

Type Infeasible Call Chains*

Amie L. Souter and Lori L. Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{souter, pollock}@cis.udel.edu

Abstract

While some software engineering applications perform static analysis over the whole program call graph, others are more interested in specific call chains within a program's call graph. It is thus important to identify when a particular static call chain for an object-oriented program may not be executable, or feasible, such that there is no input for which the chain will be taken. This paper examines type infeasibility of call chains, which is the infeasibility caused by inherently polymorphic call sites and sometimes also due to imprecision in call graphs. The problem of determining whether a call chain is type infeasible is defined and exemplified, a key property characterizing type infeasible call chains is described, empirical results from examining the call graphs for a set of Java programs are described, and two approaches to automatically deciding the type infeasibility of a call chain due to object parameters are presented.

1. Introduction

A number of software engineering applications involve computing information along specific *call chains*, (also called *call strings*), within a program's call graph, in contrast to whole program analysis. 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. 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.

Static call chains represent a potential sequence of procedure calls, as computed by traversing the program call graph at static analysis time.

The runtime information provided by dynamic call chains is particularly useful for both profiling and debugging. Static call chains are particularly important for static program slicing, software testing, program understanding, and interprocedural analysis used in optimizing compilers. An interprocedural program slice is composed of call chains. Call chains also 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.

Because a static call chain computation is based on the program's call graph, a particular computed static call chain may not be executable, that is, the chain may be infeasible in that there is no input for which the chain will be taken. Detecting infeasible call chains is important in software applications because they can lead to undesirable consequences. 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 program understanding tool could mislead the user and cause them to waste time and resources.

One cause of an infeasible call chain is imprecision in the analysis information, collected either intraprocedurally or interprocedurally. Another cause that occurs particularly in object-oriented software is the inherently polymorphic call sites, and the resulting call graphs constructed for these programs. 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

*This work was supported in part by NSF EIA-9806525 and NSF EIA-9870370.

consideration have been developed, although it is impossible to solve the general problem of identifying all infeasible paths [4, 14].

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. At compile time, a conservative call graph can always be built, 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 [6]. Alternatively, more precise call graphs can also be constructed by precisely computing the set of potential receiver classes at each call site. Polymorphism makes determining precise type information difficult 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 is 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. Many approaches for exhaustively computing call graphs for object-oriented software have been developed [21, 15, 18, 1, 5, 9, 11, 6, 16, 12, 7, 20, 22, 23].

In addition to the precision of the call graph generated for an object-oriented program, the inherently polymorphic call sites may cause some call chains to be type infeasible. In this paper, we define and motivate the type infeasibility problem with several examples illustrating different situations in which a given call chain can be type infeasible. We show how the type infeasibility problem differs from the problem of building a precise call graph, or eliminating call graph edges from a conservative call graph for object-oriented programs. 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, we sketch two algorithms for automatically detecting whether a particular call chain is type infeasible due to object parameters. While we believe this is the first investigation into type infeasible call chains, we give an overview of work in related topics, and conclude with an outline of future directions.

2. Definition and Example

A *call chain* is a tuple of call sites $(CS_1, CS_2, \dots, CS_n)$ 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 tuple. Each call site CS_i (an edge in the program call graph) can be represented by the method name of the caller, the source line number containing the call site, and the method name of the callee. 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. In this paper, we focus on static call chains.

A particular call chain 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 .

To illustrate how a type infeasible call chain can occur, we use Figure 1, which 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. Consider the call chain represented by $m-x-y_1$, 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 different sets 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 $m-x-y_1$.

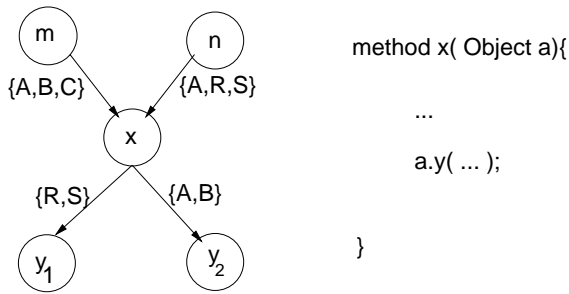


Figure 1. Infeasible call chain example.

To determine the type feasibility of call chain $m-x-y_1$, we ask the question: *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 , we find that the call sequence $m-x-y_1$ is type infeasible. Note that the call chain $m-x-y_2$ is 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 an infeasible call chain. 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

In this section, we characterize and illustrate several different situations that can cause type infeasible call sequences.

