Scalable, flow-sensitive type inference for statically typed object-oriented languages

Ramkrishna Chatterjee   Barbara Ryder

Department of Computer Science
Rutgers University, USA
{ramkrish,ryder}@cs.rutgers.edu

DCS-TR-326

August 4, 1997

Abstract

An important problem in the analysis of programs written in object-oriented languages like C^{++}/Java is to determine the values that a pointer variable (a reference to an object in Java) can have at run-time. This information can be used for applications such as virtual function resolution, side-effect analysis, detecting memory errors etc. We address issues involved in designing a **scalable, flow-sensitive solution** for this and related compile-time analysis questions.

First we present a flow-sensitive solution which uses conditional points-to analysis. Although we show that it is better than a solution using alias analysis, it has several limitations which affect its scalability and precision. Next we identify properties necessary for any scalable, flow-sensitive algorithm solving these problems. Using this characterization, we propose two techniques: **preprocessing using types** and **analysis-using-abstract-values**, which remove many of these limitations. We show that these techniques can represent a part of the solution implicitly, and thus facilitate efficient **demand-driven** computation.

We introduce analysis-using-abstract-values by presenting an algorithm for finding the **precise** solution for **may points-to** in the presence of only single-level pointers (pointers with a single level of indirection). We show that this algorithm improves the worst-case bound from $O(n^5)$ to $O(n^4)$. Finally, we present an extension of this technique for multi-level pointers (pointers with multiple levels of indirection), with examples to show improvements in efficiency as well as precision. We also show that this technique can adaptively change flow-sensitivity when it is profitable to do so.

We also show that analysis-using-abstract-values aids in **modular** analysis by allowing the analysis of part of a program and avoiding the need to keep the whole program in memory.

1 INTRODUCTION

An important problem in the analysis of programs written in object-oriented languages like C^{++}/Java is to determine the *concrete types* of objects that a pointer variable (a reference to an object in Java) can point to at run-time, at a given program point. The definition of *concrete type* depends upon the application which uses it. Consider the following program:

```
class A {
    public:
           int data;
           virtual void foo(void);
};

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

func()
{
        A* ptr;

        n1: ptr = new B;
          .
          .
          .
        n2: ptr = new B;

        n3: ptr->foo();

        n4: ptr->data = 1;
}

g()
{
       func();
}

f()
{
       func();
}
```

If we are interested in virtual function resolution [PR96] (i.e., whether a virtual function call can be statically resolved to remove the run-time overhead of dynamic dispatch), it is enough to know the class of an object pointed to by a pointer. For example, if we know that the only possible class of the objects pointed to by *ptr* at *n3* is *B*, we can uniquely resolve the call to *B::foo*. We don't need to know whether the object pointed to was created at *n1* or *n2*. On the other hand, if we are interested in finding the side-effects [LRZ93] of a statement, a different definition of *concrete types* is needed. For example, *n4* modifies the *data* field of the object created at *n2* and not the one created at *n1*. So it is not enough to know the class of an object, we need to know the *actual* object pointed to by a pointer.

The total number of objects that could exist at run-time could be unbounded, so we need to represent them using finite number of equivalence classes. One possibility is to group all objects created at the same program point into an equivalence class. We may further refine these equivalence classes using finite approximations of the call stack (stack of current procedure activations at a program point). For example, we can divide the objects created at *n1* into two disjoint sets: those which are created when *func* is called from *g*, and those which are created when *func* is called from *f*. Similarly we can further refine by using callers of *g* and *f*. Although the precision of the solution obtained by an algorithm is affected by the coarseness of these equivalence classes (i.e., we may get a more precise solution by first using a partition more refined than the one needed by an application and then projecting the solution on to a coarser partition; we would like to further investigate this experimentally), this is generally orthogonal to other approximations in an algorithm. So it is better to consider the definition of equivalence classes to be part of the problem definition. Note that approximating objects using the classes to which they belong, as in virtual function resolution, amounts to grouping all objects of a class into one equivalence class.

For some applications it is not enough to know just the value/*concrete type* of a pointer variable. Additional information about the relationship between the values of different pointer variables is needed (for an example, please see the definition of hold-together in section 3). In section 3.3, we define a set of queries which seem to satisfy the needs of most applications that we are considering at present. We call such a set "**the complete set of queries**". We will expand this set as we gain experience with other applications. We have chosen a subset (defined in section 3.3) of these queries to investigate in this paper. This subset is sufficient for virtual function resolution and side-effect analysis. We hope to handle the rest of these queries in the future.

In the rest of this paper, we investigate issues involved in designing a scalable, flow-sensitive algorithm for the above queries. We begin, in section 2, by pointing out some drawbacks of doing type inference through alias analysis. In section 3, we present a solution using points-to analysis which removes some of the drawbacks of the solution through alias analysis. But this algorithm also has many limitations. In section 4, we identify properties that any scalable, flow-sensitive algorithm for this problem must have. Using this characterization we propose two techniques which help in achieving these properties. Section 5 is devoted to the first of these techniques - preprocessing using types. In sections 7 and 8, we present the second technique - analysis-using-abstract-values. Section 7 covers the single-level algorithm, proof of its precision, and its worst-case complexity - we show that this technique improves the worst-case bound from O(n^5) to O(n^4) for single-level pointers. We also show that this technique aids in **modular** analysis by allowing the analysis of part of a program and avoiding the need to keep the whole program in memory. In section 8, we present some initial ideas about how to do analysis-using-abstract-values in the presence of multi-level pointers and recursive types. We also present some examples which illustrate additional advantages of this technique. Finally, we conclude by summarizing the main contributions of this paper.

2 Drawbacks of type inference through alias analysis

One of the approaches to type inference for C^{++} is to do alias analysis as shown by Pande and Ryder [PR96]. They present a flow-sensitive algorithm which stores the context for interprocedural analysis using a single assumed alias - an alias which holds at the entry of a procedure and approximately represents the context in which the procedure has been called. This approach has not turned out to be scalable because of the following reasons:

1. **Too much information:** Alias analysis is expensive in the presence of recursive structures. But object-oriented programs have an abundance of structures which represent dynamically created objects. Moreover sub-objects are frequently shared (i.e., many fields point to the same object) as shown in the following example:

```
class Scope {           class Variable {
      ...                 public:
};                          Scope* scope;
                            ...
                        };

proc() {
```

```
  Scope* s;
  Variable* v1, v2, .., vn;

  n0: s = new Scope;

  n1: v1 = new Variable;
      v1->scope = s;
  ...

  nn: vn = new Variable;
      vn->scope = s;

  // *(obj_ni.scope) and *(obj_nj.scope)
  // are aliases for all i,j

  // only <*(obj_ni.scope),obj_n0>
  // are needed for type inference
  // or side-effect analysis
}
```

As a result, due to k-limiting [1] (for recursive structures) and shared sub-objects, the number of aliases may be quadratic; although the size of the solution set for type inference may still be linear. Moreover, as shown by Bill Landi in his thesis [Lan92], alias analysis has quadratic error propagation : a single spurious alias can give rise to O(n^2) spurious aliases at a program point.

2. **Loss of precision due to single assumed alias:** Consider the following frequently occurring pattern in object-oriented programs:

```
class base {
    private:
        Object* data;
    public:
        void update( Object* param ) {
                    data = param;
              }
};

Object V;

f1() {
     ...
     l1: p1 = new base;
     s1: p1->update( &V );
     t1:
}

...

fn() {
     ...
     ln: pn = new base;
     sn: pn->update( &V );
```

[1] k-limiting is a standard technique for approximating names in a priori unbounded heap structures [JM79]

```
        tn:
    }
```

Suppose only one assumed-alias is used for representing context. Without loss of generality let this be $\langle\ param, V\ \rangle$ for the aliases $\langle\ obj\_li.data,\ V\ \rangle$, where *obj_li* represents the object created at location *li*. As a result, each $\langle\ obj\_li.data, V\ \rangle$, $1 \leq i \leq n$, will be inferred at each *ti*, although only $\langle obj\_li.data, V\rangle$ may be valid at *ti*. Note that this is a frequently occurring pattern because most operations in a method are with respect to the receiver's fields.

