Characterization and Automatic Identification of Type Infeasible Call Chains

Amie L. Souter, Lori L. Pollock

Computer and Information Sciences
University of Delaware
Newark, DE 19716

Abstract

Many software engineering applications utilize static program analyses to gain information about programs. Some applications perform static analysis over the whole program's call graph, while others are more interested in specific call chains within a program's call graph. A particular static call chain for an object-oriented program may in fact be impossible to execute, or infeasible, such that there is no input for which the chain will be taken. Identifying infeasible static call chains can save time and resources with respect to the targeted software development tool. This paper examines *type infeasibility* of call chains, which may be caused by inherently polymorphic call sites and are sometimes due to imprecision in call graphs. The problem of determining whether a call chain is type infeasible is defined and exemplified, and a key property characterizing type infeasible call chains is described. An empirical study was performed on a set of Java programs, and results from examining the call graphs of these programs are presented. Finally, an algorithm that automatically determines the type infeasibility of a call chain due to object parameters is presented.

Key words: object-oriented, static program analysis, call chains

1 Introduction

Program analyses can be categorized as those that compute information over the entire program's call graph and those that compute information along

Email addresses: `souter@cis.udel.edu` (Amie L. Souter), `pollock@cis.udel.edu` (Lori L. Pollock).

specific *call chains*, (also called *call strings*), within a program's call graph. While the program call graph represents all possible runtime calling relationships among a program's procedures, a *call chain* represents one path of execution through a call graph starting at a given procedure, recording a particular sequence or chain of procedure calls, eventually ending with invocation of a procedure of interest or one that makes no more procedure calls. There are two types of call chains. *Dynamic* call chains represent a particular execution's actions, and are computed by recording the actual sequence of procedure calls executed during a particular program run. Alternatively, *static* call chains represent a potential sequence of procedure calls, as computed by traversing the program call graph at static analysis time.

While dynamic call chains provide useful runtime information for both profiling and debugging, static call chains are also important for many software engineering applications, including static program slicing, software testing, program understanding, and interprocedural analysis used in optimizing compilers. Call chains play a role in program understanding activities, particularly tools based on static analysis, where the tools gather source code information about particular sequences of calls. Extracting such information using only the program call graph can possibly hinder the tool user due to the conservative nature of the call graph analysis. Interprocedural def-use analysis also uses call chains, in particular determining a call chain that covers the definition of a variable and a subsequent use of the same variable.

A fundamental problem associated with the computation of static call chains utilizing a program's call graph is the fact that a particular computed static call chain may not be executable. It is possible that there is no possible input that will execute the sequence of calls associated with the static call chain. Automatically detecting infeasible call chains is important, primarily to avoid wasted effort. For example, infeasible paths may be selected for testing, resulting in wasted effort to generate input data. Infeasible program slices used in a debugger or a program understanding tool could mislead the user and cause them to waste time and resources.

Infeasible call chains occur for several reasons. First, the imprecision in the analysis information usd in call graph construction can cause infeasible call chains to exist in the computed static call graph. Second, inherently polymorphic call sites in object-oriented programs, and their resulting call graphs also create infeasible call chains. In this paper, we examine the second cause, which we call *type infeasibility* of a static call chain. The first cause has been investigated by other researchers, and various methods for identifying infeasible paths and improving precision of static analyzers by excluding these paths from consideration have been developed, although it is impossible to solve the general problem of identifying all infeasible paths [1,2].

Object-oriented language features such as inheritance and polymorphism increase the ability to easily extend and reuse code. Though these features benefit the programmer, they hinder static analysis of such code due to the unknown exact calling relationship of methods caused by dynamic binding of message sends. A conservative call graph can always be built at compile time by including a set of edges for each call site, such that each edge from the node representing the caller points to a node representing a potential callee method based on a conservative static analysis, such as class hierarchy analysis [3]. Alternatively, more precise call graphs can also be constructed by precisely computing the set of potential receiver classes at each call site. Many approaches for exhaustively computing call graphs for object-oriented software have been developed [4–8,3,9–13]. Determining precise type information is difficult when dealing with polymorphic objects because an object can potentially be bound to different types throughout the execution of a program. A precise call graph provides advantages in terms of precision for client applications, but the time and space requirements to build a precise call graph are often prohibitive. In addition, the most precise call graph construction algorithm will fail to determine a singleton set of receiver objects for all call sites because some call sites are inherently polymorphic. Therefore, a less precise, but fast call graph construction algorithm is often used to generate a call graph. Regardless, inherently polymorphic call sites coupled with the precision of the call graph may cause some call chains to be type infeasible.

In this paper, we provide a formal treatment of the problem of type infeasible call chains, including several examples which illustrate different situations that a given call chain can be type infeasible. A characterization of the key property of a type infeasible call chain is presented, and the issues in determining the type infeasibility of a call chain are outlined. We have conducted an empirical investigation of a set of Java programs in which we gathered statistical information that helps reveal the potential for type infeasible call chains in a call graph using two different call graph construction algorithms of varying precision. Finally, an algorithm for automatically detecting whether a particular call chain is type infeasible due to object parameters is presented. Initial investigation into this problem was reported in [14]. This paper gives a more formal treatment of the problem and provides a more complete detailed description of the automatic detection algorithm, with a detailed example.

2 Definition and Example

A *call chain* is a tuple of call sites (CS_1- CS_2-...-CS_n) for which there exists a path in the call graph from CS_1 to CS_n, passing through each of the CS_i in the order specified by the chain. Because call chains can become arbitrarily large due to recursive procedures, call chains are typically restricted in their length

by recording only the last k calls for some $k \geq 0$, or by collapsing strongly connected regions, which represent recursion.

A particular call chain of interest, $\text{chain}_{interest}$, can be type infeasible, even when a precise call graph construction algorithm has been used to construct the call graph. During call graph construction, a call graph edge is added from the node for method a to the node representing method b because the type information at a call site CS in method a indicates that method b could be called at the call site CS. However, the type information computed at CS is based on the flow of type information from any path in the call graph to CS, not just a single path. That same type information may not flow along the call chain of interest, $chain_{interest}$, that passes through CS, but instead from a different call chain that also passes through CS. In this situation, $chain_{interest}$ is type infeasible due to the type infeasibility of the edge (a,b) representing method a calling method b at CS with respect to $chain_{interest}$. However, the call graph edge (a,b) needs to be included in the call graph because it lies along at least one type feasible call chain through CS.

Figure 1 illustrates how a type infeasible call chain can occur. The figure shows a segment of a call graph with multiple incoming edges as well as multiple outgoing edges from call graph node x. Each edge is labeled with the set of potential receiver types computed by a static reaching type analysis. A reaching type analysis determines the set of types that can reach a particular object. For type A to reach receiver o, there must be some execution path through the program that starts with a constructor of the form v = new A() followed by some chain of assignments of the form x_1 = v, x_2 = x_1,...,x_n=x_{n-1},o = x_n [12].