- 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 $y.z(\dots)$ 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, we use Figure 1 again. A call graph construction algorithm such as [2, 6] 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 . Notice 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 we can not determine the type of the receiver object a . 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 we were following a single call chain, say $m-x-y$, we are 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.

```

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 &&
16:            t.containsKey(key)))
17:        return false;
18:    } else {
19:      if (!value.equals(t.get(key)))
20:        return false;
21:    }
22:  }
23:  return true;

```

To illustrate, consider the `equals` method shown above from the `Hashtable` class of the Java class library. The important aspect of this code segment pertains to lines 9-18. The `while` loop is iterating through the objects stored in the `Hashtable` object that invoked 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 all instantiated classes that implement an *equals* method could be called at the call site because we do not know the types stored in the *Hashtable*. If we knew more information associated with the caller of this method (i.e., the receiver object), we might be able to determine a smaller set of objects stored in this container class. In the context of a call chain and the knowledge of a smaller set of objects stored in the container class, we could potentially determine 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.

```

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:            field1 = new myType1();

c3:        else
c4:            field1 = new myType2();

    }
}

```

This situation occurs when an object is created, and a single field of the object could be instantiated to

be of different types. The code segment above illustrates this situation. The *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 if we were 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.

4. Type Infeasibility Property

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. A type infeasible edge within a call chain can be identified by the characteristics of the call subgraph leading to the type infeasible edge.

We call the characteristics of the call subgraph leading to the type infeasible edge, the **type infeasibility property, (TIP)**. The TIP of a call subgraph *g* consists of two parts: (1) the form of the subgraph containing nodes that contribute types to a polymorphic call site 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* with the type infeasibility property must include (1) a node with at least one polymorphic call site, defined as TIP_{poly} , and (2) a node with more than one caller, TIP_{join} . For the potential presence of type infeasibility, this form of a call subgraph is necessary because multiple callers can potentially propagate different type information to the method containing the polymorphic call site. Therefore, each caller is contributing to potential receivers at the polymorphic call sites in the method represented by TIP_{poly} . Without multiple callers and a polymorphic call site, there is no possibility that a call chain could be type infeasible. Figures 2(a) and 2(b) illustrate the forms of the subgraph containing the type infeasibility property. Figure 2(a) is the most simple form of the subgraph containing the TIP property, in which the

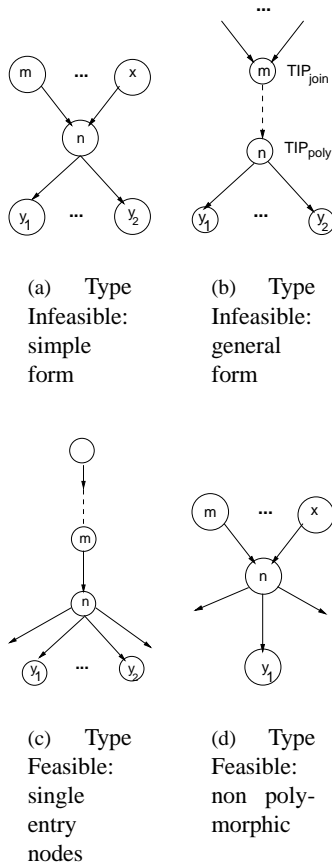


Figure 2. Type Feasibility Subgraph Forms.

same node, n in this case, is both TIP_{poly} and TIP_{join} . Figure 2(b) is the general form of the subgraph in which there are nodes between the TIP_{join} and TIP_{poly} that are single entry nodes.

Figures 2(c) and 2(d) illustrate other possible subgraph forms that may look like they could lead to type infeasible call chains, but they do not lead to them. In Figure 2(c), node n has one incoming edge, but multiple outgoing edges. Since there is only one caller of method n , and the entire call chain consists of single entry nodes, all call chains containing n must include this one incoming edge, and all of the outgoing edges of n must be feasible from all call chains containing n because the types are either propagated from one of the single callers along the call chain to method n , or generated inside method n .

Figure 2(d) illustrates the case where n has multiple callers, and possibly multiple outgoing edges, but all outgoing edges are singleton, not polymorphic. 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 are type infeasible with respect to a given call chain passing through n .