3. **Too much information at each node:** Ideally the number of data-flow facts stored at a node by the analysis algorithm should be proportional to the facts referenced/modified by the node. But to ensure termination of the algorithm, the alias analysis algorithm stores all reaching data-flow facts at a node. This increases the number of facts stored at a node in the worst case from $O(n)$ to $O(n^2)$.
4. **Data-flow facts are not killed effectively:** As a result, we get precision similar to a flow-insensitive algorithm at a cost similar to a flow-sensitive algorithm. We discuss, with an example, this problem further in section 8.6.
5. **No separation of context insensitive information from context sensitive information:** Consider the previous example. Method *update* assigns the value of *param* to the *data* field of the receiver. This information (context insensitive) can be inferred without knowing the actual values (context sensitive) of *param* and the receiver. Effective use of such context insensitive information can increase both precision and efficiency as we shall see later.

3 Basic Algorithm for Type Inference using Points-To

3.1 Motivation for points-to

Type inference, as defined in section 1, is essentially the points-to relation [EGH94]. So it is natural to compute points-to directly instead of obtaining this as a subset of alias solution [PR96] . This approach has the following advantages over alias analysis:

- No k-limiting is needed for recursive data structures.
- Linear (quadratic for alias analysis as shown in section 1) behaviour in the presence of shared structures.
- One spurious points-to may induce at most O(n) spurious points-tos at a program point. In contrast, a single spurious alias may induce up to $O(n^2)$ spurious aliases at a program point as mentioned earlier.

3.2 Hold-together Relation

The simplest approach to computing points-to is to assume, whenever there is a need, that for any set of points-tos which reach a program point, there exists an execution path along which all elements of this set reach the program point. This assumption may sometimes lead to a solution which is worse than the one obtained as a subset of the approximate solution for alias analysis, although there are cases where, even with this assumption, the points-to solution is better than the approximate solution for alias analysis.

To overcome this simplistic assumption, we define the **hold-together** relation.

definition 1 *Let S be a set of points-tos and n be a program point. Hold-together(S,n) is true if and only if there exists a path from the start node to n such that all elements in S reach n along this path.*

Note that alias relation is a special case of hold-together: two pointers p and q are aliases of each other at a program point n if and only if there exists a value v such that hold-together($\{\langle p,v\rangle, \langle q,v\rangle\}$, n).

To remove the above simplistic assumption, we need an efficient scheme for computing hold-together. But computing hold-together for all possible sets is extremely expensive. So we optimistically compute only the minimal information needed to infer *concrete types*; expensive relations like hold-together are computed only on demand as shown later in section 3.4.2.

3.3 Complete Set of Queries

What is the smallest set of queries which can efficiently answer all useful questions about pointers (object references in Java) ? Following is a possible answer to this question, although it is not clear whether this is a complete set (all queries are with respect to a program point):

1. Given a pointer, what does it point to?
2. Given a set of points-tos, do they hold-together?
3. Given an object, which pointers point to it? (for example, this is needed for detecting memory errors like a memory leak, dangling pointers etc.)
4. Given two sets of points-tos S_1 and S_2, does there exist a path along which all the points-tos in S_1 hold, but none of the points-tos in S_2 hold? (this is needed for detecting memory leaks)

As we stated earlier, at present we are interested only in the first three queries. In the following section we introduce our *basic* algorithm which can answer each of these three queries.

3.4 Outline of the Basic Algorithm

Our *basic* algorithm for concrete type inference is an iterative worklist algorithm similar to Landi-Ryder algorithm [LR92] for alias analysis, but instead of aliases, points-tos are computed. A points-to has the form $\langle loc,obj\rangle$; where *loc* and *obj* are one of the following:

- A global variable.
- A local variable.
- An invisible [2]
- An array object [3]
- *obj_n.f* - field *f* of object created at program point n. If some approximation of the call stack is used to classify objects, as mentioned in section 1, then *_n* is replaced by it.
- *obj_n* - object created at program point n. Again, if some approximation of the call stack is used to classify objects, *_n* is replaced by it.

In order to restrict data-flow only to realizable paths [LR92] , points-tos are computed conditioned on assumed points-tos (akin to assumed alias in [LR92] [PR96]), which represent points-tos reaching the entry of a procedure, and approximate the context in which the procedure has been called. We will refer to such points-tos as **conditional points-tos** in the rest of this paper. Each assumed points-to for methods has the form $\langle$*receiver_object*, $\langle$*loc, obj*$\rangle\rangle$, where $\langle$ *loc,obj* $\rangle$ is a points-to which reaches the entry of the method when the receiver of the method is *receiver_object*. This representation removes the loss in precision due the use of a single assumed points-to, shown in section 1.

Next we describe two other important aspects of this algorithm: how objects created at the same program point are distinguished and how hold-together relation is computed.

[2] An invisible is a local variable of a caller, which is not syntactically visible inside a called procedure [LR92].
[3] We represent all the locations in an array using a single location.

3.4.1 Context for Objects

In object-oriented programs the number of sites at which objects are created is relatively small compared to the size of the programs. So it is important for higher precision to distinguish between objects created at the same program point. As mentioned in section 1, one way to do this is to partition the set of objects created at the same program point using a finite approximation of the call stack (stack of procedure activations) existing at the time of object creation. Suppose, we use 1-level calling context to partition the objects. Corresponding to each object creation site C, the algorithm generates one *abstract* object a and one object o_i for each of the call sites cl_i of the procedure p containing C. The algorithm generates a at C. If a propagates to return node r_i corresponding to cl_i, it is replaced by o_i. When the algorithm halts, if a is present in the solution set of a node, it means o_i for each i is in the solution set. If o_i is present then it stands only for itself. More than one level of context can be handled similarly. In this case an object with n-level context represents objects with context n and more.

3.4.2 Computation of hold-together

In order to compute hold-together on demand, we associate a creation ID with each points-to. So a points-to looks like $\langle$ *loc,obj, creation_ID* $\rangle$, where *creation_ID* identifies the program point at which this points-to was created. Now suppose we want to test whether hold-together($\{pt_1, ..., pt_k\}$, n). Let c_1, ..., c_k be the creation IDs of these points-tos. We check if there exists i such that at c_i, each pt_j, $1 \leq j \leq k$, is present in the solution computed at c_i so far. If this is not true then hold-together($\{pt_1, ..., pt_k\}$, n) is false with respect to the solution computed so far. But even if some pt_j is not present in the solution computed at c_i so far, it may be added later. So if no c_i contains all pt_j, we insert each pt_j (if it is not present already) in the solution set of each c_i, with additional information to indicate the test that needs to be done once pt_j reaches c_i. If k is large, alternatively we may store hold-together($\{pt_1, ..., pt_k\}$, n) in a global data structure and perform the test after a fixed point is reached. If the test succeeds and this hold-together implies an increase in the solution computed so far, we restart propagation. We don't want to use this strategy if k is small as there will be large number of queries for such hold-togethers and many of them are likely to be false. In this case, the first strategy is preferable. The second strategy is good only if either the number of hold-togethers to be stored is small or most of them are true.

In an object-oriented program, the members of an object are generally modified in its methods. So the number of creation sites for a points-to involving the field of an object is relatively small, and independent of the program size. The same is true for local variables as methods are generally small in object-oriented programs. Points-tos involving global variables can have many creation sites, but global variables are rare in object-oriented programs. Only points-tos involving parameters could be problematic. The same points-to involving a parameter may be created at many call sites. However, within a method/procedure these can be abstractly represented using a single *creation_ID* which could be expanded only when a test needs to be done.

This basic test may be strengthened in many different ways. For example consider the following:

```
if ( _ )
   u: t = &obj1;
   n: q = &obj2;
else {
   o: q = &obj3;
   m: t = &obj4;
}

l: p = q;

s:  hold-together( { <p,obj2,l> , <t,obj4,m> } ) ?
// Note: for simplicity, we ignore assumed points-tos
// in this example.
```

The basic test succeeds, but in reality the hold-together is false. This could be checked by continuing the basic test for { ⟨ *q,obj2,n* ⟩ , ⟨ *t,obj4,m* ⟩ }.

Note that the *basic* algorithm can answer each of the three queries we are interested in. So in the following sections we focus on how to increase its efficiency rather than answering all the queries.

4 What is a Scalable Algorithm for Points-to ?

An algorithm is scalable only if it shows close to linear (with a small constant) performance in practice. To achieve this, information stored at a program point should be constant on average. Two factors affect this:

1. The number of variables (globals, locals, members of objects) for which information is stored at a program point.
2. The quantity of information stored for each variable at a program point.