Consider the call chain represented by m-x-y_1 in Figure 1, and its potential type infeasibility. The question of type infeasibility can be phrased as: *Is there a set of types that propagate from* m *to* x *to* y_1 *that make the call chain feasible?* A type infeasible call chain can occur due to the fact that different type information is being propagated along the two incoming edges into the node x. The set of types {A,B,C} is being propagated from method m, while the set of types {A,R,S} is being propagated from method n. Thus, multiple callers are calling method x, each propagating a different set of types to method x. Both callers influence the set of potential receiver objects at call sites within method x. However, with respect to the single call chain m-x-y_1, the set of types from method n (which is not on the call chain of interest) will not influence the set of types necessary to invoke y_1 along the call chain m-x-y_1. Assume that the methods y_1 and y_2 are declared in class R and A, respectively. Then, method y_1 is included by having a receiver of type R, and similarly y_2 can be invoked through a receiver of type A. Any subclass of a base class could also be the receiver object of a method if the subclass does not have its own implementation of that method. This situation can be determined using

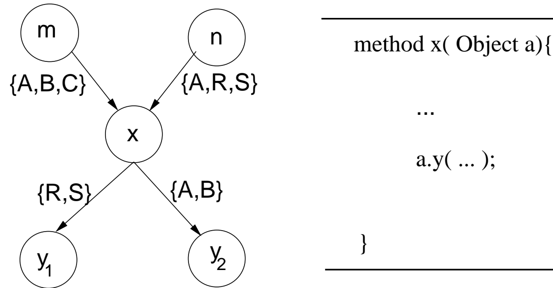


Fig. 1. Infeasible call chain example.

the class hierarchy, and therefore, in the evaluation of the receiver type, the assumption in this paper is that while the receiver possibly could be a set of types, this is not explicitly stated.

To determine the type feasibility of call chain m-x-y_1, the following question must be considered: *Is it possible to invoke y_1 based only on the types propagated from method m?* Given that the set {A,B,C} is propagated from m, and y_1 is invoked through the receiver type R, it can be determined that the call sequence m-x-y_1 is type infeasible. Note that the call chain m-x-y_2 is type feasible because type A is propagated from method m to the call site in x that calls y, which is invoked through the receiver type A. Therefore, this call graph includes infeasible call chains. Note that in order for the call graph to be correct, both outgoing edges from x to y_1 and x to y_2 are necessary because method m or method n could potentially be the caller of method x.

3 Causes of Type Infeasible Call Chains

Several different situations arise that possibly cause type infeasible call chains to occur. Each of these situations result from inherently polymorphic method calls throughout a program. This section characterizes and illustrates the different types of situations that cause type infeasible call chains.

Calling methods of formal parameter objects

When there are multiple callers to a method x, and each caller passes objects of different types to x, polymorphic call sites in x can be created. Type infeasible call chains can be created by the call sites in method x that call methods by y.z(...) where y is a formal parameter object of x. The potential for a type infeasible call chain is increased when the formal parameter is of static type Object.

To illustrate how type infeasible call chains can occur through parameters of a method, again refer to Figure 1. A call graph construction algorithm

such as CHA and RTA [15,3] could construct the call graph shown for this code segment. The multiple outgoing edges from x to y_1 and y_2 represent a polymorphic call site in which the exact method to be invoked cannot be determined during the construction of the call graph. In Figure 1, method m and method n both call method x. Note that the set of types passed to method x from each caller differs. The call site shown in method x is polymorphic due to the fact that statically the type of the receiver object a can not be determined. The static type of a is Object, therefore the exact type depends on the types propagated from the callers, namely methods m and n. If a single call chain were to be followed, say m-x-y, it is certain that only y_2 is possible, due to the set of types propagated from method m, and the knowledge that the formal parameter, a, must be bound to type A, which is the receiver type for y_2 to be called.

Polymorphic container classes

Container classes can potentially contain objects of different types, therefore, iterating through the objects of a container class could lead to unknown receiver objects at call sites.

To illustrate, consider the equals method (shown in Figure 2) from the Hashtable class of the Java class library. The important aspect of this code segment pertains to lines 9-19. The *while* loop is iterating through the objects stored in the Hashtable object that invokes the equals method. We are unable to determine the exact types stored in the Hashtable object by analyzing this method because it was implemented in a polymorphic fashion, allowing any type to be stored in the hash table. Therefore, the polymorphic call site at line 18 associated with the object value is inherently polymorphic. Potentially, method equals of any instantiated class that implements an equals method could be called at this call site because the types stored in the Hashtable are unknown.

If more information associated with the caller of this method (i.e., the receiver object) was known, a smaller set of objects stored in this container class might be able to be determined. In the context of a call chain and the knowledge of a smaller set of objects stored in the container class, it could potentially be determined that at the inherently polymorphic call site, (18: equals), some of the callees are infeasible with respect to the call chain of interest.

Polymorphic fields of objects

When there are multiple callers to a method, and the receiver object has polymorphic fields, the fields of the object are instantiated as different types, and the call sites associated with a field of the object could be polymorphic. This situation occurs when an object is created, and a single field of the object

```
1: public  boolean equals(Object o) {
2: if (o == this)
3:    return true;

4: if (!(o instanceof Map))
5:    return false;
6:Map t = (Map) o;
7:if (t.size() != size())
8:    return false;

9:Iterator i = entrySet().iterator();
10:while (i.hasNext()) {
11:   Entry e = (Entry) i.next();
12:   Object key = e.getKey();
13:   Object value = e.getValue();
14:       if (value == null) {
15:     if (!(t.get(key)==null &&
                    t.containsKey(key)))
16:    return false;
17:   } else {
18:if (!value.equals(t.get(key)))
19:    return false;
   }
}
20:return true;
```

Fig. 2. Polymorphic container class example.

could be instantiated to be of different types. The code segment in Figure 3 illustrates this situation. The method method_n is being invoked through two different receiver objects, namely, x and y at lines m3 and m4, respectively. The two objects x and y are instances of MyClass, which has a polymorphic field, field1, shown in the constructor of MyClass. Object x's and object y's field1 could potentially have different runtime types after instantiation due to line c1. Therefore, the call site in method_n at line n1 cannot be statically determined; two potential callees are possible, namely myType1's method_z and myType2's method_z. Even following a call chain, say method_m-method_n-method_z, starting with the call to method_n through object x at line m3, we would not be able to determine which method_z would be invoked through the receiver object field1 at line n1 because the type of field1 is dependent on a statement for which static analysis cannot provide exact information. Type infeasible call chains are possible in this situation, but static analysis can not help to detect them without information about the result of the conditional at line c1.

