Class Hierarchy Specialization*

Frank Tip and Peter F. Sweeney

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
E-mail: {tip,pfs}@watson.ibm.com

December 10, 1999

Abstract

Many class libraries are designed with an emphasis on generality and extensibility. Applications often exercise only part of a library's functionality. As a result, the objects created by an application may contain unused (user-specified or compiler-generated) members. Redundant members in objects are undesirable because they increase an application's memory usage.

We present an algorithm for *specializing* a class hierarchy with respect to its usage in a program $\mathcal{P}$. That is, the algorithm analyzes the member access patterns for $\mathcal{P}$'s variables, and creates distinct classes for variables that access different members. The algorithm addresses the inheritance mechanisms of C++ in their full generality, including multiple inheritance and virtual (shared) inheritance.

Class hierarchy specialization reduces object size, and can be viewed as a space optimization. However, execution time may also be reduced through reduced object creation or destruction time, and caching and paging effects. Class hierarchy specialization may also create new opportunities for existing optimizations such as call devirtualization and inlining. In addition, specialization may be useful in tools for software maintenance and program understanding.

1 Introduction

The development of applications has become increasingly dependent on class libraries in recent years. Class libraries contain code and data structures that are common to many applications in the form of a class hierarchy and associated methods. Libraries make programmers more productive by helping them avoid reinventing the wheel, and allowing them to concentrate on the application-specific parts of a program instead.

There is, however, a disadvantage to class library usage, which is caused by the fact that libraries are typically designed with an emphasis on generality and extensibility. An application that uses a class library often exercises only part of the library's functionality. Unfortunately, this leads to situations where the objects created by the program contain

*This is a revised and extended version of [35].

```
class S {   int m1;
            int m2;
            int m3;
};

void main(){
  S s1; S s2; S s3;
  s1.m1 = 10;
  s2.m1 = 20; s2.m2 = 30;
  s3.m1 = 40; s3.m3 = 50;
  s1 = s2;
  s2 = s3;
}
```

(a)

```
class T1 { int m1; };
class T2 :  T1 { int m2; };
class T3 :  T2 { int m3; };

void main(){
  T1 s1; T2 s2; T3 s3;
  s1.m1 = 10;
  s2.m1 = 20; s2.m2 = 30;
  s3.m1 = 40; s3.m3 = 50;
  s1 = s2;
  s2 = s3;
}
```

(b)

Figure 1: **(a)** Example program $\mathcal{P}_1$. **(b)** Result of specialization.

unused components. For example, for a member m in a given class C, it may be the case that certain C-objects never use m. We present an algorithm that *specializes* a class hierarchy with respect to its usage in a program $\mathcal{P}$. The algorithm analyzes the member access patterns for the variables in $\mathcal{P}$, and creates distinct classes for variables that access different members. The benefits of specialization can be manifold:

- The space requirements of a program are reduced at run-time, because objects no longer contain unnecessary members.
- Specialization may eliminate virtual inheritance (i.e., shared multiple inheritance) from a class hierarchy. This reduces member access time, and it may reduce object size.
- Creation and destruction of objects requires less time due to reduced object size. Time requirements may also be reduced through caching and paging effects.
- Specialization may create new opportunities for existing optimizations such as virtual function call resolution [6, 12, 4, 11, 8] and inlining.
- Specialization may be of use in program understanding and debugging tools. For example, specialization can be used as a means to suppress the displaying of unused parts of objects during a debugging session.
- Specialization may be used in tools for finding imperfections in the design of class hierarchies [27].

Since class hierarchy specialization constructs a version of a class hierarchy that is customized for a specific application, it only applicable in cases where a library is statically compiled or linked with an application. Although we expect class hierarchy specialization to be primarily of use in the context of an optimizing compiler, we present the algorithm as a source-to-source translation for the sake of illustration.

1.1 Scope of this paper

The motivation for this work is to reduce the overhead incurred by class library usage in large object-oriented applications. In this paper, we focus on the foundational aspects

of the technique and in order to prevent our definitions and algorithms from becoming too unwieldy, we will focus on a small, idealized subset of C++, which we will refer to as $\mathcal{L}$. Language $\mathcal{L}$ contains the inheritance mechanisms of C++ in their full generality, including multiple inheritance and virtual (shared) inheritance but omits a number of C++ features that would needlessly clutter the presentation of the algorithm (e.g., access rights of classes and members). A number of other language features (e.g., nested structures) are only discussed informally. This being said, our techniques are in principle applicable to realistic languages such as C++ and Java, although this would involve a major engineering effort. The syntax and semantics of $\mathcal{L}$ are very close to those of C++, and the example programs presented below have their usual meanings. For the interested reader, details of $\mathcal{L}$ are provided in Appendix A.