The *basic* algorithm potentially stores information for all globals, all locals (visible at the program point) and all members of all objects. Moreover it may store $O(n^2)$ - where n is approximately the total number of globals, locals, objects, and members of objects - information for each variable. This quadratic, worst-case bound is due to the use of assumed points-tos. Section 7.3 contains an example which shows this worst-case behaviour.

In the following sections we describe two techniques which help in overcoming these drawbacks of the *basic* algorithm. The first technique tries to address (1), while the second technique addresses (2) as well as (1). Another drawback of the *basic* algorithm is that it needs to keep information about all program points simultaneously in memory. The second technique solves this problem also.

5 Preprocessing and demand-driven computation

5.1 Goal of Preprocessing

As stated in section 4, a scalable points-to algorithm should store information about as few variables as possible at a program point. However, we need to answer queries about an arbitrary variable at an arbitrary program point. Therefore a scalable algorithm should compute only a part of the solution at a program point **explicitly**; the rest it must represent **implicitly** and expand on **demand**. The goal of preprocessing is to facilitate this. We use a prepass to compute *non-standard types* referenced/modified by each procedure, which are then used for demand-driven computation as shown later.

5.2 Definition of Type

We use the following non-standard definition of *type*:

```
'type' ->   C.f ; where f is a pointer_type field of class/structure C
          | pointer_type
          | Var
          | Array_type

 Var   ->   l ; where l is any local variable or formal
          | g ; where g is a global variable
          | StructVar.f; where f is pointer_type field of
                         class/structure variable StructVar
```

```
StructVar ->   v; where v is a class/structure variable
             | StructVar.f; where f is class/structure field  of
                           StructVar
```

Note that in C^{++} an array variable can be treated as a pointer, but in Java this is not the case. Also in Java pointer_type is a reference type.

The above definitions have been motivated by safety and efficiency of implicit representation and demand-driven computation. This will become clear when we describe demand-driven computation in section 5.5.

5.3 Algorithm for Preprocessing using Types

Preprocessing consists of the following steps:

1. Call graph CG is constructed. All virtual function calls are resolved using hierarchy analysis [DMM96] and by considering only instantiated types as in Bacon and Sweeney's algorithm [BS96]. Calls through function pointers can also be resolved by only considering those functions whose addresses have been stored in a function pointer and whose signatures are compatible with the type of the function pointer through which the call has been made.

2. Directed acyclic graph SCCG of strongly connected components of CG is constructed. This can be done in linear time [CLR92].

3. Types referenced and modified by each procedure/method are computed. For this following definitions are needed :

   - local(P) = local variables (including fields of structure/class variables) of procedure/method P.
   - refType(S) = types directly (excluding procedure calls) referenced by statement S.
   - modType(S) = types directly modified by statement S.
   - type(S) = modType(S) $\cup$ refType(S).
   - localRefType(P) = $\cup_{S \in P}$ refType(S); where P is a procedure/method.
   - localModType(P) = $\cup_{S \in P}$ modType(S).
   - localRefTypeEl(P) = localRefType(P) $-$ local(P).
   - localModTypeEl(P) = localModType(P) $-$ local(P).
   - localType(P) = localRefType(P) $\cup$ localModType(P).
   - localTypeEl(P) = localType(P) $-$ local(P).
   - localRefType(C) = $\cup_{P \in C}$ localRefTypeEl(P); where C is a node of SCCG.
   - localModType(C) = $\cup_{P \in C}$ localModTypeEl(P); where C is a node of SCCG.

   Following example illustrates the above definitions:

```
class A {
   public:
      int *fieldA;
};

class B {
   public:
      int *fieldB;
};
```

```
A* p; /* global variables */
B* q;

S: p->fieldA = q->fieldB;

refType(S) = { p,q,B.fieldB }
modType(S) = { A.fieldA }
```

Types modified and referenced by each node of SCCG are given by the following equations:

$$refType(C) = localRefType(C) \cup_{(C,C_i)\in SCCG} refType(C_i) \quad (1)$$

$$modType(C) = localModType(C) \cup_{(C,C_i)\in SCCG} modType(C_i) \quad (2)$$

Equations (1) and (2) are solved by postorder traversal of SCCG. The types modified and referenced by a procedure P are then given by:

refTypeEl(P) = refType(C); where P is in C
modTypeEl(P) = modType(C)
typeEl(P) = refTypeEl(P) $\cup$ modTypeEl(P)
type(P) = localType(P) $\cup$ typeEl(P)

4. For each node in SCCG, the set of nodes reachable from it is computed by postorder traversal of SCCG. This computation may be combined with the previous step.

5. The final step is **intraprocedural separation** using def-use associations for non-standard types. Before describing this step, we want to motivate this step using the following example.

```
class A{
};

class A1: public A{
};

...

class An: public A{
};

A* alloc( int flag)
{
    switch( flag ) {
       s1: case 1 : return  new A1;

       ...

       sn: case n : return  new An;
    }
}

void proc( void )
```

```
{
   int flag;
   A*  p;
   A*  q;

   flag = read_flag();

   l0: p = alloc( flag );

   /* statements l1 through ln do not refer to p */

   l1: ...

   ...

   ln: ...

   s: q = p;
}
```

The *basic* algorithm stores each $\langle p, obj\_si \rangle$, $1 \leq i \leq n$, at each of the program points *li*. This results in a quadratic solution. But using def-use associations for types (recall p is a type), it is found that the definition of type p created at *l0* reaches s, and none of the intervening statements refer to it. So the n points-tos created at *l0* during points-to analysis can be directly forwarded to s, without going through any *li*. This results in a linear size solution. Hence, the main advantage of intraprocedural separation is that it can figure out the path of n points-tos using the path of a single definition of a type which abstractly represents all these points-tos. Thus, it cuts down on the amount of information that must be stored at each program point.

Now we will describe the algorithm for this step. Each statement S in a procedure P is considered to define types in modType(S) and use types in refType(S). Types to be considered are those in localType(P). Entry node is assumed to define all types in localTypeEl(P). A call node is assumed to define all types in localTypeEl(P) $\cap$ modTypeEl(Pc), where Pc is the called procedure, and it is assumed to refer to all types in localTypeEl(P) $\cap$ refTypeEl(Pc). In addition each call node and the entry node is assumed to define a **special type** called *extType* which stands for types in typeEl(P) $-$ localTypeEl(P). The exit node is assumed to use all types in localTypesEl(P) and the *extType*. Using these assumptions, intraprocedural equations for def-use associations [ASU86] for types are solved using an iterative worklist algorithm, where each statement (including call nodes) kills previous definitions of types which it defines.

5.4 Complexity of preprocessing

In this subsection we analyze the complexity of the algorithm for preprocessing. Note that our arguments are rough and approximate, and our goal is to identify the characteristics of a program, which determine the complexity of this algorithm.

Let

- Average size of a procedure/method = $\bar{S}$.
- Average number of nodes reachable from a node in CG = $\bar{R}$.
- Number of nodes reachable from a node m in CG = Rm.
- Average number of nodes reachable from a node in SCCG = $\bar{C}$.

- The number of nodes reachable from a node C in SCCG = Cn.
- Total number of methods/procedures = M.
- Total number of edges in SCCG = Ec.
- Total number of edges in CG = E.
- Average number of fields per type (considering inherited fields) = $\bar{FT}$.
- Average number of methods per type (without considering inherited methods) = $\bar{MT}$.

$\Rightarrow$ Total number of fields = $\bar{FT} \star$ total number of types = $\bar{FT} \star (M/\bar{MT})$.

$\Rightarrow$ Maximum number of types ref/mod by a procedure/method = $\bar{FT} \star (M/\bar{MT})$. Note that for simplicity we ignore globals and locals as their number is small compared to the total number of fields in an object-oriented program.

So this implies that the worst case cost of step 3 is = Ec $\star$ $\bar{FT} \star (M/\bar{MT})$. But Ec is O(E). As a result, the worst-case cost of preprocessing is quadratic in the size of CG.

But the number of types ref/mod by a method $m \leq$ total number of statements reachable from it $\sim$ Rm $\star$ $\bar{S}$. Note that although this approximation is valid for most programs, it does not hold in general. We can easily construct examples (one large procedure reachable from many small procedures) for which this is an underestimate.