Reaching Type Relations: For a call subgraph g that has the form described above for potential type infeasibility with respect to a single call chain, the second condition for type infeasibility involves the relations between the reaching type sets along the edges involved in the subgraph g . These relations can be formulated as a data flow equation in terms of the types propagating from callers of the node representing TIP_{join} , types generated within the method represented by TIP_{poly} , and types propagating to callees of TIP_{poly} through call sites in TIP_{poly} . Figure 3 shows a segment of a call graph, for which we consider the type infeasibility property of node X with respect to the call chain $Y \dots X - W_1$. The question is: *Is the call from X to W_1 type feasible with respect to the call chain $Y \dots X - W_1$?*

The formulation of this question in terms of a data flow equation is as follows. 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 .

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[M,N-...-M] = the set of all types reaching the entry of method M , from the call site N , represented by the call graph path $N-...-M$.

In terms of the call chain $Y \dots X - W_1$ in Figure 3, the contents of these sets are:

$$\begin{aligned} REC_TYPE[W_1] &= \{Z\} \\ GEN[X, X - W_1] &= \{X, Y\} \\ IN[X, Y - \dots X] &= \{A, B, C\} \end{aligned}$$

The edge $X - W_1$ is said to be type infeasible with respect to the call chain $Y \dots X - W_1$ if:

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

That is, if

$$REC_TYPE[W_1] \notin (IN[X, Y - \dots X] \cup GEN[X, X - W_1])$$

This condition can be generalized as follows. Given a call chain C that starts at CS_{j-1} and contains a TIP_{poly} node

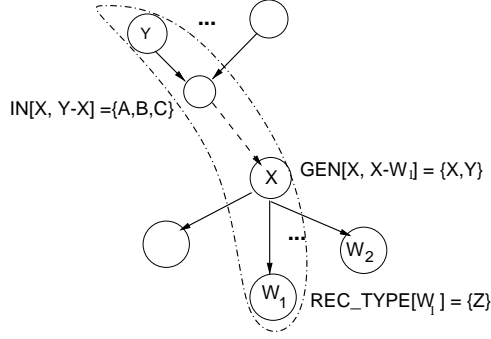


Figure 3. Type infeasibility property example.

CS_i , a TIP_{join} node CS_j , which is located before TIP_{poly} on the call chain C , the edge $CS_i - CS_{i+1}$ in C is type infeasible with respect to the call chain C , if

$$REC_TYPE[CS_{i+1}] \notin (IN[CS_i, CS_{j-1} - \dots CS_i] \cup GEN[CS_i, CS_i - CS_{i+1}])$$

This equation states that in order for an edge in a call chain to be infeasible, the receiver type that is needed to invoke the next method in the call chain must not 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} :

5. Empirical Study

We conducted an empirical study of the characteristics of call graphs for object-oriented programs, focusing on the frequency in which TIP_{poly} nodes occur as well as nodes exhibiting the simplest TIP subgraph form. The benchmarks that we studied were from the SpecJVM98 suite of programs, and we also analyzed a couple of free programs downloaded from various websites. 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 that we studied. The number of classes and methods include both user code and Java library code called from the user program.

Table 2 presents call graph characteristics for the benchmark programs using two different call graph construction algorithms, namely a conservative call graph construction algorithm, similar to RTA [2], and a more precise call graph construction algorithm, similar to CPA [1]. The table shows 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 which exhibit the simple form of a subgraph for type infeasibility (we call these nodes 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 table reveals that the percentage of simple-TIP-prone nodes is approximately 10% of the nodes in a call graph. This statistic verifies that type infeasible call chains are possible in real programs. A 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. Such nodes are necessary for type infeasible call chains to occur. 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. We believe 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. We believe 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.

6. Automatic Detection

We have developed two solutions for determining a call chain's type infeasibility caused by object parameters. As input, our algorithm expects the program call graph, the call chain $chain_{interest}$, and a control flow graph for each method (at least along $chain_{interest}$). Given $chain_{interest} = (CS_0, \dots, CS_n)$, we first traverse the call chain to create a set of all TIP-prone subchains, represented by tuples of the form $(TIP_{join-1}, TIP_{join}, TIP_{poly})$. We then identify the subset of the TIP-prone subchains that represent poten-

Name	Problem Domain	# of lines	# of byte instr	# 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

Table 1. Program characteristics.

Name	RTA				CPA (more precise)			
	# of CG nodes	% TIP-prone nodes	% poly node	% poly CS	# of CG nodes	% TIP-prone nodes	% poly node	% poly CS
compress	820	9	13	4	641	6	11	3
db	885	10	16	5	697	8	14	3
javalex	824	10	16	5	745	9	13	3
jasmin	1022	10	15	5	896	7	13	3
jack	1118	10	17	9	928	9	17	9
jess	1326	8	13	3	1173	6	10	2
soot	9582	19	39	14	8314	17	39	11

Table 2. Call graph characteristics for different call graph construction algorithms.

tial type infeasibility due to object parameters, by a simple inspection of the code for the polymorphic call site in the method represented by TIP_{poly} .