Infeasible Slices

Type infeasible call chains can also be constructed during program slicing. Traditionally, an interprocedural slice is constructed using a system dependence graph (SDG), which is a collection of program dependence graphs, one for each procedure in the program [17]. A program dependence graph (PDG) represents a procedure as a graph in which the control and data dependences are both represented explicitly [18]. Figure 4 illustrates an SDG for the code of

```
public void method_m(){

  m1: MyClass x = new MyClass(...);
  m2: MyClass y = new MyClass(...);
   ...
  m3: x.method_n();
   ....
  m4: y.method_n();
}

method_n(){
   ...
  n1:    field1.method_z();
   ...
}

class MyClass{
  Object field1;

c0:   MyClass() {
c1:     if(...)
c2:       Object field1 = new myType1();
c3:     else
c4:       Object field1 = new myType2();
  }
}
```

Fig. 3. Polymorphic field example.

methods C and D. Note that there are several types of edges in an SDG, namely control dependence edges (solid lines), data dependence edges (dashed lines), call and parameter bindings (dotted lines), and summary edges (bolded lines), which connect actual-in parameter nodes with actual-out parameter nodes if the value associated with the actual-in node may affect the value associated with the actual-out node. The nodes represent parameters and program statements. The illustrated SDG has been constructed according to Liang and Harrold for precise slices of object-oriented software [16]. Their modifications to the traditional SDG representation effectively solve problems associated with distinguishing data members of different objects, which allows for more precise data dependence information. However, they did not consider the problem associated with infeasible slices based on types.

In order to precisely model a slice for object-oriented programs, Liang and Harrold added type information for object parameters. For example, method C takes a Base object as a parameter. Therefore, the PDG for C represents that either a Derived or Base object could be based as a parameter. This is represented in the graph as a tree, where the parameter ba is the root of the tree and its children are possible types of ba. In this case, the possible types are Base and Derived.

To illustrate how a slice can be type infeasible, consider the slice for parameter b beginning in m1, which is shown as the shaded nodes in Figure 4. The slice was computed according to the traditional slicing algorithm by Horwitz et al[17]. However, the backward slice represents the call chain (D-C-Derived::m1).

```
class Base {
   int a,b;
   void vm(){
         a = a+ b;
   }
public:
  Base(){
            a = 0;
            b = 0;
  }

   m2(int i){

            b = b+1;
   }
   m1(){
            if(b > 0)
                   vm();
            b = b+1;
    }

 };
```

```
D() {
   Base o = new Base();
    C(o);
     o.m1();
}

class  Derived extends Base {
  int d;
  vm() {
     d = d +b;
  }
  m1() {
     b = b +1;
  }

  ...

  };
```

```
main() {
   int i;
   Base p;
   if( i > 0)
     p = new Base();
   else
     p = Derived();
   C(p);
   p.m1();

   }


   C(Base ba){

     ba.m1();
     ba.m2(1);

   }
```

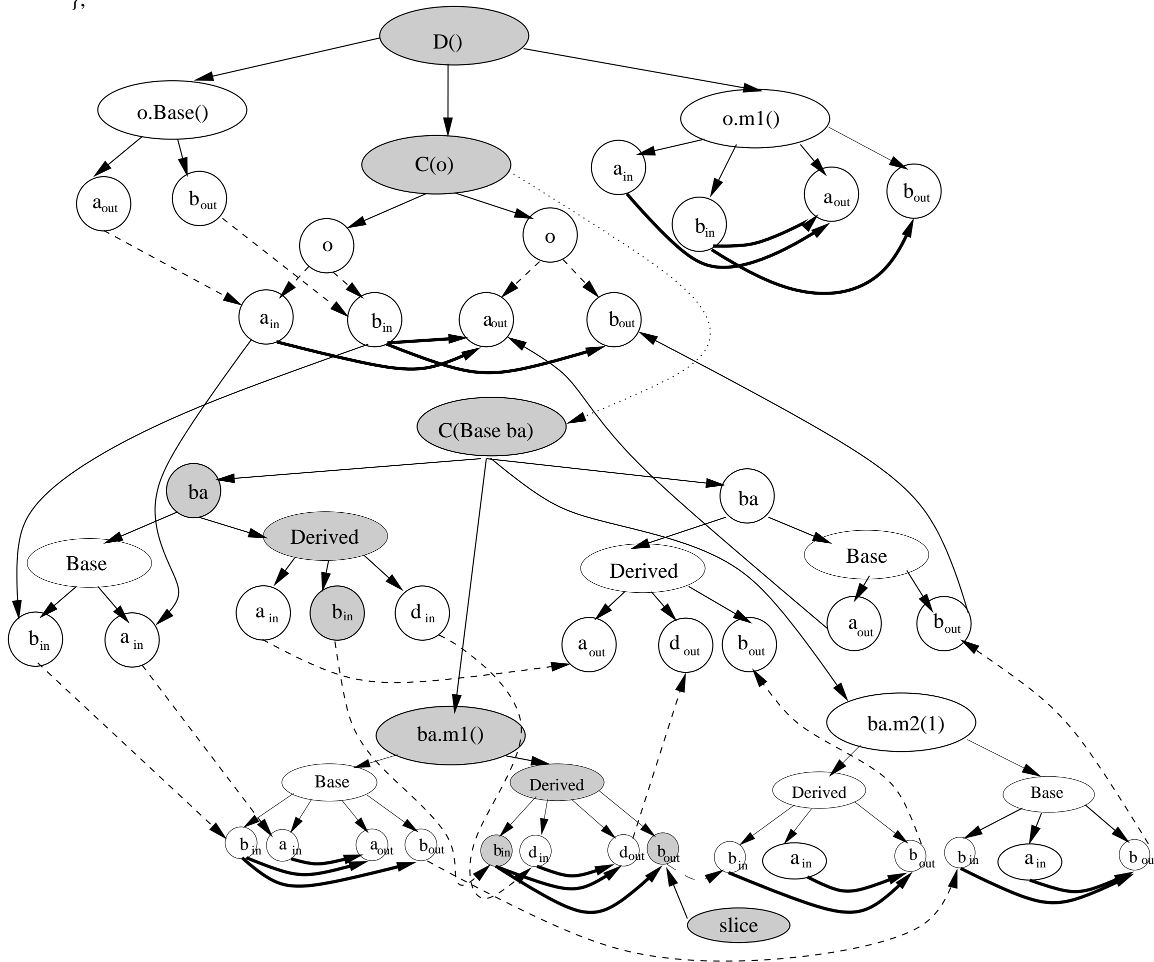


Fig. 4. Infeasible slice example [16].

Looking at the code for D, you can see that this is infeasible due to D creating only a Base type object. The call to m1 from within D could only be to the m1 method of the Base class. However, both the Derived and Base objects need to be represented in C's PDG, because the call from main to C could pass a Derived object as a parameter. Thus, the SDG is properly formed, but the

traditional slicing algorithm results in an infeasible slice.