1.2 Motivating examples

Fig. 1(a) shows an example program $\mathcal{P}_1$, which contains three objects `s1`, `s2`, and `s3`, each of type `S`. Careful analysis of $\mathcal{P}_1$ reveals that member `m1` is accessed from all three objects, member `m2` is accessed from `s2`, and member `m3` is accessed from `s3`. In order to save space at run-time, we would like to remove `m2` from `s1` and `s3`, and `m3` from `s1` and `s2`. Note that this requires `s1`, `s2`, and `s3` to have *different types*, since objects of the same type contain the same members.

However, the types of `s1`, `s2`, and `s3` are not completely unrelated because the assignments `s1 = s2` and `s2 = s3` impose constraints on them. If `s1`, `s2`, and `s3` have three different, *unrelated* types, the compiler would report a type error in the assignments. Observe, however, that `s1`, `s2`, and `s3` need not necessarily have *exactly* the same type: in general, an assignment $x = y$ only requires that y's type be transitively derived from x's type[1]. The specialized class hierarchy of Fig. 1(b) shows how this observation can be exploited, by introducing new types `T1`, `T2`, and `T3` for `s1`, `s2`, and `s3`, respectively, and inheritance relations between these types. Note that `s1` and `s2` now contain fewer members (the number of members of `s3` remains the same) while program behavior is preserved.

Fig. 2(a) shows an example program $\mathcal{P}_2$ that will be used as a running example throughout the remainder of the paper. $\mathcal{P}_2$ has a class hierarchy with two virtual functions, `f()` and `g()`. The result of specialization is shown in Fig. 2(b), where we have used the convention that the type of variable v is represented by class $T_{var(\mathtt{v})}$. Pictorial views of the original and specialized class hierarchies are shown in Fig. 2(c) and (d), respectively. Note that the methods `A::f()`, `A::g()`, `B::g()`, and `C::f()` are dispersed over four classes $T_{var(*\mathtt{ap})}$, $T_{var(\mathtt{a})}$, $T_{var(\mathtt{b})}$, and $T_{var(\mathtt{c})}$, and that class $T_{var(*\mathtt{ap})}$ only contains a *declaration*[2] of method `g()`. Observe that the use of a common base class $T_{var(*\mathtt{ap})}$ with only virtual methods allows us to eliminate the `x` data member from `b` and `c`.

Since the size of an object is strongly compiler-dependent it is difficult to make general statements about the space savings obtained by specialization. Using the IBM `xlC` C++ compiler on the RS/6000/AIX 4.1 platform, the size of variable `a` would remain

[1] More precisely, for an assignment $x = y$, where x has type X and y has type Y, there must be exactly one X-subobject inside a Y-object [1, Section 10.2].

[2] In $\mathcal{L}$, methods only need to be defined if they are invoked. This is not the case in C++ where an object cannot contain a pure declaration of a method for which there is no overriding definition.

```
class A {
  virtual int f(){ return g(); };
  virtual int g(){ return x; };
  int x;
};
class B : A {
  virtual int g(){ return y; };
  int y;
};
class C : B {
  virtual int f(){ return g() + z; };
  int z;
};
```

```
void main(){
  A a; B b; C c;
  A *ap;
  if (...)  { ap = &a; }
  else { if (...)  { ap = &b; }
         else { ap = &c; } }
  ap->f();
}
```

(a)

```
class T_var(*ap) {
  virtual int f(){ return g(); };
  virtual int g(); /* declaration only */
};
class T_var(a) : T_var(*ap) {
  virtual int g(){ return x; };
  int x;
};
class T_var(b) : T_var(*ap) {
  virtual int g(){ return y; };
  int y;
};
class T_var(c) : T_var(b) {
  virtual int f(){ return g() + z; };
  int z;
};
```

```
void main(){
  T_var(a) a; T_var(b) b; T_var(c) c;
  T_var(*ap) *ap;
  if (...)  { ap = &a; }
  else { if (...)  { ap = &b; }
         else { ap = &c; } }
  ap->f();
}
```