$\Rightarrow$ cost of step 3 is $\sim \bar{R} \star \bar{S} \star E$, again assuming that the number of nodes, in CG, whose in-degree is greater than the average in-degree of a node is small.

Now cost of preprocessing is $\sim$ cost of step 3 + cost of step 4 + cost of step 5; where cost of step 4 is $\sim \bar{C} \star Ec$ (assuming that the number of nodes in SCCG for which Cn is greater than $\bar{C}$ is small) and the cost of step 5 is O(M) assuming that the maximum size of a method is bounded. But $\bar{S}$ ($\sim$ 10 to 20) is typically small in object-oriented programs, and $\bar{FT}$, $\bar{R}$ and $\bar{C}$ also appear to be bounded and relatively independent of the program size, although we need to verify this experimentally. For example, in a balanced tree $\bar{R}$ is same as the depth of the tree, which is O($\log n$) if n is the number of nodes in the tree. Hence with these assumptions, the cost of preprocessing is **approximately** linearly related to the size of CG.

5.5 Demand driven computation using ref/mod types

The preprocessing phase is followed by iterative computation of points-to. During this phase *ref/mod* types computed by preprocessing help in representing a part of the solution implicitly [4], which can be expanded on demand. Suppose $\langle$*assumed-points-to*,$\langle$*loc*,*obj* $\rangle\rangle$is a points-to which reaches a call node, *Cnode*, which calls a procedure/method *P*. It is forwarded to the entry node of *P* if and only if the type of *loc* belongs to typeEl(*P*). Otherwise $\langle\langle$*loc*,*obj* $\rangle$, *Cc* $\rangle$, where *Cc* is the node of SCCG containing *P*, is stored in a **global table**. Suppose a query about a location *loc* needs to be answered with respect to a program point *n* in a procedure *Q* and the type of *loc* is not present in type(*Q*). Then the global table is searched to check if there exists an entry for *loc*. Suppose such an entry *e* exists and the node of SCCG with respect to which this was created is *Ci*. Let *C* be the node of SCCG containing *Q*. If *C* is reachable from *Ci* [5] then the value of *loc* in *e* is a value of *loc* at *n*, so *e* is used to answer the query. If no such entry exists then *loc* does not reach *n*.

5.5.1 Subtype Relation

Because of indirect modifications and references through pointers, a variable *loc* of one type can be modified indirectly through pointers to a supertype. Thus we need to refine our use of preprocessing information in

[4] Note that applications like virtual function resolution and side-effect analysis do not need the implicitly represented part of the solution, hence this is an optimization for them.

[5] In the presence of virtual function calls and calls through function pointers, the "reachability" in the call graph may be refined during postprocessing using the information computed during the analysis phase; we plan to investigate this in future.

the following manner. **The value of *loc* needs to be forwarded to the entry node of a procedure *P* if and only if the type of *loc* or a supertype of the type of *loc* belongs to typeEl(*P*).** For this, we use an extension of the usual subtype relation to take care of variables and arrays. Note that since the first member (*loc*) of a points-to ($\langle loc,\ obj \rangle$) must be a local variable, an invisible, a global variable, a field of an object or an array object, the following subtype relations are **sufficient for safe** demand-driven computation using preprocessing information.

t1 is a subtype of *t2* if and only if

1. *t1* is *B.f*, *t2* is *A.f*, and *B* is a subtype of *A*. For example:

```
class A {
    public:
       int* f;
};

class B: public A {
};

proc1()
{
    A* p;
    ....
    p->f = new int;
}

proc2()
{
   B* p;
   ....
   p->f = new int;

}

proc3()
{
    A* p;
    B* q;

    n1: p = new A;
    n2: p->f = new int;

    n3: q = new B;
    n4: q->f = new int;

    if (_) {
      /* <obj_n1.f,obj_n2> and <obj_n3.f,obj_n4> need to be passed */
      proc1();
    }
    else {
      /* only <obj_n3.f,obj_n4> needs to be passed as
         A is not a subtype of B */
      proc2();
    }
```

```
}
```

2. *t1* is a local/global variable (including fields of structure/class variables) whose address has been taken and *t2* is the declared type of *t1*. For example:

```
int** q;

proc1(int **s)
{
   /* int* belongs to modTypeEl(proc1) */
   *s = new int;
}

proc2()
{
   int* p;
   int* r;

   n1: p = new int;

   n2: r = new int;

   q = &r;
   /* r becomes a subtype of int* */

   /* <p, obj_n1> need not be passed in because
      p is not a subtype of int* as its address has
      not been taken */
   /* <r, obj_n2> needs to be passed in as r is a
      subtype of int* */
   proc1(q);
}
```

3. *t1* is *A.f*, the address of the field *f* of an object of type *A* has been taken, and *t2* is the declared type of *f*. For example:

```
int** q;

struct A{
 int* f;
};

proc1(int **s)
{
   /* int* belongs to modTypeEl(proc1) */
   *s = new int;
}

proc2()
```

```
{
   struct A *p;

   n1: p = new struct A;

   n2: p->f = new int;

   q = &(p->f);
   /* A.f becomes a subtype of int* */

   /* <obj_n1.f, obj_n2> needs to be passed in as A.f is a
      subtype of int* */
   proc1(q);
}
```

4. This case is only for Java. *t1* and *t2* are array types, and the type of an element of *t1* is a subtype of the type of an element of *t2*.

5. This case is only for C^{++}. We represent an array by a single object (for both C^{++} and Java). If the address of any subarray is taken through an array variable then the object representing the bigger array becomes a subtype of the type of the subarray. For example:

```
proc0()
{
   int* (*p)[6];

  (*p)[1] = new int;
  /* int* [6] belongs to modTypeEl(proc0) */

}

proc1()
{
   int*  (*p)[5];

   n0: int* a[5][5];
   /* entire array is represented by a single object obj_n0 */

   (*p)[1] = new int;
   /* int* [5] belongs to modTypeEl(proc1) */

   a[2][3] = new int;
   /* only obj_n0 belongs to modType(proc1) and
      not int* [5][5] */
}

proc2()
{
   n1: int* a[5][5]; /* this array is represented by obj_n1 */
       int* (*p)[5];

   n2: a[2][3] = new int;
```

```
   /*  <obj_n1,obj_n2> */

   p = a[2];
   /* obj_n1 becomes subtype of int* [5] */
   /* But int* [5][5] does not become subtype of int* [5] */

   if (_) {
      /* <obj_n1, obj_n2> needs to be passed in
         because obj_n1 is a subtype of int* [5] and
         int* [5] belongs to modTypeEl(proc1) */
      proc1();
   }
   else {
      /* <obj_n1, obj_n2> need not be passed in because
         obj_n1 is not a subtype of int* [6] */
      proc0();
   }
}

proc3()
{
   n3: int* a[5];
       int* *p;

       p = a;
       /* obj_n3 becomes a subtype of int* */
}
```

If address of a subarray is taken through a non-array variable then the bigger array type becomes a subtype of the subarray type. For example:

```
proc()
{
   n1: int* a[5][5][5];
       int* (*p)[5][5];
       int* (*q)[5];
       int* *o;

       p = a[2];
       /* obj_n1 becomes subtype of int* [5][5] */

       q = (*p)[1];
       /* int* [5][5] becomes subtype of int* [5] */

       o = *q;
       /* int* [5] becomes a subtype of int* */
       /* transitively obj_n1 is a subtype of int* */
}
```

Note that if f is an array field of a structure/class of type A, then $A.f$ represents the array object for preprocessing. If its address is taken then the subtype relation is determined by a combination of the cases for structure fields and arrays.

