Solution
  entry4        { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
  n12           { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
  exit4         { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,object_n12> }
Next N3 (i.e. proc3) is analyzed to give:
 Node           Solution
  entry3        { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
  n11           { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
  exit3         { <p1,p1_init>,<p2,p2_init>,<p3,object_n11>,<p4,p4_init> }
Now proc1 and proc2 are analyzed simultaneously and iteratively to obtain
the following fixed point solution:
 Node           Solution
 entry1         { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n5             { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n6             { <p1,p1_init>,<p2,p1_init>,<p3,p3_init>,<p4,p4_init> }
 n7             { <p1,p1_init>,<p1,object_n11>,
                  <p2,p1_init>,
                  <p3,p3_init>, <p3,object_n11>,
                  <p4,p4_init>, <p4,p1_init>, <p4,object_n11> }
 n8             { <p1,p1_init>,<p1,object_n11>,
                  <p2,p1_init>,
                  <p3,p3_init>,<p3,object_n11>,
```

```
                  <p4,p1_init>,<p4,object_n11> }
 exit1          { <p1,object_n11>, <p1,p3_init>,
                  <p2,p1_init>,
                  <p3,p3_init>,<p3,object_n11>,
                  <p4,p1_init>,<p4,object_n11> }
 entry2         { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n9             { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n10            { <p1,p1_init>,<p2,p2_init>,<p3,object_n11>,<p4,p4_init> }
 exit2          { <p1,object_n11>,
                  <p2,p1_init>,
                  <p3,object_n11>,
                  <p4,p1_init>,<p4,object_n11> }
Note <exit1,p1,p3_init> => <exit2,p1,object_n11> => <n7,p1,object_n11>
=> <exit1,p4,object_n11> => <exit2,p4,object_n11> is inferred iteratively.
Finally main is analyzed to give:
 Node          Solution
 entry_main    { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n1            { <p1,p1_init>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n2            { <p1,object_n1>,<p2,p2_init>,<p3,p3_init>,<p4,p4_init> }
 n3            { <p1,object_n1>,<p2,p2_init>,<p3,object_n2>,<p4,p4_init> }
 n4            { <p1,object_n2>, <p1,object_n11>,
                 <p2,object_n1>, /* since p1_init at n3 is object_n1 */
                 <p3,object_n2>, <p3,object_n11>,
                 <p4,object_n1>, /* since <exit1,p4,p1_init> and
                                      p1_init is object_n1 at n3 */
                 <p4,object_n11> }
 exit_main     { <p1,object_n2>, <p1,object_n11>,
                 <p2,object_n1>,
                 <p3,object_n2>, <p3,object_n11>,
                 <p4,object_n12> /* since <exit4,p4,object_n12> */ }
```

Next, SCCG is traversed in the topological order to propagate initial values.

```
Let p1_in, p2_in, p3_in and p4_in be the initial values of
p1,p2,p3 and p4 respectively at the entry-node of main.
First N1 (i.e. main) is considered, and initial values at proc1
and proc4 are instantiated as follows:
Node           Initial Values
entry1         { <p1_init,{object_n1}>,
                 <p2_init,{p2_in}>,
                 <p3_init,{object_n2}>,
                 <p4_init,{p4_in}> }
entry4         { <p1_init,{object_n2,object_n11}>,
                 <p2_init,{object_n1}>,
                 <p3_init,{object_n2,object_n11}>,
                 <p4_init,{object_n1,object_n11}> }
Next N2 (i.e. proc1 and proc2) is analyzed iteratively to give:
Node           Initial Values
entry1         { <p1_init, {object_n1}>,
                 <p2_init, {p2_in,object_n1}>,
                 <p3_init, {object_n2,object_n11}>,
```

```
                <p4_init, {p4_in}> }
entry2        { <p1_init, {object_n1}>,
                <p2_init, {object_n1}>,
                <p3_init, {object_n2,object_n11}>,
                <p4_init, {p4_in}> }
entry3        { <p1_init, {object_n1}>, /* note this is same as at entry2 */
                <p2_init, {object_n1}>,
                <p3_init, {object_n2,object_n11}>,
                <p4_init, {p4_in}> }
After this N3 (i.e. proc3) is considered. It is a leaf node and
proc3 is the only method in this node, so its solution has
already been computed.
Finally N4 (i.e. proc4) is considered. Its solution also has been
computed. This marks the end of Phase II.
```

Finally, initial values are used to compute solution at non-entry nodes in each method as follows.