(b)

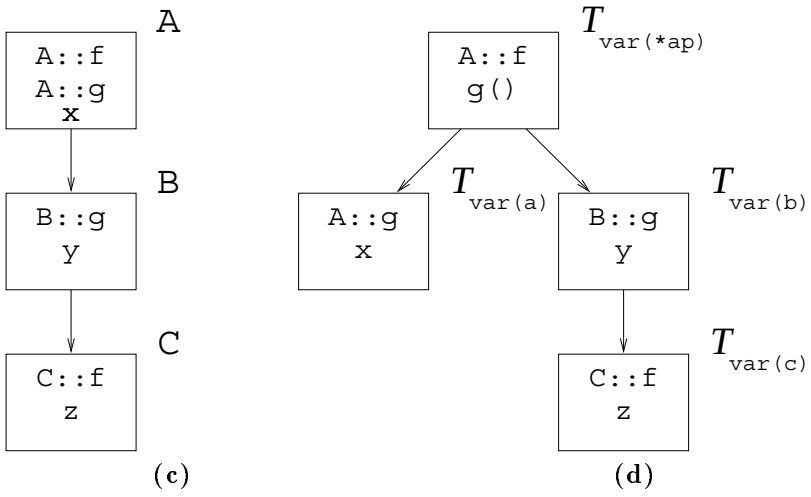


Figure 2: **(a)** Example program $\mathcal{P}_2$. **(b)** Specialized program and class hierarchy. **(c)** Pictorial view of the original class hierarchy, including the methods and fields that occur in each class. **(d)** Pictorial view of the specialized class hierarchy.

unchanged at 8 bytes, the size of `b` would be reduced from 12 to 8 bytes, and the size of `c` from 16 to 12 bytes.

We will now consider an example where class hierarchy specialization can transform virtual inheritance into nonvirtual inheritance. This is of interest because virtual inheritance requires indirections in objects that increase object size and member access time. Fig. 3(a) shows example program $\mathcal{P}_3$, which has a class hierarchy that exhibits a fairly typical use of virtual inheritance. Fig. 3(c) shows a pictorial view of this hierarchy, in which virtual inheritance relations are indicated using dashed lines. The hierarchy of $\mathcal{P}_3$ contains an "interface" class `I` that contains a declaration of method `f`, and another interface class `J` that "extends" `I` and adds a declaration for method `g`. In addition, the hierarchy contains classes `A` and `B` in which `I` and `J` are "implemented", respectively, and which contain definitions of `f` and `g`. Using the object model of the `xlC` compiler, object `a` occupies 24 bytes, and object `b` 40 bytes. Both objects contain an indirection[3] for accessing their respective `I`-subobjects.

Fig. 3(b) shows the result of specializing $\mathcal{P}_3$, Fig. 3(d) shows a pictorial view of this hierarchy. Observe that the class hierarchy no longer contains virtual inheritance. As a result of removing the indirections to shared subobjects and removing data member `x` from object `b`, both objects now occupy only 12 bytes. In addition, accessing method `f` from the pointers `ip` and `jp` no longer involves following an indirection and hence requires less time.

Program $\mathcal{P}_3$ also illustrates another benefit of specialization: it enables the transformation of virtual methods into nonvirtual methods under certain conditions. The reader may observe that the two definitions of method `f` in Fig. 3(b) are completely unrelated, since they do not have a base class in common in which `f` occurs. As a result, the virtual methods `f` in Fig. 3(b) may be transformed into nonvirtual methods without affecting program behavior. Note also that there is a single occurrence of method `g` in the hierarchy of Fig. 3(b), so that `g` can be "devirtualized" as well. Interactions between specialization and other optimizations are discussed in more detail in Section 9.

1.3 Organization of this paper

The remainder of this paper is organized as follows. Section 2 discusses related work. The next three sections closely follow the organization of the algorithm, which consists of four distinct phases. Section 3 discusses Phase I, in which basic program information is collected by inspecting the source code of input program $\mathcal{P}$. This information comprises the variables, class members, assignments, and member access operations that occur in $\mathcal{P}$, as well as pointer-alias information for pointer-typed variables and an equivalence relation on variables. Section 4 presents Phase II, which is concerned with the computation of *type constraints* that precisely capture the required subtype-relationships between the types of variables, and the visibility relations between class members and variables that must be retained in order to preserve program behavior. Section 5 addresses Phase III which constructs a new class hierarchy from the type constraints computed in Phase II. In addition, the variable declarations in the program are updated in Phase III to take the new hierarchy into account.