5.5.2 Effect of Intraprocedural Separation

Due to intraprocedural separation, described above, values of only those locations whose types are in type(S) are forwarded to a statement S. A statement S forwards the value of a location *loc* in modType(S) directly to all the statements Si such that the type of *loc* is in type(Si) and the definition created by S has been found to reach Si during intraprocedural separation. The entry node of a procedure/method P and the return nodes (recall each call node has an associated return node) contained in P, forward the value of a location whose type is not in localType(P) to a call node contained in P or the exit node of P, if the definitions of *extType* created by the entry node or the return nodes have been found to reach this call node or the exit node during intraprocedural separation. A query about the value of a *loc* with respect to a statement S in a procedure/method P is satisfied as follows:

1. Let the type of *loc* be t. If $t \in$ type(S), information stored at S is used to answer the query (i.e., t is explicitly used in S).
2. If t is not in type(S) but $t \in$ localType(P), the query is answered by accessing information stored at all points whose definitions of t reach S (i.e., t is not explicitly used in S but a ref/mod of t can reach S from within P).
3. If t is not in localType(P) but $t \in$ type(P), the query is answered by accessing information stored at all points whose definitions of extType reach S (i.e., $t \in refType(C)$ or $t \in modType(C)$, where C is the SCC to which P belongs, and a ref/mod of t can reach S from outside P).
4. If t is not in type(P), information in the **global table**, described in section 5.5, is accessed (i.e., $t \notin refType(C)$ and $t \notin modType(C)$, where C is the SCC to which P belongs, and a ref/mod of t can reach S from outside P).

5.6 Type Cast

A type cast can change the subtype relation. For example:

```
class A{
};

class B: public A {
};

A* p;
B* q;

q = (B*)p;
/* A becomes a subtype of B */
```

But such a cast makes sense only if p points to an object of type B. So we ignore such a cast by assuming p points to an object whose type is a subtype of B. If this assumption is incorrect then the algorithm will detect this at the first incorrect type cast: whose value of right hand side does not depend on any other incorrect cast. If the algorithm detects an incorrect type cast, it stops. In future we want to extend the algorithm so that it can detect all incorrect type casts.

6 Analysis-Using-Abstract-Values

Now we present the second technique which tries to reduce the amount of information stored for a variable at a node, and hence tries to remove the second weakness of the basic algorithm. First consider a motivating example.

```
int* p;
int* q;
int v1;
...
int vn;

proc0( )
{
      n0: p = q;

      exit_proc0:
}

proc1( )
{
          q = &v1;
     c1: proc0();
     n1:
}

....

procn( )
{
          q = &vn;
     cn: proc0();
     nn:
}
```

p may point to any vi, $i \in \{1, ..., n\}$, at *exit_proc0*. The basic algorithm explicitly stores at *exit_proc0* each of these points-tos conditioned on the value of q. As a result, each of these O(n) conditional points-tos has to be passed to return node of each ci. This implies quadratic behaviour. Now suppose instead of storing these conditional points-tos explicitly, we store only $\langle p,\ q\_\mathrm{init}\ \rangle$ at *exit_proc0*. This means p points to whatever q pointed to at the entry of *proc0*, and abstractly represents all the above conditional points-tos. This single points-to is then passed to the return node of each ci where it is instantiated with the value of q before the call, reducing the complexity from $O(n^2)$ to O(n) for the above example. This is the essence of the second technique - **analysis-using-abstract-values**.

In the following sections, we first present an algorithm for points-to analysis using abstract values in the presence of only single-level [LR91] pointers and then show how to extend it for multi-level pointers. The algorithm for single-level pointers is precise, whereas the one for multi-level pointers is approximate but safe.

7 Points-To Analysis using Abstract Values in the presence of Single-level Pointers

7.1 Algorithm

The algorithm consists of the following steps:

1. DAG, SCCG, of strongly connected components of the call graph CG is computed.

2. **Phase I:** Nodes of SCCG are traversed in a reverse topological order (bottom-up) and for each node the following iterative algorithm is executed:
   - Each procedure is analyzed assuming that the parameters and global variables have some unknown initial (abstract) values.
   - When a points-to involving a global variable is passed from the exit node of a procedure to a return node corresponding to a call node for this procedure, 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 procedure represent the effect of the procedure on pointer variables in terms of the initial values of global variables and parameters. These represent the **transfer function** of the procedure for pointer variables. Since there is a cyclic dependence between the transfer functions of procedures 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 procedures belonging to different nodes of SCCG have a hierarchical dependence, so they can be solved by a post-order traversal of SCCG.
3. **Phase II:** Nodes of SCCG are traversed in a topological order (top-down) to propagate initial values for global variables and parameters to entry nodes. For procedures belonging to the same node of SCCG, values are propagated iteratively until a fixed point is reached.
4. **Phase III:** At each non-entry node, the value of a pointer variable is computed by replacing the initial values of global variables and parameters by sets of their values computed in the previous step. Note that this step needs to be done at a node only if we are interested in the solution at this node. For example, for virtual function resolution this step needs to be done only at those nodes which have virtual function calls. Hence, this step can be done on **demand**.

Note that each node of SCCG - and hence each procedure - 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 procedure (which represents its transfer function during bottom-up traversal) or the solution at the entry node of a procedure (during top-down traversal) needs to be in memory. Hence this is a **modular** approach and saves memory over whole-program analysis techniques. 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. We want to further investigate the complexity of finding the best possible topological order for steps 2 and 3 of the algorithm; if this task is NP-hard, we want to find an efficient heuristic.

Now we illustrate the above algorithm by applying it to the following example program.

```
int* p1,p2,p3,p4;
int i1,i2,i3,i4;

main()
{ /* entry_main */

   n1: p1 = &i1;
   n2: p3 = &i3;

   n3: proc1();

   n4: proc4();

} /* exit_main  */
```

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

  if ( _ )
     n6: proc2();

  n7: p4 = p1;

  n8: p1 = p3;
} /* exit1  */

proc2()
{ /* entry2 */
  n9: proc3();
  n10: proc1();
} /* exit2  */

proc3()
{ /* entry3 */
  n11: p3 = &i3;
} /* exit3  */

proc4()
{ /* entry4 */
  n12: p4 = &i4;
} /* exit4  */
```

SCCG for the above program is as follows.

```
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
```

Result of step 2 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,i4> }

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,i3>,<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,i3>,
                  <p2,p1_init>,
                  <p3,p3_init>, <p3,i3>,
                  <p4,p4_init>, <p4,p1_init>, <p4,i3> }

 n8             { <p1,p1_init>,<p1,i3>,
                  <p2,p1_init>,
                  <p3,p3_init>,<p3,i3>,
                  <p4,p1_init>,<p4,i3> }

 exit1          { <p1,i3>, <p1,p3_init>,
                  <p2,p1_init>,
                  <p3,p3_init>,<p3,i3>,
                  <p4,p1_init>,<p4,i3> }

 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,i3>,<p4,p4_init> }

 exit2          { <p1,i3>,
                  <p2,p1_init>,
                  <p3,i3>,
                  <p4,p1_init>,<p4,i3> }

Note <exit1,p1,p3_init> => <exit2,p1,i3> => <n7,p1,i3>
=> {<exit1,p4,i3>,<exit1,p1,i3>} => <exit2,p4,i3> 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,i1>, <p2,p2_init>, <p3,p3_init>, <p4,p4_init> }
 n3             { <p1,i1>, <p2,p2_init>, <p3,i3>, <p4,p4_init> }
```

```
 n4               { <p1,i3>,
                    <p2,i1>, /* since p1_init at n3 is i1 */
                    <p3,i3>,
                    <p4,i1>, /* since <exit1,p4,p1_init> and p1_init is
                                i1 at n3 */
                    <p4,i3> }

 exit_main        { <p1,i3>,
                    <p2,i1>,
                    <p3,i3>,
                    <p4,i4> /* since <exit4,p4,i4> */ }
```

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 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,{i1}>,
                <p2_init,{p2_in}>,
                <p3_init,{i3}>,
                <p4_init,{p4_in}> }

entry4        { <p1_init,{i3}>,
                <p2_init,{i1}>,
                <p3_init,{i3}>,
                <p4_init,{i1,i3}> }

Next N2 (i.e. proc1 and proc2) is analyzed iteratively to give:

Node          Initial Values
entry1        { <p1_init, {i1}>,
                <p2_init, {p2_in,i1}>,
                <p3_init, {i3}>,
                <p4_init, {p4_in}> }

entry2        { <p1_init, {i1}>,
                <p2_init, {i1}>,
                <p3_init, {i3}>,
                <p4_init, {p4_in}> }

entry3        { <p1_init, {i1}>, /* note this is same as at entry2 */
                <p2_init, {i1}>,
                <p3_init, {i3}>,
                <p4_init, {p4_in}> }