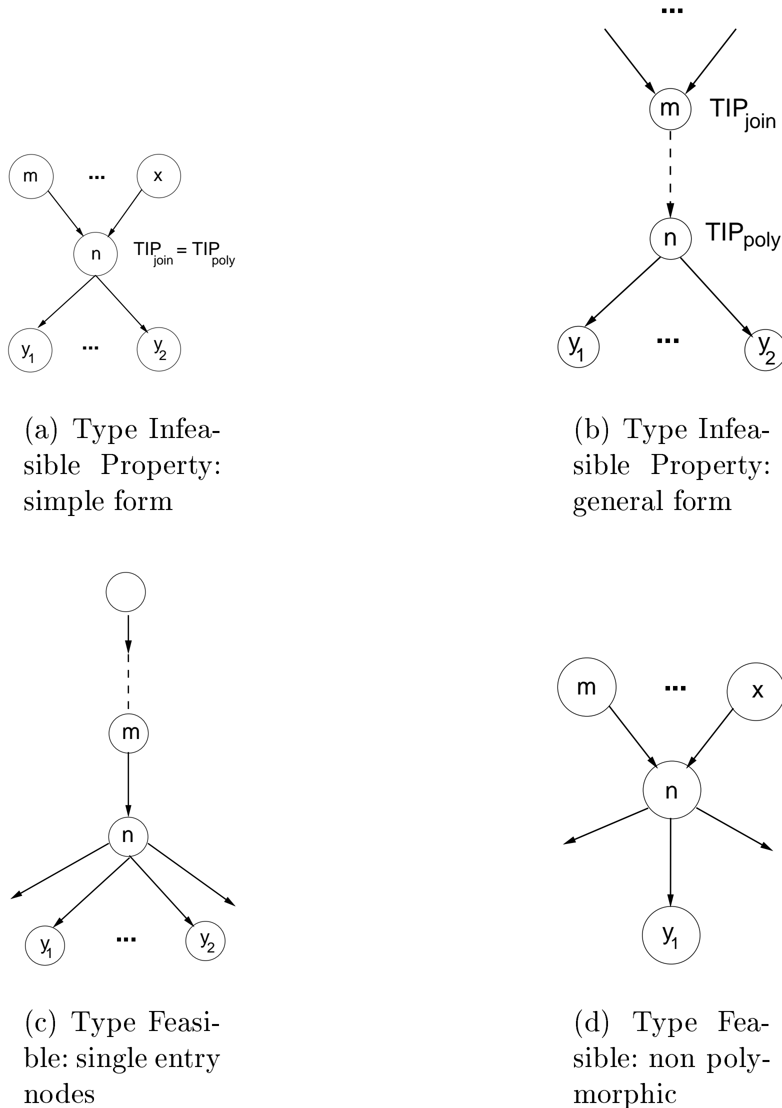
The call chain represented by this slice illustrates another example of how type infeasible call chains can be constructed for applications that utilize a graph that represents the entire program. The entire graph is necessary in order to account for all calling relationships, but additional analysis is needed in order to eliminate calls based on infeasible types along a particular call chain.

4 Type Infeasibility Property

This section presents a characterization of the properties observed in a call subgraph associated with a type infeasible call chain. A *type infeasible call chain* is a call chain that contains at least one edge that is type infeasible with respect to the call chain of interest, $\text{chain}_{interest}$. A type infeasible edge within a call chain, such as $\text{e}_{ti[chain_{interest}]}$, can be identified by the characteristics of the call subgraph leading to the edge $\text{e}_{ti[chain_{interest}]}$. Since these characteristics can be automatically identified and used to determine the type infeasibility of $\text{chain}_{interest}$, a call subgraph involved in $\text{chain}_{interest}$ and exhibiting these characteristics is said to exhibit the *type infeasibility property*, (TIP). The TIP of a call subgraph g consists of two parts: (1) the form of the subgraph g, and (2) the relationships between the reaching type sets along the edges associated with the subgraph g.

Form of the Call Subgraph: A call subgraph g exhibiting the type infeasibility property must include (1) a node $\text{TIP}_{poly} \in \text{chain}_{interest}$ with at least one polymorphic call site, and (2) a node TIP_{join}, which is in $\text{chain}_{interest}$, an ancestor of TIP_{poly}, and has more than one caller. For the potential presence of type infeasibility with respect to $\text{chain}_{interest}$, this form of a call subgraph is necessary because it enables multiple callers (to TIP_{join}) to potentially propagate different type information to the method, represented by TIP_{poly}, containing the polymorphic call site. Therefore, each caller of TIP_{join} is contributing to potential receivers at the polymorphic call sites in the method represented by TIP_{poly}. Without multiple callers to some node in $\text{chain}_{interest}$ and a polymorphic call site in $\text{chain}_{interest}$, there is no possibility that a $\text{chain}_{interest}$ could be type infeasible. TIP_{join} nodes are critical in the analysis of type infeasible call chains because type information would have been merged at the entry of the method represented by TIP_{join} due to multiple callers.

Figures 5(a) and 5(b) illustrate the forms of a subgraph exhibiting the type infeasibility property. Figure 5(a) is the most simple form of the subgraph, in which the same node, n in this case, is both TIP_{poly} and TIP_{join}. Figure 5(b) is a more general form of the subgraph in which there exist single entry nodes



(a) Type Infeasible Property: simple form

(b) Type Infeasible Property: general form

(c) Type Feasible: single entry nodes

(d) Type Feasible: non polymorphic

Fig. 5. Type feasibility subgraph forms.

between a TIP_{join} and its closest TIP_{poly} in $\text{chain}_{interest}$. Note that a given $\text{chain}_{interest}$ could include multiple TIP_{join} and TIP_{poly} nodes. The algorithms for detecting type infeasible call chains handle this possibility.

Figures 5(c) and 5(d) illustrate other possible subgraph forms that may appear to lead to type infeasible call chains, but they do not cause type infeasibility. In Figure 5(c), node n, which contains a polymorphic call site (i.e., n is TIP_{poly}), has one incoming edge, but multiple outgoing edges. Since there is only one caller of method n, and $\text{chain}_{interst}$ consists only of single entry nodes (i.e., there exists no TIP_{join} in $\text{chain}_{interest}$), all of the outgoing edges of n must be type feasible with respect to $\text{chain}_{interest}$ because the types are either propagated from one of the single callers along the call chain to method n, or generated inside method n.

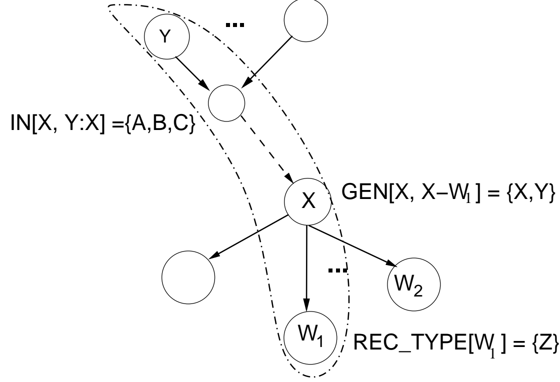


Fig. 6. Type infeasibility property example for reaching type relations of subgraph.