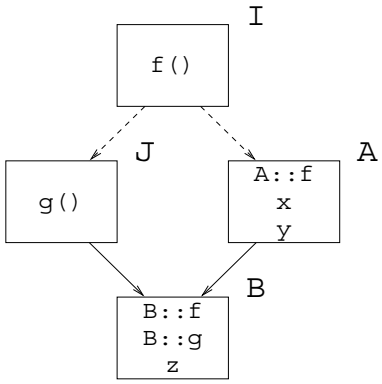
[3] In the object model used by the IBM compiler, the objects contain a pointer to the shared subobject. Other object models store such information in (virtual dispatch) tables instead of in objects.

```
class I { virtual int f(); };
class J : virtual I { virtual int g(); };
class A : virtual I {
  virtual int f(){ return x + y; };
  int x;
  int y;
};
class B : J, A {
  virtual int f(){ return y + g(); };
  virtual int g(){ return z; };
  int z;
};


void main(){
  A a;
  a.x = 10; a.y = 20;
  I *ip; ip = &a;
  int p; p = ip->f();
  B b;
  b.y = 30; b.z = 40;
  J *jp; jp = &b;
  int q; q = jp->f();
}
```

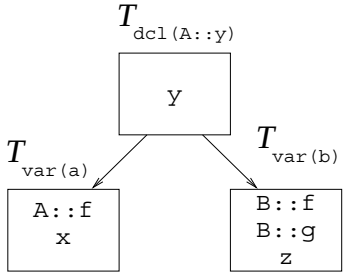
(a)

```
class T_dcl(A::y) {
  int y;
};
class T_var(a) : T_dcl(A::y) {
  virtual int f(){ return x + y; };
  int x;
};
class T_var(b) : T_dcl(A::y) {
  virtual int f(){ return y + g(); };
  virtual int g(){ return z; };
  int z;
};

void main(){
  T_var(a) a;
  a.x = 10; a.y = 20;
  T_var(a) *ip; ip = &a;
  int p; p = ip->f();
  T_var(b) b;
  b.y = 30; b.z = 40;
  T_var(b) *jp; jp = &b;
  int q; q = jp->f();
}
```

(b)



(c)



(d)

Figure 3: **(a)** Example program $\mathcal{P}_3$. **(b)** Specialized program and class hierarchy. **(c)** Pictorial view of the original class hierarchy (dashed lines indicate virtual inheritance). **(d)** Pictorial view of the specialized class hierarchy.

Section 6 discusses some of the issues that must be addressed in order to express the new class hierarchy in terms of $\mathcal{L}$'s inheritance mechanisms. In Section 7, we show that specialization is a semantics-preserving program transformation by demonstrating that the behavior of member access and type cast operations is preserved.

In the class hierarchy that results from Phase III, redundant data members and methods have been removed from objects. This hierarchy is not optimal however, since it typically exhibits an abundance of virtual inheritance. Virtual inheritance is undesirable because it is usually implemented in a way that increases member access time, and in some cases object size as well. Phase IV addresses this problem by applying a set of semantics-preserving transformation rules that simplify the specialized hierarchy, and eliminate (virtual) inheritance where possible. Section 8 discusses Phase IV.

In Section 9, we investigate how specialization interacts with a number of other program transformations/optimizations. Finally, conclusions and directions of future work are presented in Section 10.

2 Related work

2.1 Techniques for eliminating unused components from objects

The first category of related work consists of techniques for eliminating unused components from objects or class hierarchies. Tip, et al. [33] present an algorithm for *slicing* of class hierarchies that eliminates members and inheritance relations from a C++ hierarchy. In a sense, class hierarchy specialization can be viewed as a refinement of class hierarchy slicing. Like specialization, class slicing is concerned with eliminating unused members from hierarchies, but slicing can only remove a member from a class C if it is not used in *any* C-object. In contrast, specialization is capable of making finer distinctions at the variable level: By giving different types to variables that previously had the same type, members may be eliminated from certain objects while being retained in others.