After this N3 (i.e. proc3) is considered. But it is a leaf node and
proc3 is the only procedure in this node, so its solution has
already been computed.
```

Finally N4 (i.e. proc4) is considered. But its solution also has been computed. This marks the end of top-down propagation phase.

Finally initial values are used to compute solution at non-entry nodes in each procedure 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,i1>, <p2,p2_in>, <p3,p3_in>, <p4,p4_in> }
n3                { <p1,i1>, <p2,p2_in>, <p3,i3>, <p4,p4_in> }
n4                { <p1,i3>, <p2,i1>, <p3,i3>, <p4,i1>, <p4,i3> }
exit_main         { <p1,i3>, <p2,i1>, <p3,i3>, <p4,i4> }

Solution for proc1

Node             Solution

entry1           { <p1,i1>, <p2,p2_in>,<p2,i1>,<p3,i3>,<p4,p4_in>}

n5               { <p1,i1>, <p2,p2_in>,<p2,i1>,<p3,i3>,<p4,p4_in>}

n6               { <p1,i1>,     /* since p1_init is {i1} */
                   <p2,i1>,     /* since p1_init is {i1} */
                   <p3,i3>,     /* since p3_init is {i3} */
                   <p4,p4_in>  /* since p4_init is {p4_in} */ }

n7               { <p1,i1>,   /* since p1_init is {i1} */
                   <p1,i3>,
                   <p2,i1>,   /* since p1_init is {i1} */
                   <p3,i3>,   /* since p3_init is {i3} */
                   <p4,p4_in>, /* since p4_init is {p4_in} */
                   <p4,i1>,    /* since p1_init is {i1} */
                   <p4,i3> }

n8               { <p1,i1>,  /* since p1_init is {i1} */
                   <p1,i3>,
                   <p2,i1>,
                   <p3,i3>,
                   <p4,i1>,  /* since p1_init is {i1} */
                   <p4,i3> }

exit1            { <p1,i3>, <p2,i1>, <p3,i3>,
                   <p4,i1>,<p4,i3> }


Solution for proc2

Node             Solution

entry2           { <p1,i1>, <p2,i1>,<p3,i3>,<p4,p4_in> }
```

```
n9                { <p1,i1>, <p2,i1>,<p3,i3>,<p4,p4_in> }

n10               { <p1,i1>, <p2,i1>, <p3,i3>, <p4,p4_in> }

exit2             { <p1,i3>, <p2,i1>, <p3,i3>, <p4,i1>, <p4,i3> }


Solution for proc3

Node              Solution

entry3            { <p1,i1>, <p2,i1>, <p3,i3>, <p4,p4_in> }

n11               { <p1,i1>, <p2,i1>, <p3,i3>, <p4,p4_in> }

exit3             { <p1,i1>, <p2,i1>, <p3,i3>, <p4,p4_in> }


Solution for proc4

Node              Solution

entry4            { <p1,i3>, <p2,i1>, <p3,i3>, <p4,i1>, <p4,i3> }

n12               { <p1,i3>, <p2,i1>, <p3,i3>, <p4,i1>, <p4,i3> }

exit4             { <p1,i3>, <p2,i1>, <p3,i3>, <p4,i4> }
```

7.2 Proof of Precision

In this section we present an outline of a proof of the **precision** of the single-level algorithm described in section 7. 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.

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 procedure P. Further, let v be a global variable (for simplicity we consider only global variables in this proof), 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 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. Similarly, using induction on the length of a balanced path

[6] i.e., along which each entry node has a matching exit node except possibly for the initial entry node of P; this corresponds to the notion of balanced path in [RHS95]

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 procedure, x be a global variable, and *val_conc* be a concrete value. If there exists an execution path starting from the start 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.

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 procedure 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 copy 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 this 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 (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.

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 starting from 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, it means 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.

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

7.3 Complexity

First we show that the worst-case complexity of conditional may points-to (i.e., queries of type 1 as described in section 3.3) ,using a single assumed points-to, in the presence of only single-level pointers is $O(n^5)$. Let the total number statements be n. This means that there could be up to $O(n)$ variables used in the program. Now consider a program point l. There could be up to $O(n)$ pointer variables visible at l. Each of them may point to $O(n)$ different locations. Each such points-to may be implied by up to $O(n)$ points-tos at the entry of the procedure containing l. This implies that the solution set at l may contain up to $O(n^3)$ points-tos. Suppose l is not an exit node. For each conditional points-to reaching l, a constant amount of work is done along an edge to a successor of l. Since there are $O(n)$ edges, total amount of work done for non-exit nodes is $O(n^4)$. Now consider an exit node *ex-node*. Let *r-node* be one of the return nodes corresponding to *ex-node*. For each conditional points-to passed from *ex-node* to *r-node*, the assumed points-to may expand into $O(n)$ assumed points-tos at *r-node*. So for each conditional points-to passed along an edge from an exit node to a return node, up to $O(n)$ amount of work may be done. So total amount of work along such an edge may be $O(n^4)$ as there could be $O(n^3)$ conditional points-tos at an exit node. But there could be up to $O(n)$ such edges. This means that the worst-case complexity is $O(n^5)$.

The above argument shows that the worst-case complexity of the algorithm can be no worse than $O(n^5)$. To prove that this bound is "tight", we need an example for which the algorithm has this worst-case complexity. The following is such an example.

```
int i1,i2,i3, ..., in;

int* p1 = NULL;
...
int* pn = NULL;


int* q1 = NULL;
...
int* qn = NULL;


int* r1 = NULL;
...
int* rn = NULL;


p()
{
   p1 = &i1;

   if( _ )
       p1 = &i2;

   ...

   if( _ )
       p1  = &in;

   p2 = p1;

   ....

   pn = p1;

   /* each <pj,ik>, 1 <= k,j <= n, holds at this point */
   q();
}


q()
```

```
{
   q1 = p1;

   if( _ )
       q1 = p2;

   ...

   if( _ )
       q1  = pn;

   /* <<pm,ij><q1,ij>> 1 <= j,m <= n */
   /* O(n*n) points-tos */

   q2 = q1;

   ...

   qn = q1;

   /* <<pm,ij><qk,ij>> 1 <= j,m,k <= n */
   /* O(n*n*n) points-tos */

       r();
   /* <<qt,ij>,<rk,ij>> expands to O(n) points-tos:
      <<pm,ij>,<rk,ij>> 1 <= m <= n */
   /* <<pm,ij><rk,ij>> 1 <= j,m,k <= n */
   /* O(n*n*n) points-tos */
   l1: s();

   ...

       r();
   /* <<pm,ij><rk,ij>> 1 <= j,m,k <= n */
   /* O(n*n*n) points-tos */
   ln: s();
}

s()
{
   r1 = NULL;
   ...
   rn = NULL;
}


r()
{
   r1 = q1;

   if( _ )
       r1 = q2;

   ...
```

```
    if( _ )
        r1  = qn;

    /* <<qm,ij><r1,ij>> 1 <= j,m <= n */
    /* O(n*n) points-tos */


    r2 = r1;

    ....

    rn = r1;

    /* <<qm,ij><rk,ij>> 1 <= j,m,k <= n */
    /* O(n*n*n) points-tos */
}
```

Now we show that the worst-case complexity of *analysis-using-abstract-values* for may points-to in the presence of only single-level pointers is $O(n^4)$. This algorithm has three distinct phases:

- bottom-up analysis using abstract initial values.
- top-down propagation of initial values.
- substitution of abstract values in the solution at a non-entry node by the initial values computed in the previous step.

First consider the complexity of phase one. Consider a program point l. There could be up to $O(n)$ pointer variables visible at l. Each of them may point to $O(n)$ different locations (including initial values at the entry of the procedure containing l). This is same as in the case of conditional points-to. But now there is no conditional part. This means solution set at l may contain up to $O(n^2)$ points-tos. Suppose l is not an exit node or an entry node. For each points-to reaching l, a constant amount of work is done along an edge to a successor of l. Since there are $O(n)$ edges, total amount of work done for such nodes is $O(n^3)$. 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 $O(n)$ distinct values at *r-node*. So for each points-to passed along an edge from an exit node to a return node, up to $O(n)$ amount of work may be done. So the total amount of work along such an edge may be $O(n^3)$ as there could be $O(n^2)$ points-tos at an exit node. But there could be up to $O(n)$ such edges. This means that the worst-case complexity is $O(n^4)$.