Figure 5(d) illustrates the case where n has multiple callers (i.e., n is TIP_{join}), and possibly multiple outgoing edges, but all outgoing edges are non-polymorphic (i.e., there exist no TIP_{poly} in $\text{chain}_{interest}$). Therefore, for each (non-polymorphic) call site in n, only a single type was propagated from the set of callers to the call site or the call site is independent of the incoming type information, and instead the receiver is based on type information generated within method n. Thus, none of the outgoing edges of n are type infeasible with respect to $\text{chain}_{interest}$.

Reaching Type Relations: The second condition for a call subgraph g involved in $\text{chain}_{interest}$ to cause type infeasibility with respect to $\text{chain}_{interest}$ involves the relations between the reaching type sets along the edges involved in the subgraph g. Subgraph g is defined to include only the nodes in $\text{chain}_{interest}$ from a TIP_{join} through the closest $\text{TIP}_{poly} \in \text{chain}_{interest}$, and relevant nodes and edges connected to this subchain of $\text{chain}_{interest}$. These relations can be formulated as a data flow equation in terms of the types propagating from callers of the TIP_{join} node, types generated within the method represented by TIP_{poly}, and types propagating to callees of TIP_{poly} through call sites in TIP_{poly}. Figure 6 shows a segment of a call graph, for which the type infeasibility property of node X with respect to the call chain Y-...-X-W_1 is considered. In this example, the TIP_{join} node is the successor of Y and the TIP_{poly} is X. The question is: *Is the call from X to W_1 type feasible with respect to the call chain* Y-...-X-W_1*?*

The formulation of this question in terms of a data flow equation is as follows. For a given method M, let

REC_TYPE[M] = the receiver type needed to invoke the method M.

GEN[M,CS] = the set of types generated in the method M and reaching the call site CS in method M. This set of types can be calculated in several different ways, either a conservative local analysis that makes conservative assumptions about the effects of call sites before CS in M, or an analysis that

includes a more precise side effect analysis to more precisely obtain the effects of the call sites before CS in M.

IN[M] = the set of all types reaching the entry of method M from any caller of M.

IN[M,N-M] = the set of all types reaching the entry of method M, from the call site represented by the edge N-M in $\text{chain}_{interest}$.

IN[M,N:M] = the set of all types reaching the entry of method M, from the call site N, through the call graph path N:M in $\text{chain}_{interest}$.

In terms of the call chain Y-...-X-W_1 in Figure 6, the contents of these sets are:

$$REC_TYPE[W_1] = \{Z\}$$
$$GEN[X, \text{X-W}_1] = \{X, Y\}$$
$$IN[X, \text{Y:X}] = \{A, B, C\}$$

The edge X-W_1 is said to be type feasible with respect to the call chain Y-...-X-W_1 if:

$$\{Z\} \in (\{A, B, C\} \cup \{X, Y\})$$

That is, only if:

$$REC_TYPE[W_1] \in (IN[X, \text{Y:X}] \cup GEN[X, \text{X-W}_1])$$

This condition can be generalized as the *test for type feasibility* as follows.

Definition: Test for type feasibility.

Given a call chain C that contains a TIP_{poly} node N_i, a TIP_{join} node N_j, which is located before TIP_{poly} on the call chain C with no intervening TIP_{join} nodes, the edge N_i-N_{i+1} in C is type feasible with respect to the call chain C, only if

$$REC_TYPE[N_{i+1}] \in$$
$$(IN[N_i, \text{N}_{j-1}\text{:N}_i] \cup GEN[N_i, \text{N}_i\text{-N}_{i+1}])$$

This test states that in order for an edge in a call chain C to be type feasible with respect to C, the receiver type that is needed to invoke the next method in the call chain C must be included in the set of types propagated from the

caller of TIP$_{join}$ in the call chain to TIP$_{poly}$, or included in the set of types generated in TIP$_{poly}$ prior to the call site in question.

Theorem: Test for type feasibility.

Proof: Assume a call chain C is type feasible and C contains a TIP$_{poly}$ node N$_i$, a TIP$_{join}$ node N$_j$, which is located before TIP$_{poly}$ on the call chain C with no intervening TIP$_{join}$ nodes. Then by definition, the edge N$_i$-N$_{i+1}$ in C is type feasible with respect to the call chain C only if,

$$REC_TYPE[N_{i+1}] \in \\ (IN[N_i, \mathrm{N}_{j-1}{:}\mathrm{N}_i] \cup GEN[N_i, \mathrm{N}_i\text{-}\mathrm{N}_{i+1}])$$

Suppose by way of contradiction that the call chain C is feasible, but the test for type feasibility does not hold, i.e.,

$$REC_TYPE[N_{i+1}] \notin \\ (IN[N_i, \mathrm{N}_{j-1}{:}\mathrm{N}_i] \cup GEN[N_i, \mathrm{N}_i\text{-}\mathrm{N}_{i+1}])$$

If the REC_TYPE[N$_{i+1}$] $\notin$ (IN[N$_i$,N$_{j-1}$:N$_i$] $\cup$ GEN[N$_i$,N$_i$-N$_{i+1}$]), then the receiver type for N$_{i+1}$ does not reach the call site N$_i$-N$_{i+1}$. But, in order to invoke N$_{i+1}$ from N$_i$ the receiver type of the call to N$_{i+1}$ must reach the call site N$_i$ - N$_{i+1}$. Thus, the call from N$_i$ to N$_{i+1}$ at N$_i$-N$_{i+1}$ call site is not feasible. Thus, types at N$_i$-N$_{i+1}$ call site prevent N$_{i+1}$ from being called. This infeasibility of call chain C contradicts the assumption that call chain C is feasible. □

5 Empirical Study

An empirical study of the characteristics of call graphs for object-oriented programs was conducted. The goal of this study was to determine the frequency that the type infeasibility property occurs within call graphs, as well as to determine how the precision of the call graph affects the frequency of the type infeasibility property. Because it is impractical to examine all possible call chains and their type infeasibility properties, an estimate was computed by focusing on the frequency in which TIP_{poly} nodes occur as well as nodes exhibiting the simplest TIP subgraph form (i.e., $TIP_{poly} = TIP_{join}$). The frequency of TIP_{poly} nodes provides an indication of the largest possible number of potentially infeasible edges, while the $TIP_{poly} = TIP_{join}$ nodes provides an estimate of the potential for infeasible call chains occurring within a program.

Table 1
Program characteristics.

Name	Problem Domain	# of lines	# of byte instrs	# of classes	# of methods
compress	text compression	910	22968	164	641
db	database retrieval	1026	24842	161	697
javalex	scanner generator	7590	30606	158	824
jasmin	java assembler	~7500	38898	193	841
jack	parser generator	~7500	40225	208	928
jess	expert system	9734	39410	262	1173
soot	optimization framework	10000+	173910	1312	8314

The benchmarks analyzed were taken from the SpecJVM98 suite and various websites containing free Java programs. The overall program characteristics and problem domain of the benchmarks are shown in Table 1. The table shows the number of lines of Java source code, the number of bytecode instructions, the number of classes, and the number of methods in each program in order to illustrate the size and complexity of the programs studied. The number of classes and methods include both user code and Java library code called from the user program.