In [31], Sweeney and Tip present an efficient conservative algorithm for detecting dead data members in C++ applications. In essence, this algorithm reports a data member to be dead if the program never reads that data member's value. This algorithm is evaluated on a set of C++ benchmark programs ranging from 600 to 58,000 lines of code. Sweeney and Tip found that up to 27.3% of the data members in the benchmarks are dead (average 12.5%), and that up to 11.6% of the object space of these applications may be occupied by dead data members at run-time (average 4.4%).

The algorithm of [31] is also used in the context of *Jax*, an "application extraction" tool for reducing the size of Java applications [34]. *Jax* reads in the class files that constitute a Java application and uses Rapid Type Analysis [6, 5] to determine a set of reachable methods. Then, unaccessed and write-only accessed data members are removed, and the class hierarchy transformations that will be presented in Section 8 are used to simplify the class hierarchy. After performing these transformations, JAX writes out a ZIP file containing the compressed application. In [34], *Jax* is evaluated on a number of realistic benchmark applications, and an average ZIP file size reduction of 48.7% is reported.

Agesen and Ungar [3] and Agesen [2] describe an algorithm for the dynamically typed language Self that eliminates unused slots from objects (a slot corresponds to either a

data member, a method, or an inheritance relation). This work relies on a type inference algorithm to compute for each message send that may be executed, a set of slots that is needed to preserve that send's behavior, and produces a source file in which redundant slots have been eliminated. Comparing Agesen and Ungar's work to ours is difficult due to the differences between Self and C++. Much of the complexity of our approach is due to the fact that removing members from objects requires changing the class hierarchy. This issue does not come up in Self, a dynamically typed language without classes.

We consider class hierarchy specialization to be a technique that is largely complementary to techniques for eliminating unused *executable* code [6, 29, 22]. In the scenario we have in mind, unused executable code is removed from an application first, after which the class hierarchy could be specialized in order to reduce object size. The benefit of this approach is that members that are only accessed from useless code are removed from the class hierarchy altogether. A specific technique that could be used to this end is *program slicing* [36, 32], which determines the set of executable statements that may affect the values computed at some designated point(s) of interest in a program. Unnecessary statements can be removed from a program by slicing w.r.t. all output values.

2.2 Type inference algorithms

Class hierarch

formation Gathering" steps described in Section 3 to populate a table in which access and subtype relationships between variables and class members are expressed. From this table, a *concept lattice* is derived, which exposes the hidden underlying structure in these relationships. This concept lattice provides valuable insight into the design of a class hierarchy by exposing design anomalies such as unused members and variables from which no members are accessed, and by indicating situations where it may be appropriate to split a class into multiple classes because different subsets of members are accessed from different groups of objects. Snelting and Tip also describe how the concept lattice can serve as a basis for interactive class hierarchy restructuring tools.

Godin and Mili [13, 14] also use concept analysis for class hierarchy (re)design. The starting point in their approach is a set of interfaces of (collection) classes. A table is constructed that specifies for each interface the set of supported methods. The lattice derived from this table suggests how the design of a class hierarchy implementing these interfaces could be organized in a way that optimizes the distribution of methods over the hierarchy.

Another category of related work is that of techniques for restructuring class hierarchies for the sake of improving design, improving code reuse, and enabling reuse. Opdyke [21] and Opdyke and Johnson [20] present a number of behavior-preserving transformations on class hierarchies, which they refer to as *refactorings*. The goal of refactoring is to improve design and enable reuse by "factoring out" common abstractions. This involves steps such as the creation of new superclasses, moving around methods and classes in a hierarchy, and a number of similar steps. In Opdyke and Johnson's approach, the transformation of class hierarchies is guided by the user. In contrast, class hierarchy specialization has the opposite goal: class hierarchies are *customized* for a particular application, as opposed to being generalized for the sake of reusability and maintenance. Unlike refactoring, where the programmer determines what restructurings should take place, the restructuring operations performed by class hierarchy specialization require no programmer intervention.

Moore [18] presents a tool that automatically restructures inheritance hierarchies and refactors methods in Self programs. The goal of this restructuring is to maximize the sharing of expressions between methods, and the sharing of methods between objects in order to obtain smaller programs with improved code reuse. Since Moore is studying a dynamically typed language without explicit class definitions, a number of complex issues related to preserving the appropriate subtype-relationships between classes of objects do not arise in his setting. Another important difference between our work and Moore's is that while Moore's algorithm rearranges methods in a hierarchy, it is not capable of eliminating unused members. Moore's work can be viewed as complementary to our work and some of the techniques mentioned above (e.g., [33]) because it removes methods and expressions that are unnecessary due to duplication, as opposed to unnecessary due to being unused.