To motivate the approach, consider the call chain $X-Y-M_1$ and the code segment for method Y below. Assuming that the node corresponding to the method Y is both the TIP_{join} and TIP_{poly} node for a TIP-prone subchain, we want to answer the question: *Is it possible, in method Y , to call method M of class Z , which is the necessary type of the receiver object, a , for Z 's method M (represented by call graph node M_1) to be called?* Therefore, we want to know if type Z could reach the variable a through call chain $X-Y-M_1$.

```

1: public void Y( Obj a ) {
    ...
    i: a.M();
    ...
}

```

The overall strategy is based on a backward analysis to determine whether the receiver object at call site $Y-M_1$ can potentially be of type Z . Note that we cannot 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. We are interested in reaching type information with respect to a particular call chain, which is a subset of the reaching type information at a given program point.

We present two approaches to this problem. The first approach relies on precomputation of the IN and GEN sets defined in section 2. 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 N along edge $N-M$ must be available.

The algorithm begins at TIP_{poly} , and checks whether $Z \in GEN[Y, i]$. If this holds, then the edge is type feasible, and the algorithm reports FEASIBLE and halts. If not, then $IN[Y,X-Y]$ is checked. If $Z \notin IN[Y, X-Y]$, then the edge $Y-M_1$ is type infeasible and we report INFEASIBLE and halt. Otherwise, we need to continue the backwards traversal because the presence of Z in $IN[Y,X-Y]$ does not imply that the edge $Y-M_1$ is feasible, because Z could flow from a path to X that is not included in the call chain $chain_{interest}$. We continue the backwards traverse to the caller of X along the call chain, first checking $GEN[caller(X), callsite(X)]$, and then $IN[X,caller(X)-X]$ as above. If we reach the first node in the call chain and it is the root of the call graph, and Z is not in the GEN set, then the edge $Y-M_1$ (and call chain) is infeasible. This approach is straightforward and does not require control flow graphs since the local data flow is pre-computed with respect to each call site. The downfall of this approach is the need to have precomputed GEN sets relative to each call site for each method.

Another approach is to use demand-driven backwards

analysis based on query propagation [10]. The backwards analysis is based on queries of the form: $query = \langle type, var \rangle$. This query represents the question: May $type$ hold for the var in question? The analysis starts at the polymorphic call site CS in TIP_{poly} and in the call chain being analyzed, and proceeds backwards in the local control flow graph, and then through the control flow graphs of each node in the call chain in reverse order of the chain, until one of the following conditions is encountered:

- The query is true. The variable could be of type Z . It is determined that the type of the variable is a set that contains Z . Therefore, the edge is type feasible with respect to the call chain, and the analysis for CS can stop.
- The query is false. The type Z does not hold for the variable var . An exact type definition of the variable as a type not equal to Z is encountered that dominates all paths from the definition to CS along the call chain, making the query false. The edge is thus type infeasible with respect to the call chain, and the analysis for CS can stop.
- The start of the call chain is encountered. If we reach the start of the call chain without encountering any exact definitions, then we are unable to determine if the call chain is infeasible.

During the analysis, we need only examine the statements that can influence the receiver of an object. In particular, statements that we need to examine include:

1. `x = new Type(); //object creation site`
The type of the object x is initialized to be of $Type$, and the object is an exact type.
2. `x = o.call(...); //method call`
The return type influences the type of the object x . We can use the static return type for the potential type of x . The call can be polymorphic therefore a set of types can be returned, one for each possible callee.
3. `x = (CAST) y; //an assignment statement`
The type of x can be influenced by the type of the cast, therefore x can potentially hold the type of cast.
4. `public void mymethod(Type1 x, Type2, ...)`
The type of the variable can only widen through the entry of a method. The entry of a method may potentially lead to more specific information as we proceed up the call sequence.

The local query propagation within a given method from the call site represented by the call chain to the entry of the

method can proceed either in a flow insensitive or flow sensitive manner. The tradeoffs are precision versus analysis time. The backward traversal from callee to caller follows only the call chain for which type infeasibility is being determined. This approach avoids precomputed GEN sets for all call sites in all methods, and instead computes information on demand only as needed.

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 [21, 15, 17, 18, 1, 5, 9, 11, 6, 16, 12, 7], many of these techniques suffer by not scaling well to large programs and not having the capability of handling incomplete programs.