Tables 2 and 3 present call graph characteristics for the benchmark programs using two different call graph construction algorithms, namely a conservative call graph construction algorithm, similar to RTA [15], and a more precise call graph construction algorithm, similar to CPA [7]. The tables show the number of call graph nodes, which corresponds to the number of reachable methods in the program including library methods, the percentage of call graph nodes that exhibit the simple form of a subgraph for type infeasibility (these nodes are called simple-TIP-prone), the percentage of methods containing a polymorphic call site (TIP_{poly}), and the percentage of polymorphic call sites within the program. The percentage of simple-TIP-prone nodes was calculated as the total number of nodes identified in the call graph having multiple incoming edges and a polymorphic call site, divided by the total number of nodes in the call graph. TIP_{poly} nodes were calculated as the total number of methods containing at least one polymorphic call site, divided by the total number of nodes in the call graph. Similarly, the percentage of polymorphic call sites was calculated by the total number of polymorphic call sites (edges) divided by the total number of call sites (edges) in the program.

The data in the tables reveal that the percentage of simple-TIP-prone nodes is on average approximately 10% of the nodes in a call graph. This statistic verifies that type infeasible call chains are possible in real programs. A

Table 2
Call graph characteristics for the RTA call graph construction algorithm.

Name	# of CG nodes	% TIP prone nodes	% poly node	% poly CS
compress	820	9	13	4
db	885	10	16	5
javalex	824	10	16	5
jasmin	1022	10	15	5
jack	1118	10	17	9
jess	1326	8	13	3
soot	9582	19	39	14

Table 3
Call graph characteristics for the CPA call graph construction algorithm.

Name	# of CG nodes	% TIP prone nodes	% poly node	% poly CS
compress	641	6	11	3
db	697	8	14	3
javalex	745	9	13	3
jasmin	896	7	13	3
jack	928	9	17	9
jess	1173	6	10	2
soot	8314	17	39	11

more precise call graph construction algorithm does decrease the percentage of simple-TIP-prone nodes because more unreachable methods are pruned from the call graph, but TIP-prone nodes still exist in call graphs constructed with a more precise algorithm. The percentage of TIP_{poly} nodes is significantly greater than simple-TIP-prone nodes. The number of TIP-prone subgraphs, which are more difficult to count, should be somewhere between the number of simple-TIP-prone nodes and TIP_{poly} nodes.

The number of polymorphic call sites within the program is relatively low, but they are present. Also, it is believed that one reason that these statistics are relatively low in general is that these applications do not reflect strong object-oriented design principles, and therefore show little polymorphism in some cases. Also, it is believed that as programmers become more experienced, the amount of polymorphism in a program will increase. In addition, as more generic/reusable code is used, inherently polymorphic call sites will most likely

become more prevalent. However, programs that contain a large number of polymorphic call sites are not likely to frequently occur, since experienced programmers often avoid using complex inheritance structures because of the inefficient code that is generated to handle polymorphism.

6 Automatic Detection of Type Infeasibility

The overall strategy for determining the type feasibility of a call chain is based on a backward analysis to determine whether the receiver object at a call site can potentially be of a certain type. To motivate this approach, consider the call chain Y-...-X-W_1 in Figure 6 and the question: *Is it possible, in method X, to call method W_1?* To answer this question, we must determine if type Z could reach the receiver object that invokes W_1 through call chain Y-X-W_1. Note that it is impossible to rely solely on the reaching type sets that are computed for each edge (or node) for call graph construction, because the reaching type set for a given point in the program is based on flow from any path to that point. Instead, reaching type information is needed with respect to $chain_{interest}$, which is a subset of the entire reaching type set at a given program point.

Our type feasibility detection algorithm illustrated in Figure 7 relies on precomputation of IN and GEN sets defined in Section 4. In particular, GEN[M,CS], the set of types generated in method M and reaching the call site CS in method M, must be available for each call site along $chain_{interest}$. Similarly, IN[M,N-M], the set of all types reaching the entry of method M from the call site represented by the edge N-M must be available.

In addition to the precomputed IN and GEN sets, the algorithm expects the program call graph and the call chain $chain_{interest}$ as input. Given $chain_{interest}$ = (CS_0,...,CS_n), the call chain is first traversed to create a set of all TIP-prone subchains. A TIP-prone subchain of $chain_{interest}$ is defined to be $(TIP_{join-1}, TIP_{join}, TIP_{poly}, TIP_{poly+1})$, where TIP_{join-1} is the predecessor of the last TIP_{join} node in the $chain_{interest}$, TIP_{join} is the closest join node to CS_0, TIP_{poly} is the node containing a polymorphic call site, and TIP_{poly+1} is the method invoked at the polymorphic call site. Each TIP-prone subchain is analyzed separately because each polymorphic call site must be analyzed to determine if it is feasible, as any of them could cause $chain_{interest}$ to be type infeasible. Next, the subset of the TIP-prone subchains that represent potential type infeasibility due only to object parameters is identified by performing a reaching type analysis from the formal parameters of the TIP_{poly} method to the polymorphic call site in TIP_{poly}. If a formal parameter can influence the receiver type at a polymorphic call site in the method, represented by TIP_{poly}, then the subchain is maintained for further analysis.

Algorithm: Type_Infeasibility_Detector(chain$_{interest}$, CG)

```
Input:        chain_interest = (CS_0,...,CS_n)
              CG = program call graph
Precomputed:
              GEN and IN sets for methods in chain_interest

Output:       Determination of type feasibility of chain_interest

              Identify TIP-prone subchains (TIP_{join-1}, TIP_join, TIP_poly, TIP_{poly+1})
              foreach TIP-prone subchain do
                 Let X = method represented by TIP_poly
                 Let REC_type = receiver type for TIP_{poly+1} method to be called at CS_i
                 Let Y = caller(X) along chain_interest

                 do
                    feasible =false;
                    result = receiver_gen(REC_type, X,CS_i) //examine locally generated
                                                            // reaching definitions for CS_i
                    if( result == feas )
                       feasible = true;
                    else if( result == infeas )
                       return INFEASIBLE
                    else if( !receiver_in(REC_type, Y, X) )
                       return INFEASIBLE
                    else
                       Let CS_i = call site Y-X in Y
                       Let REC_type = receiver type to invoke X
                       Let X = Y                      // along call chain_interest
                       Let Y = caller(X)              // along call chain_interest
                 while(X != TIP_join and !feasible)

                 if( X == TIP_join)
                    if( REC_type ∉ GEN[X, CS_i] or REC_type ∉ IN[X] )
                       return INFEASIBLE
              end for
              return FEASIBLE
```

Fig. 7. Type infeasibility detector based on precomputed GEN and IN sets.