```
Solution for main
Node          Solution
entry_main    { <p1,p1_in>, <p2,p2_in>, <p3,p3_in>, <p4,p4_in> }
n1            { <p1,p1_in>, <p2,p2_in>, <p3,p3_in>, <p4,p4_in> }
n2            { <p1,object_n1>, <p2,p2_in>, <p3,p3_in>, <p4,p4_in> }
n3            { <p1,object_n1>, <p2,p2_in>, <p3,object_n2>, <p4,p4_in> }
n4            { <p1,object_n2>,<p1,object_n11>, <p2,object_n1>,
                <p3,object_n2>, <p3,object_n11>, <p4,object_n1>,
                <p4,object_n11> }
exit_main     { <p1,object_n2>,<p1,object_n11>, <p2,object_n1>,
                <p3,object_n2>, <p3,object_n11>, <p4,object_n12>  }
Solution for proc1
Node       Solution
entry1     { <p1,object_n1>, <p2,p2_in>,<p2,object_n1>,<p3,object_n2>,
             <p3,object_n11>, <p4,p4_in>}
n5         { <p1,object_n1>, <p2,p2_in>,<p2,object_n1>,<p3,object_n2>,
             <p3,object_n11>, <p4,p4_in>}
n6         { <p1,object_n1>,  /* since p1_init is {object_n1} */
             <p2,object_n1>,  /* since p1_init is {object_n1} */
             <p3,object_n2>,  /* since p3_init is {object_n2,object_n11} */
             <p3,object_n11>,
             <p4,p4_in>       /* since p4_init is {p4_in} */ }
n7         { <p1,object_n1>,  /* since p1_init is {object_n1} */
             <p1,object_n11>,
             <p2,object_n1>,  /* since p1_init is {object_n1} */
             <p3,object_n2>,  /* since p3_init is {object_n2,object_n11} */
             <p3,object_n11>,
             <p4,p4_in>,      /* since p4_init is {p4_in} */
             <p4,object_n1>,  /* since p1_init is {object_n1} */
             <p4,object_n11> }
n8         { <p1,object_n1>,  /* since p1_init is {object_n1} */
             <p1,object_n11>,
             <p2,object_n1>,
```

```
              <p3,object_n2>, <p3,object_n11>,
              <p4,object_n1>,  /* since p1_init is {object_n1} */
              <p4,object_n11> }
exit1       { <p1,object_n2>, <p1,object_n11>, <p2,object_n1>,
              <p3,object_n2>, <p3,object_n11>, <p4,object_n1>,
              <p4,object_n11> }

Solution for proc2
Node        Solution
entry2      { <p1,object_n1>, <p2,object_n1>,<p3,object_n2>,
              <p3,object_n11>, <p4,p4_in> }
n9          same as at entry2
n10         { <p1,object_n1>, <p2,object_n1>, <p3,object_n11>, <p4,p4_in> }
exit2       { <p1,object_n11>, <p2,object_n1>, <p3,object_n11>,
              <p4,object_n1>, <p4,object_n11> }

Solution for proc3
Node         Solution
entry3       { <p1,object_n1>, <p2,object_n1>,<p3,object_n2>,
               <p3,object_n11>, <p4,p4_in> }
n11          same as at entry3
exit3        { <p1,object_n1>, <p2,object_n1>, <p3,object_n11>, <p4,p4_in> }

Solution for proc4
Node         Solution
entry4       { <p1,object_n2>, <p1,object_n11>, <p2,object_n1>,
               <p3,object_n2>, <p3,object_n11>, <p4,object_n1>,
               <p4,object_n11> }
n12          same as at entry4
exit4        { <p1,object_n2>, <p1,object_n11>, <p2,object_n1>,
               <p3,object_n2>, <p3,object_n11>, <p4,object_n12> }
```

D How queries are answered

In this appendix, we present details of how a query for concrete type-inference is answered using the solutions computed by Phase I, II and III, and the *DemandTable*.

Given a *reference variable* r and a program point l in a method M, the solution computed by Phase I for l is used to compute (by following chains of indirections) the objects (or initial values) to which r points and the associated conditions. Then the *concrete values* of abstract values are used to evaluate these conditions and obtain the values of r.

If the values of an initial value's field not used by M are needed while computing the objects pointed to by r, the initial value is instantiated to obtain the *concrete values* represented by this initial value and then the values of this field of these *concrete values* are used.

Whenever the values of a field of a *concrete value* not used by M (i.e. it is not the *var* field of any data-flow element computed for M by Phase I) are needed at l, the *DemandTable* is accessed to compute its initial values. The values of this field computed by Phase III at l are added to the values obtained from the *DemandTable*. The latter account for the indirect modifications to this field through initial values.

On the other hand, if the values of a field of a *concrete value* used by M are needed at l, the solution computed by Phase III for l is sufficient to find these.

E Example for interprocedural flow-sensitivity

```
class B {
};
class A {
  public:
    static B method1( B param ) {
      return param;
    }

    static void method2( ) {
      B p,q;

      l1: p = new B;
      q = method1( p );

      // solution computed by an interprocedurally flow-sensitive algorithm:
      // ⟨q, object_l1 ⟩. object_l1 represents objects created at program point l1.

      // solution computed by an interprocedurally flow-insensitive algorithm:
      // ⟨q, object_l1 ⟩, ⟨q, object_l2 ⟩

      // Note: ⟨q, object_l2 ⟩results from the unrealizable path:
      // method3 $\xrightarrow{calls}$ method1 $\xrightarrow{returns-to}$ method2
    }

    static void method3( ) {
      B p,q;

      l2: p = new B;
      q = method1( p );

      // solution computed by an interprocedurally flow-sensitive algorithm:
      // ⟨q, object_l2 ⟩

      // solution computed by an interprocedurally flow-insensitive algorithm:
      // ⟨q, object_l1 ⟩, ⟨q, object_l2 ⟩
    }
}
```

Fig. 13. Interprocedural flow-sensitivity or context-sensitivity