Now consider the complexity of phase two. An initial value at an entry node may expand into at most $O(n)$ different values. So the maximum size of the solution set at an entry node is $O(n^2)$. Consider a call site. Up to $O(n)$ variables may point to an initial value at the entry of the procedure containing the call site. So for each instantiation of an initial value at an entry node, up to $O(n)$ amount of work has to be done at a call site. Since there could be up to $O(n^2)$ instantiations at an entry node, total amount of work done at a call site is $O(n^3)$. But there could be up to $O(n)$ call sites. Hence, the worst-case complexity is $O(n^4)$.

Finally we analyze the complexity of phase three. Consider a non-entry program point l in a procedure P. A variable may point to up to $O(n)$ initial values at l. Each of these initial values may represent up to $O(n)$ actual values. So we need to take the union of these $O(n)$ subsets of the set of actual values. Note that the set of actual values is of size $O(n)$. 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. Note that 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 $O(n^2)$ time for computing the actual values of a variable. There could be $O(n)$ variables. So maximum amount of work done at any such node is $O(n^3)$. There could be up to $O(n)$ such nodes. So the worst-case complexity is $O(n^4)$.

So the complexity of each of the three phases is $O(n^4)$. Hence, the worst-case complexity for the whole algorithm is also $O(n^4)$. Note that the worst-case example for conditional points-to also works for this algorithm.

Answering other Queries So far in this section we have concentrated on queries of type 1 because we wanted to describe the essential features of *analysis-using-abstract-values* by comparing it with conditional points-to and both are precise for this type of queries, making comparison easier. Queries of type 2 can also be precisely answered by making an additional pass through all the nodes. For each object at each node, we need to store all the variables that point to it. Since there are at most $O(n)$ objects and $O(n)$ variables, we need to do at most $O(n^2)$ amount of work at each node. So the whole pass will cost at most $O(n^3)$ as there are n nodes. Queries of type 3 can be answered by using creation IDs as described in section 3.4.2; however, for these the answer will be approximate, but safe, as is the case with conditional points-to.

8 Points-To Analysis using Abstract Values in the Presence of Multi-level Pointers

In this section, we discuss some initial ideas about how to do *analysis-using-abstract-values* in the presence of multi-level pointers and recursive types. We also present some examples to show the advantages of this technique. We will describe the details of this algorithm in a separate TR.

8.1 Why we need to extend the single-level algorithm for multi-level pointers

Consider the following example.

```
class A {
  public:
    C* member1;
    C* member2;
};

void copy( A* param1, A* param2 )
{
   l1: param1->member1 = param2->member2 ;
}
```

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*. But *param1_init* could be same as *param2_init*. **So this statement can potentially change the value of *member1* of *param2_init* in addition to *member1* of *param1_init*.** This problem is unique to multi-level pointers; so we need to extend the single-level algorithm to take care of such situations.

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 )
entry_copy:
/* <param1,param1_init>,
   <param1_init.member1,param1_init.member1_init>,
   <param2,param2_init>,
   <param2_init.member1, param2_init.member1_init>
*/
{
   l1: param1->member1 = param2->member2 ;

   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>
    */
}
```

This seems to take us back to conditional analysis. But in this case, the conditions are inferred in a bottom-up manner rather than being propagated in a top-down manner. As a result, only those conditions which are relevant are inferred, instead of all possible conditions which may exist at the entry of *copy*. For example, relationship between *param1_init.member1_init* and *param1_init.member2_init* is not relevant, although they may be same at a call site for *copy*. Hence, the number of such conditions in the worst case is proportional to the number of statements reachable from *copy* (which is small on average) rather than the size of the whole program, as in the case of the *basic* algorithm. This makes this approach more scalable than the *basic* algorithm. Moreover, this bottom-up inference facilitates **modular** analysis as shown earlier. Note that if more than one condition is associated with a points-to, all of them are stored with the points-to. This is in contrast to the Landi-Ryder algorithm [LR92] which stores only one such condition (without affecting the safety of the solution).

8.2 Virtual function calls and calls through function pointers

Virtual function calls and calls through function pointers can be handled as shown below.

```
int i,j;
int* x = NULL;

class A {
  public:
    virtual void foo(void) { x = &i; };
};

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

class C: public A {
};
```

```
proc0( A* param )
/* <x,x_init> */
{
   l1: param->foo();

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

In the above example, at the exit of *proc0* x points to i, if the type of *param_init* is such that *A::foo* gets called at *l1*, while its value remains unchanged if the type is such that *B::foo* gets called. Hence, values of x at the exit of *proc0* are inferred conditioned on the type of *param_init*. Since the same function 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 functions rather than just the type of *param_init*. Calls through function pointers can be handled similarly by conditioning on relationships between functions which could be called (i.e., those functions whose addresses have been stored in a function pointer and whose signatures match with the type of the function pointer through which the call has been made) and the initial values of function pointers.

8.3 Recursive Structures

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, consider the following:

```
class B {
   public:
     int* fld1;
     int* fld2;
};

class A {
   public:
     B* member1;
};

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.

```
class A;

class B {
   public:
```

```
    A* parent;
};

class A {
   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 the initial values of an arbitrary variable in the program; thus if we allow recursive structure in our objects, this requires a method for keeping the space of initial values finite. We have devised a way of dividing initial values into equivalence classes such that each class has a representative value, with the invariant that any points-to involving a member of a class is taken to involve any member of that same class.

To intuitively describe our algorithm, whose pseudocode is listed below, suppose we have a variable name $f_1.f_2 \ldots f_k$. [7] 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 f_1. Each internal node has as its children, all the fields of its type. A path through this tree to a leaf node is a variable name of the form above. Our algorithm essentially collapses specific paths in this tree which end in the same type to one path, thus handling the 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., all names (f_i) have different types); intuitively any "repeating" same sequences of certain types have been "collapsed".

As a rule of thumb, given a variable name $f_1.f_2 \ldots f_k$, we know that in its equivalence class all variables are of type type(f_k), and all begin with field selector f_2. Looking at the tree in which this name is a path, we can select each field f_i in turn, for $i = 2..k$, 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's f_s. Then we delete $f_{i+1}, \ldots, f_s$ 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. To obtain the corresponding intial value, we merely substitute corresponding initial values for each field, obtaining $f_{1_init}.f_{2_init}.\ldots.f_{k_init}$.

Given below is pseudo code for this algorithm and an illustrating example.

```
eqv_class get_eqiv_class( initial_value y )
/* y is an element of S */
{
   /* types eqv_class and initial_value are used for
      clarity, although physically they are represented
```

[7] The f_i can consist either of member names or dereferences thereof, e.g., `a.b, (*(a.c))`.

```
      as strings. So we can convert a value of one type
      to another. */
   eqv_class retVal;
   initial_value tmp;

   tmp = x_init;

   /* to take care of multi-level pointers like A **p */
   while ( get_type(tmp) is not a class type and y != tmp ) {
      tmp = (initial_value)concat( "(*", (string)tmp, ")" );
   }

   if ( tmp == y )
      /* tmp contains the equivalence class of y */
      return (eqv_class)tmp;

   /* type of tmp is a class type */
   retVal = (eqv_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" );

      /* to take care of multi-level pointers like A **p */
      while ( get_type(tmp) is not a class type and y != tmp ) {
         tmp = (initial_value)concat( "(*", (string)tmp, ")" );
         retVal = (eqv_class)concat( "(*", (string)retVal, ")" );
      }

      if ( tmp == y )
         /* retVal contains the equivalence class of y */
            return retVal;

      /* type of tmp is a class type */
      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 refer to 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 this scheme.

```
class B;
class C;
class D;

class A {
   public:
     B* fld1;
     B* fld2;
     A( void ) { fld1 = fld2 = NULL; };
};

class B {
   public:
     C* fld1;
     A* fld2;
     B( void ) { fld1 = NULL; fld2 = NULL; };
};
```

```
class C {
   public:
     B* fld1;
     D* fld2;
     C* fld3;
};

class D {
};


proc( A* param, int **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) ] = { (*formal_init) }.

}
```

Now we show, using the following example, how these equivalence classes are used to represent the transfer function of a procedure.

```
A,B,C, and D are as in the previous example.