After computing the TIP-prone subchains, the algorithm begins by processing each TIP-prone subchain. The set of subchains could be processed in any order, however, it might be more efficient to analyze the subchains based on the length of the subchain from TIP$_{join}$ to TIP$_{poly}$, (i.e., shortest to longest subchain), because potentially less analysis is needed if short subchains are analyzed first and found to be infeasible, since, finding an infeasible subchain

causes the whole chain to be infeasible. However, additional experimentation needs to be performed to determine if any such ordering would reduce analysis time.

For a given TIP-prone subchain, the information for the TIP_{poly} node is examined to determine the receiver type REC_{type} for the particular method TIP_{poly+1} to be called at the polymorphic call site CS_i in question. This receiver type information could be a set of types based on the class hierarchy and the caller of the TIP_{poly} method. To determine whether the subchain is feasible due to locally generated type information in TIP_{poly}, a *receiver_gen* test is performed as follows,

$$receiver_gen(REC_{type}, TIP_{poly}, CS_i) = \begin{cases} feas & \text{if REC}_{type} \in \text{GEN[TIP}_{poly}\text{,CS}_i] \\ infeas & \text{if GEN[TIP}_{poly}\text{,CS}_i] \neq \text{empty} \wedge \\ & \text{REC}_{type} \notin \text{GEN[TIP}_{poly}\text{, CS}_i] \\ undefined & \text{otherwise} \end{cases}$$

If *receiver_gen* holds, then the subchain is type feasible, and the algorithm proceeds to the next TIP-prone subchain. If GEN is non-empty, but REC_{type} is not locally generated, then the call chain is infeasible because the other types for CS_i generated locally prevent REC_{type} from possibly propagating to CS_i from callers of the current method. If GEN is empty, analysis of the chain proceeds with another test, *receiver_in*, based on the types in IN flowing from the caller Y into the currently analyzed method X, where

$$receiver_in(REC_{type}, Y, X) = \begin{cases} true & \text{if REC}_{type} \in \text{IN[X,Y-X]} \\ false & \text{otherwise} \end{cases}$$

Again, this is only the set of types flowing directly from Y to X along the call chain. Types flowing from other callers of X are not considered.

If *receiver_in* does not hold, then the edge X-CS_i is type infeasible and INFEASIBLE is reported, and the algorithm halts. Otherwise, the backwards traversal is continued because the presence of $\text{REC}_{type} \in$ IN[X,Y-X] does not imply that the edge X-CS_i is feasible, because the $\text{REC}_{type} \in$ IN[X,Y-X] could occur due to a path to X that is not included in $chain_{interest}$. The traversal continues backwards to the caller of X along $\text{chain}_{interest}$, first checking GEN[caller(X), call site(caller(X)-X))], and then IN[X, caller(X)-X] as above.

The backwards traversal continues until the last join node in the call chain is encountered (i.e., TIP_{join}), or a GEN of the REC_{type} is determined. TIP_{join} is the stopping point due to the fact that no more join points along this TIP-prone subchain will be encountered. Therefore, at this point, the set of IN types are only those types that can influence the polymorphic call site along the call chain. After reaching TIP_{join}, another check of $REC_{type} \in$

$GEN[TIP_{join-1}, TIP_{join}]$ is performed, if this test holds, then this TIP-prone subchain of $\text{chain}_{interest}$ is feasible, and the same process is repeated for the other TIP-prone subchains. If the test does not hold and $\text{TIP}_{join-1} \neq \text{root}_{callgraph}$, then one more test must be performed, $REC_{type} \in IN[TIP_{join}]$. The REC_{type} must come from the caller of TIP_{join} on $\text{chain}_{interest}$. If the test does not hold, then the call chain is infeasible, and no other TIP-prone subchains need to be processed. In order for a call chain to be feasible, all TIP-prone subchains must be determined to be feasible.

To illustrate the algorithm, consider the call chain A:F_1 shown in Figure 8. The question at hand is: *Is call chain A:F_1 feasible?* The algorithm proceeds by determining the set of TIP-prone subchains to be analyzed. In this example, the following TIP-prone subchains are constructed: (A, B(TIP_{join}), C(TIP_{join}), D_1(TIP_{join}), E(TIP_{poly}), F_1(TIP_{poly+1}), (A, B(TIP_{join}), C(TIP_{poly}), D_1(TIP_{poly+1})). Assume that both C and E have polymorphic call sites based on polymorphic parameter objects.

Next, the algorithm processes each TIP-prone subchain. In this example, we begin with subchain (A, B(TIP_{join}), C(TIP_{join}), D_1(TIP_{join}), E(TIP_{poly}),F_1(TIP_{poly+1})), where the REC_{type} to invoke F_1 is determined to be P, and the caller of E is D_1. Next, the test of *receiver_gen(REC_{type}, E, E-F_1)* is performed. If the test indicates feasible, then the algorithm proceeds to the next subchain, otherwise as in this case, the test indicates undefined, so the *receiver_in(REC_{type}, D_1, E)* test must be performed. Since this test indicates true, the backward traversal along the call chain continues to determine if the REC_{type} is found along this call chain. By performing these tests at each node in the call chain, it can be determined whether the REC_{type} was generated only along $\text{chain}_{interest}$ and not from a caller at a join point in the chain. In this case, the backward analysis continues all the way to the last join node B, and then it is determined that P is generated in the predecessor, A, of B. Therefore, the polymorphic call to F_1 from E is type feasible with respect to $\text{chain}_{interest}$.

The next TIP-prone subchain, (A, B(TIP_{join}), C(TIP_{poly}), D_1(TIP_{poly+1})), is then processed determining whether the call site C-D_1 is type feasible with respect to $\text{chain}_{interest}$. REC_{type} = M is determined and the caller of C in $\text{chain}_{interest}$ is B. The *receiver_gen(REC_{type}, C, D_1)* test is performed, which indicates infeasible, and the analysis of $\text{chain}_{interest}$ is complete.

Overall, this approach is straightforward and does not require control flow graphs to be maintained for each method since the local data flow is precomputed with respect to each call site. The downfall of this approach is the need to have precomputed GEN and IN sets relative to each polymorphic call site for each method.