3 Phase I: Information Gathering

Phase I of the specialization algorithm consists of gathering basic information about the input program $\mathcal{P}$, which we will assume to be type-correct. This information will be used in Phase II (discussed in Section 4) to compute the set of type constraints (e.g.,

subtype-relationships between variables) that must be preserved in the specialized class hierarchy.

In the sequel, v, w, ... denote variables in $\mathcal{P}$ whose type is a class; p, q, ... denote variables[4] in $\mathcal{P}$ whose type is a pointer to a class. In addition, x, y, ... will be used to denote expressions in $\mathcal{P}$. In the definitions that follow, $\mathit{TypeOf}(\mathcal{P}, x)$ denotes the *declared* (i.e., static) type of expression x in program $\mathcal{P}$.

3.1 Variables

Definition 3.1 below defines $\mathit{ClassVars}(\mathcal{P})$ and $\mathit{ClassPtrVars}(\mathcal{P})$ as the set of all variables in $\mathcal{P}$ whose type is a class, and a pointer to a class, respectively. $\mathit{ClassPtrVars}(\mathcal{P})$ contains elements for variables that occur in declarations as well as elements for implicitly declared `this` pointers of methods. In order to distinguish between `this` pointers of different methods, the `this` pointer of method `A::f()` will be denoted by the fully qualified name of its method, i.e., `A::f`.

Definition 3.1 *Let $\mathcal{P}$ be a program. Then, we define the sets of class-typed variables and pointer-to-class-typed variables as follows:*

$$\begin{array}{l}\mathit{ClassVars}(\mathcal{P}) \triangleq \\ \quad \{\, v \mid v \textit{ is a variable in } \mathcal{P},\ \mathit{TypeOf}(\mathcal{P}, v) = C, \textit{ for some class } C \textit{ in } \mathcal{P} \,\} \\ \mathit{ClassPtrVars}(\mathcal{P}) \triangleq \\ \quad \{\, p \mid p \textit{ is a variable in } \mathcal{P},\ \mathit{TypeOf}(\mathcal{P}, *p) = C, \textit{ for some class } C \textit{ in } \mathcal{P} \,\}\end{array}$$

Example 3.2 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$$\begin{array}{lcl}\mathit{ClassVars}(\mathcal{P}_2) & = & \{\, \texttt{a}, \texttt{b}, \texttt{c} \,\} \\ \mathit{ClassPtrVars}(\mathcal{P}_2) & = & \{\, \texttt{ap}, \texttt{A::f}, \texttt{A::g}, \texttt{B::g}, \texttt{C::f} \,\}\end{array}$$

□

3.2 Class members

For a given program $\mathcal{P}$, $\mathit{Members}(\mathcal{P})$ denotes the set of unqualified names of the class members that occur in $\mathcal{P}$. In addition, the sets $\mathit{DataMembers}(\mathcal{P})$, and $\mathit{VirtualMethods}(\mathcal{P})$ contain the unqualified names of data members and virtual methods of $\mathcal{P}$, respectively. For convenience, we assume the intersection of $\mathit{DataMembers}(\mathcal{P})$ and $\mathit{VirtualMethods}(\mathcal{P})$ to be empty (if this is not the case, members can be renamed), and that there are no overloaded methods with the same name but different argument types (again, renaming is possible if this is not the case).

Example 3.3 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$$\begin{array}{lcl}\mathit{DataMembers}(\mathcal{P}_2) & = & \{\, \texttt{x}, \texttt{y}, \texttt{z} \,\} \\ \mathit{VirtualMethods}(\mathcal{P}_2) & = & \{\, \texttt{f}, \texttt{g} \,\}\end{array}$$

□

[4] We will henceforth use the word "variables" to refer to variables as well as method parameters.

3.3 Points-to analysis

We will need for each pointer-to-class-typed variable a conservative and safe[5] approximation of the set of class-typed variables that it may point to in some execution of $\mathcal{P}$. Any of several existing algorithms [10, 9, 23, 30, 26]) can be used to compute this information, and we do not make assumptions about the particular algorithm used to compute points-to information.