B* x;

proc0()
{
  A* p;
  B* q;

  l1: p = new A();

  l2: q = new B();

  p->fld1 = q;

  l3: p->fld1->fld2 = new A();

  proc1( p );
  /* [p_init] consists of obj_l1 and obj_l3. So
     [p_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. */

}


proc1( A* p )
{
   A* tmp;

   tmp = p;

   while( tmp != NULL ) {
     tmp->fld2 = x;

     if ( tmp->fld1 != NULL )
       tmp = tmp->fld1->fld2;
     else
       break;
   }

  /* <[p_init].fld2,x_init> */
}
```

Note that although a transfer function may be represented in terms of equivalence classes, the final solution, unlike k-limiting, does not contain them.

8.4 Increase in precision

Analysis-using-abstract-values, in the presence of multi-level pointers, can not only increase efficiency (as in the single-level case) but also precision (unlike the single-level case) as illustrated by the following example.

```
class A {
  public:
    D* fld;
};

class C {
  public:
   D* fld;
};

C v;

C* q = &v;

proc0( A* par1, C* par2 )
{
  a: par1->fld = par2->fld;

  /* <par1_init.fld,par2_init.fld_init> */
}

proc1( )
{
  l1: A* p = new A;
  n1: v.fld = new D;

  proc0( p,q );
  s1:
}

.....

procn( )
{
  ln: A* p = new A;
  nn: v.fld = new D;

  proc0( p,q );
  sn:
}
```

Suppose we compute conditional points-tos and use a single assumed points-to. Moreover, suppose that the points-tos generated at the program point a are conditioned on the value of *par2*. This means, $\langle\langle par2,v \rangle$, $obj\_li.fld,obj\_nj \rangle$, where $1 \leq i,j \leq n$, will be generated at a. Each of these $O(n^2)$ points-tos will be propagated back to each *si* as the condition $\langle par2,v \rangle$is true at each call site of *proc0*.

Now consider what happens if we analyze using abstract values. Only $\langle par1_init.fld,par2_init.fld_init \rangle$is generated at a. At *si*, *par1_init.fld* is replaced by *obj_li.fld* and *par2_init.fld_init* is replaced by *obj_ni*. As a result only $\langle obj_li.fld,obj_ni \rangle$is added to the solution at *si*.

Of course this imprecision could be avoided by using more than one assumed points-to. But since assumed points-tos are propagated in a top-down manner rather than being inferred in a bottom-up manner, it is not always clear where to use more than one assumed points-to. Use of receiver object in an assumed-points-to, as shown earlier, takes care of a common case but not all possible cases. The advantage of bottom-up inference is that it can take care of these and many other situations requiring more than one assumed points-to uniformly.

8.5 Adaptive flow-sensitivity

Analysis-using-abstract-values can adaptively reduce flow-sensitivity in situations where flow-sensitivity is not needed, as shown in the following example.

```
class A {
  public:
    B* field;
};

proc0( )
{
  n1: p = new A;
  n2: p->field = new B;

  s1: proc1( p );
}

proc1( A* param )
{
   /* print does not modify A.field */
   s2: print( param->field );

   proc2( param );
}

proc2( A* param )
{
   s3: print( param->field );
}
```

A.field is only referenced, not modified, in *proc1* and *proc2*. During bottom-up inference phase, the algorithm notices - using information computed during preprocessing - that *A.field* is not modified in the strongly connected components to which *proc1* and *proc2* belong. So the need for *param_init.field_init* is not propagated upwards. As a result, during the top-down propagation phase, $\langle obj\_n1.field,obj\_n2 \rangle$ is not propagated to the entry node of *proc1* at *s1*. Instead, it is stored in a global table from which it is accessed at *s2* and *s3*.

Note that this cannot be done in conditional points-to analysis because the context needs to be maintained. As a result, even if a value is only referenced, it needs to be propagated in a flow-sensitive manner. This is shown in the following example.

```
class A{
  public:
    B* field;
};

class B{
};

class C{
  public:
    B* field;
    void update( A* param ) { field = param->field; };
};

proc0( )
{
  n1: A* p = new A;
  n2: p->field = new B;
  s1: proc2( p );
}

proc1( )
{
  n3: A* p = new A;
  n4: p->field = new B;
  s2: proc3( p );
}

proc2( A* param )
{
  n5: C* p = new C;
  s3: p->update( param );
}

proc3( A* param )
{
  n6: C* p = new C;
  s4: p->update( param );
}
```

Although *proc2*, *proc3* and *update* do not modify *A.field*, values of *obj_n1.field* and *obj_n3.field* need to be propagated at *s1* and *s2*. Otherwise, in *update* the relationship between the values of the receiver and the values of *param*→*field* won't be clear, and values might be propagated along unrealizable interprocedural paths. *Analysis-using-abstract-values* does not have this problem because it does not use assumed points-tos to represent context.

8.6 Efficient killing of points-tos involving fields of a structure

The *basic* algorithm cannot efficiently kill points-tos involving fields of a structure. As a result, it sometimes produces a solution which is as imprecise as that of a flow-insensitive algorithm, but at the cost of a flow-sensitive algorithm. The following example illustrates this.

```
class A {
    public:
       B* field;
};

proc0( )
{
   while( _ ) {
      s0: A*  p = new A;

      proc1( p );
   }
}

proc1( A* param )
{
  s1: param->field = new B;
  ...
  sn: param->field = new B;
}
```

Points-to $\langle$ *obj_*s_0*.field, obj_*s_1 $\rangle$ reaches s_2 from s_1. At s_1, *obj_*s_0*.field* is assigned *obj_*s_1. But it cannot kill the previous assignment because at run-time more than one object created at s_0 could be simultaneously active, and $\langle$ *obj_*s_0*.field, obj_*s_1 $\rangle$ could have been created for a different object than the one to which *param* points at s_2. This is true for each s_i. As a result, each $\langle$ *obj_*s_0*.field,obj_*s_j $\rangle$, where $1 \leq j \leq i-1$, reaches s_i. This results in a quadratic solution.

Now consider *analysis-using-abstract-values.* Only $\langle$ *param_init.field,obj_*s_{i-1} $\rangle$ reaches s_i, which is expanded to $\langle$ *obj_s0.field, obj_*s_{i-1} $\rangle$ when abstract initial values are replaced by actual values during the final phase of *analysis-using-abstract-values.* Hence, we get a linear solution.

9 Conclusion

In this paper we have addressed issues involved in designing a **scalable, flow-sensitive, and modular solution** for type inference and related compile-time analysis questions for statically typed object-oriented languages like C++ and Java. The following are some of the contributions of this paper:

1. Identification of a set of queries about pointers, that satisfy the needs of many applications including virtual function resolution and side-effect analysis.
2. A *basic* algorithm based on conditional points-to analysis for answering a subset of these queries, which is sufficient for virtual function resolution and side-effect analysis.
3. Definition of the hold-together relation, a method to compute it in a demand-driven manner, and a discussion of its significance for type inference.
4. Identification of three important characteristics of any scalable algorithm for type inference.

5. Since the *basic* algorithm does not have these characteristics, presentation of two techniques: preprocessing using non-standard types and *analysis-using-abstract-values*, which help in achieving these characteristics.

6. A way to do **demand-driven computation** by representing a part of the solution implicitly and expanding it on demand.

7. A way to do **modular analysis** through *analysis-using-abstract-values* which allows the analysis of the part of a program and avoids the need to keep the whole program in memory.

8. A proof that the worst-case bound for conditional may points-to analysis is $O(n^5)$.

9. A proof that in the presence of only single-level pointers *analysis-using-abstract-values* produces the precise solution for may points-to and improves the worst-case bound from $\mathbf{O}(n^5)$ **to** $\mathbf{O}(n^4)$.

10. Discussion of some initial ideas about how to do *analysis-using-abstract-values* in the presence of multi-level pointers and recursive types.

11. Using *analysis-using-abstract-values* and preprocessing using types, a method to **adaptively reduce flow-sensitivity** without affecting precision.

Acknowledgements The authors would like to thank Bill Landi, Phil Stocks, Sean Zhang, Jyh-shiarn Yur, Samir Datta and Sachin Lodha for helpful discussions.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison–Wesley, 1986.

[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.

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

[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*, 1994.

[JM79] Jones and Muchnick. In *Program Flow Analysis: Theory and Applications*, pages 119–120. Prentice Hall, 1979.

[Lan92] W. A. Landi. Interprocedural aliasing in the presence of pointers, phd thesis. Technical Report LCSR-TR-174, Dept of CS, Rutgers University, 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.

[LRZ93] W.A. Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[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.

[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.