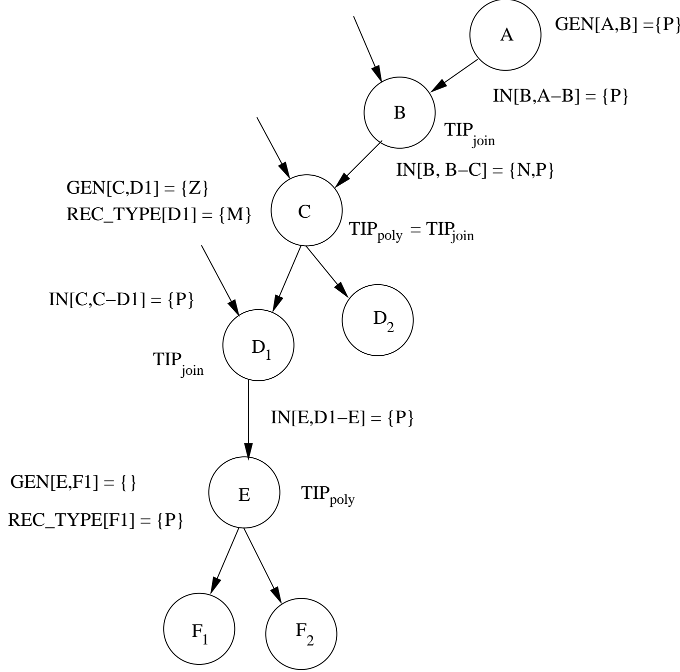


Fig. 8. Infeasible call chain example.

The number of IN and GEN sets that needs to be maintained for type feasibility detection is proportional to the size of $\text{chain}_{interest}$. The GEN and IN sets could be precomputed for the entire program or computed as needed for each call chain in question. The time to process $\text{chain}_{interest}$ is proportional to the number of TIP-prone subchains created, which is based on the number of polymorphic call sites in $\text{chain}_{interest}$, and the length of $\text{chain}_{interest}$ from the first TIP_{join} node to each polymorphic call site. The number of polymorphic call sites is relatively small if a precise call graph construction algorithm is used.

7 Related Work

There are several areas of research related to this paper, namely, type analysis and call graph construction, call strings for context-sensitive interprocedural analysis, demand-driven analysis, and infeasible path detection.

Type inference (or type analysis) is important for optimization of object-oriented languages. Without the knowledge of any type information at compile time, most optimizations can not be performed due to inheritance and polymorphism. Type analysis computes possible types (sets of classes) for each expression in a program at compile time. In order to compute interprocedural type information, interprocedural control flow (i.e., a call graph) and data flow information must be computed simultaneously. While there exists a number

of different techniques for whole program type and class analysis [4–8,3,9,10], many of these techniques suffer by not scaling well to large programs and not having the capability of handling incomplete programs.

Context-sensitive analyses have been the focus of a large amount of research. Such analyses distinguish information propagated to procedures from multiple callers within a program. One way of capturing context sensitivity is through the use of call strings, which encode the path taken to a callee. Recursion causes problems with this encoding; therefore, bounded call strings that restrict the length are also used. In terms of call graphs, context-sensitive CFA [4,19,20] algorithms use the concept of call strings to increase the precision of the analysis. The problem with such analyses is the cost of maintaining the context information for each call string to each procedure in the program.

Demand-driven analysis limits analysis to only the portions of the program that supply the needed information at a given program point, with the goal of avoiding an exhaustive analysis over the entire program. Duesterwald and Soffa investigated a demand-driven approach for efficient interprocedural data flow analysis of procedural languages in [21]. Reps et al. have also developed an approach to demand-driven analysis [22].

Infeasible path detection has been shown to be useful for improving several software engineering applications, namely, path testing, def-use testing, and static slicing. Bodik, Gupta, and Soffa have developed an analysis to determine infeasible def-use pairs through the detection of branch correlation[1]. This research was in the context of procedural languages, and mainly focused on scalar variables. Therefore, trying to eliminate infeasible paths or call chains associated with objects was not an issue.

8 Conclusions and Future Work

In this paper, we have characterized type infeasible call chains and programming situations that can cause these chains. Our empirical study of Java programs demonstrates how more precise call graph construction algorithms can reduce the potential for type infeasible call chains. However, these chains can still occur due to inherently polymorphic call sites, and furthermore we believe they will become more prevalent with increased use of object-oriented design principles. Thus, it is important to attempt to identify type infeasibility in order to reduce the wasted effort in testing and debugging and misleading information provided to programmers by program understanding tools.

We are currently refining an algorithm that automatic detects infeasible call chains, without precomputing sets. In addition, while we present an approach

for statically detecting one kind of type infeasible call chain, those created through object parameters, we believe there is more investigation to be done in this area. We plan to examine other situations that can lead to infeasible call chains, in addition to implementing our algorithms and performing experiments to evaluate the occurrences and various causes of type infeasible call chains in real programs.

References

[1] R. Bodik, R. Gupta, M. L. Soffa, Refining data flow information using infeasible paths, in: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT symposium on Software Engineering (ESEC/FSE 97), 1997.

[2] L. H. Holley, B. K. Rosen, Qualified data flow problems, IEEE Transactions on Software Engineering 7 (1) (1981) 60–78.

[3] J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in: Proceedings of the European Conference on Object-Oriented Programming, 1995.

[4] O. Shivers, Control flow analysis of higher-order languages, Ph.D. thesis, CMU (1991).

[5] J. Palsberg, M. Schwartzbach, Object-oriented type inference, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 1991.

[6] J. Plevyak, A. Chien, Precise concrete type inference for object-oriented languages, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 1994.

[7] O. Agesen, The cartesian product algorithm: Simple and precise type inference of parametric polymorphism, in: Proceedings of the European Conference on Object-Oriented Programming, 1995.

[8] A. Diwan, J. E. B. Moss, K. 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.

[9] H. Pande, B. Ryder, Static type determination for C++, in: Proceedings of Sixth USENIX C++ Technical Conference, 1994.

[10] G. DeFouw, D. Grove, C. Chambers, Fast interprocedural class analysis, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998.

[11] A. Rountev, A. Milanova, B. G. Ryder, Points-to analysis for Java based using annotated constraints, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 2001.

[12] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-Rai, P. Lam, E. Gagnon, C. Godin, Practical virtual method call resolution for Java, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 2000.

[13] F. Tip, J. Palsberg, Scalable propagation-based call graph construction algorithms, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, 2000.

[14] A. L. Souter, L. L. Pollock, Type infeasible call chains, in: Proceedings of the IEEE Workshop on Source Code Analysis and Manipulation, 2001.

[15] D. F. Bacon, P. F. Sweeney, Fast static analysis of C++ virtual function calls, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, 1996.

[16] D. Liang, M. J. Harrold, Slicing objects using system dependence graphs, in: Proceedings of the IEEE International Conference on Software Maintenance, 1998, pp. 358–367.

[17] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1).

[18] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.

[19] D. Grove, Effective interprocedural optimization of object-oriented languages, Ph.D. thesis, University of Washington (1998).

[20] J. Vitek, R. N. Horspool, J. Uhl, Compile-time analysis of object-oriented programs, in: Proceedings of the 4th International Conference on Compiler Construction, 1992.

[21] E. Duesterwald, R. Gupta, M. L. Soffa, A practical framework for demand-driven interprocedural data flow analysis, ACM Transactions on Programming Languages and Systems 19 (6) (1997) 992–1030.

[22] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural data flow analysis via graph reachability, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995.