Grove et al. developed a parameterized framework for assessing the cost, precision, and overall impact that various call graph construction algorithms have on interprocedural optimizations [8, 13]. Call graph construction algorithms can be placed in a lattice framework, where the top of the lattice represents optimistic call graphs, G_{\top} , for example, call graphs generated through profile information. The bottom region of the lattice, G_{\perp} , represents sound call graphs ranging from very precise call graphs such as Agesen's Cartesian Product Algorithm (CPA) [1] to call graphs created through Bacon and Sweeney's RTA algorithm [2, 3]. The most optimistic sound call graph representing the greatest lower bound over all call graphs corresponding to a particular program execution is defined as G_{ideal} in the lattice framework. In general, G_{ideal} can not be statically computed. Therefore, the most precise sound call graphs will still have call sites that can not be bound to a single callee due to the inherently polymorphic nature of the call sites. Thus, while a more precise type inference and call graph construction can reduce the number of TIP_{poly} nodes, type infeasible call chains are possible as long as there are inherently polymorphic call sites in the program.

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 [21, 13, 24] 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 [10]. Their demand-driven analysis technique models a query-based system, where queries are raised to determine whether a set of (data flow) facts are part of the solution at a certain program point. To answer the query, the query is propagated in the reverse direction of the original exhaustive analysis until all necessary program points are covered. Reps et al. have also developed an approach to demand-driven analysis [19]. Their approach is modeled as a graph reachability problem. The graph representation that they use is called an exploded supergraph, which is an expansion of the program's control flow graph. One concern with this representation is that it lacks the ability to scale well to large programs.

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 [4]. Branch correlation was originally used to support compiler optimization for the elimination of redundant conditional branches. Such an analysis identifies conditional branches in a program, in which the outcome of the branch can be statically determined. By using the branch correlation analysis, the paths through the not-taken branches become infeasible (i.e., no input will cause the path to be executed), and therefore can be used to eliminate def-use pairs that were constructed through the infeasible paths. 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

We believe this is the first paper to define the problem of type infeasible call chains caused by polymorphic call sites in object-oriented programs. We have described the characteristics of type infeasible call chains and programming

situations that can lead to these chains. While our statistics from an empirical study of Java programs show that more precise call graph construction algorithms can reduce the potential for type infeasible call chains, they can still occur due to inherently polymorphic call sites, which we believe will become more prevalent with increased use of object-oriented design principles. It is important to try to identify type infeasibility in order to reduce the wasted effort in testing and debugging as well as mislead programmers with misinformation in various program understanding tools.

While we present a sketch of two algorithms for statically detecting one kind of type infeasible call chains, 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] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of European Conference for Object-Oriented Programming*, 1995.
- [2] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1996.
- [3] David Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, 1997.
- [4] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, September 1997.
- [5] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings on Programming Language Design and Implementation*, 1990.
- [6] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of European Conference for Object-Oriented Programming*, August 1995.
- [7] Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. In *Proceedings of Principles of Programming Languages*, 1998.
- [8] Greg DeFouw, David Grove, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1997.
- [9] A. Diwan, J. E. B. Moss, and K. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1996.

- [10] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.
- [11] Mary F. Fernández. Simple and Effective Link-Time Optimizations of Modula-3 Programs. In *Proceedings on Programming Language Design and Implementation*, 1995.
- [12] David Grove. The Impact of Interprocedural Class Analysis on Optimization. In *Proceedings of CASCON*, 1995.
- [13] David Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.
- [14] L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, 7(1):60–78, January 1981.
- [15] Jens Palsberg and Michael Schwartzbach. Object-Oriented Type Inference. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1991.
- [16] Hemant Pande and Barbara Ryder. Static Type Determination for C++. In *Proceedings of Sixth USENIX C++ Technical Conference*, 1994.
- [17] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois, 1994.
- [18] John Plevyak and Andrew Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1994.
- [19] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Data Flow Analysis via Graph Reachability. In *Proceedings of Principles of Programming Languages*, 1995.
- [20] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to Analysis for Java Based using Annotated Constraints. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2001.
- [21] Olin Shivers. *Control Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, 1991.
- [22] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Valle-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical Virtual Method Call Resolution for Java . In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.
- [23] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms . In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.
- [24] Jan Vitek, R. Nigel Horspool, and James Uhl. Compile-time analysis of object-oriented programs. In *Proceedings of the 4th Int. Conf. on Compiler Construction, CC'92*, Paderborn, Germany, 1992. Springer-Verlag.