Points-to analysis algorithms are traditionally defined for languages without virtual dispatch, and perform an analysis of the assignments that occur in a program's call graph. In the presence of virtual dispatch, call graph construction requires that conservative approximations be made about the methods that can be reached from each virtual call site $p \rightarrow f()$. An obvious way to make such approximations is to use the points-to information associated with the receiver expression p to determine to which types of objects p can point, and to determine the definition of $f()$ that would be invoked in each case. The identification of additional methods leads to additional assignments that must be taken into account when computing points-to information. This may affect previously analyzed call sites, and iteration between the two steps (computing points-to information and resolving virtual call sites) is therefore necessary.

Definition 3.4 uses the information supplied by some points-to analysis algorithm to construct a set $\mathit{PointsTo}(\mathcal{P})$, which contains a pair $\langle p, v\rangle$ for each pointer p that may point to a class-typed variable v.

Definition 3.4 *Let $\mathcal{P}$ be a program. Then, the points-to information for $\mathcal{P}$ is defined as follows:*

$$\mathit{PointsTo}(\mathcal{P}) \triangleq \{\, \langle p, v\rangle \mid p \in \mathit{ClassPtrVars}(\mathcal{P}),\ v \in \mathit{ClassVars}(\mathcal{P}),\ p \textit{ may point to } v \,\}$$

Example 3.5 *We will use the following points-to information for program $\mathcal{P}_2$. Recall that X::f denotes the* `this` *pointer of method X::$f()$.*

$$\mathit{PointsTo}(\mathcal{P}_2) = \{\ \langle \texttt{ap}, \texttt{a}\rangle,\ \langle \texttt{ap}, \texttt{b}\rangle,\ \langle \texttt{ap}, \texttt{c}\rangle,\ \langle \texttt{A::f}, \texttt{a}\rangle,\ \langle \texttt{A::f}, \texttt{b}\rangle,\ \langle \texttt{C::f}, \texttt{c}\rangle,\ \langle \texttt{A::g}, \texttt{a}\rangle,\ \langle \texttt{B::g}, \texttt{b}\rangle,\ \langle \texttt{B::g}, \texttt{c}\rangle\ \}$$

□

Note that the following simple algorithm suffices to compute the information of Example 3.5: For each pointer p of type $*X$, assume that it may point to any object of type Y, such that (i) $Y = X$ or Y is a class transitively derived from X, and (ii) if p is the `this` pointer of a virtual method C::$m()$, no definitions of m that override C::$m()$ exist in class Y.

3.4 Assignments

Definition 3.6 below defines a set $\mathit{Assignments}(\mathcal{P})$ that contains a pair of objects $\langle x', y'\rangle$ for each assignment $x = y$ in $\mathcal{P}$ for which the types of x and y are (a pointer to) a class.

[5] It will be assumed that points-to relationships are not in conflict with the type system: if a pointer p is determined to point to a variable v with type V, then p's declared type must be V or a (transitive) base class of V.

In order to simplify the subsequent definitions, we will treat a direct[6] method call as a set of assignments between corresponding formal and actual parameters, including the `this`-parameter of instance methods. The return value of a method is treated as an additional parameter as well. For indirect[7] calls, we use points-to information to model dynamic dispatch behavior: a virtual method call $p \rightarrow f(y_1, \ldots, y_n)$ is simply treated as a set of direct calls $x.f(y_1, \ldots, y_n)$, for each $\langle p, x \rangle \in \mathit{PointsTo}(\mathcal{P})$.

Definition 3.6 *Let $\mathcal{P}$ be a program. Then, the set of assignments between variables whose type is a (pointer to a) class is defined as follows:*

$$
\begin{array}{l}
\mathit{Assignments}(\mathcal{P}) \triangleq \\
\quad \{\, \langle v, w \rangle \mid v = w \text{ occurs in } \mathcal{P},\ v,\ w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle *p, w \rangle \mid\ \ p = \&w \text{ occurs in } \mathcal{P},\ p \in \mathit{ClassPtrVars}(\mathcal{P}),\ w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle *p, *q \rangle \mid p = q \text{ occurs in } \mathcal{P},\ p,\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle *p, w \rangle \mid\ \ *p = w \text{ occurs in } \mathcal{P},\ p \in \mathit{ClassPtrVars}(\mathcal{P}),\ w \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle v, *q \rangle \mid\ \ v = *q \text{ occurs in } \mathcal{P},\ v \in \mathit{ClassVars}(\mathcal{P}),\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle *p, *q \rangle \mid *p = *q \text{ occurs in } \mathcal{P},\ p,\ q \in \mathit{ClassPtrVars}(\mathcal{P}) \,\}
\end{array}
$$

Example 3.7 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$$
\begin{array}{l}
\mathit{Assignments}(\mathcal{P}_2) = \{\ \ \langle \texttt{*ap}, \texttt{a} \rangle,\ \langle \texttt{*ap}, \texttt{b} \rangle,\ \langle \texttt{*ap}, \texttt{c} \rangle,\ \langle \texttt{*A::f}, \texttt{a} \rangle,\ \langle \texttt{*A::f}, \texttt{b} \rangle,\ \langle \texttt{*C::f}, \texttt{c} \rangle, \\
\qquad\qquad\qquad\qquad \langle \texttt{*A::g}, \texttt{a} \rangle,\ \langle \texttt{*B::g}, \texttt{b} \rangle,\ \langle \texttt{*B::g}, \texttt{c} \rangle \,\}
\end{array}
$$

Note that the last six elements in this set occur due to implicit assignments that model parameter-passing of `this` *pointers.* □

3.5 Member access operations

Definition 3.8 below defines a set $\mathit{MemberAccess}(\mathcal{P})$ of all pairs $\langle x, m \rangle$ such that m is accessed from variable x. For an *indirect* call $p \rightarrow f(y_1, \ldots, y_n)$, we also include an element $\langle x, f \rangle$ in $\mathit{MemberAccess}(\mathcal{P})$ for each $\langle p, x \rangle \in \mathit{PointsTo}(\mathcal{P})$.

Definition 3.8 *Let $\mathcal{P}$ be a program. Then, the set of member access operations in $\mathcal{P}$ is defined as follows:*

$$
\begin{array}{l}
\mathit{MemberAccess}(\mathcal{P}) \triangleq \\
\quad \{\, \langle v, m \rangle \mid\ \ v.m \text{ occurs in } \mathcal{P},\ m \in \mathit{Members}(\mathcal{P}),\ v \in \mathit{ClassVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle *p, m \rangle \mid\ \ p \rightarrow m \text{ occurs in } \mathcal{P},\ m \in \mathit{Members}(\mathcal{P}),\ p \in \mathit{ClassPtrVars}(\mathcal{P}) \,\} \cup \\
\quad \{\, \langle x, m \rangle \mid\ \ p \rightarrow m \text{ occurs in } \mathcal{P},\ m \in \mathit{VirtualMethods}(\mathcal{P}),\ \langle p, x \rangle \in \mathit{PointsTo}(\mathcal{P}) \,\}
\end{array}
$$

Example 3.9 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$$
\begin{array}{l}
\mathit{MemberAccess}(\mathcal{P}_2) = \{\ \ \langle \texttt{*A::g}, \texttt{x} \rangle,\ \langle \texttt{*B::g}, \texttt{y} \rangle,\ \langle \texttt{*C::f}, \texttt{z} \rangle,\ \langle \texttt{*A::f}, \texttt{g} \rangle,\ \langle \texttt{*C::f}, \texttt{g} \rangle, \\
\qquad\qquad\qquad\qquad \langle \texttt{*ap}, \texttt{f} \rangle,\ \langle \texttt{a}, \texttt{f} \rangle,\ \langle \texttt{b}, \texttt{f} \rangle,\ \langle \texttt{c}, \texttt{f} \rangle,\ \langle \texttt{a}, \texttt{g} \rangle,\ \langle \texttt{b}, \texttt{g} \rangle,\ \langle \texttt{c}, \texttt{g} \rangle \,\}
\end{array}
$$

□

[6] A *direct* method call is an invocation of a virtual method from a non-pointer typed variable.

[7] An *indirect* call is an invocation of a virtual method from a pointer, which requires the virtual dispatch mechanism to be invoked.

3.6 An equivalence relation on variables

We now define an equivalence relation '$\sim$' on variables. Two variables occur in the same equivalence class if they must have *exactly* the same type. Definition 3.10 below states that $x \sim y$ if x is transitively assigned to y, and vice versa. Such assignments imply that the type of x must be a transitive base class of the type of y, and vice versa, and therefore that the types of x and y must be identical. The specialized class hierarchy generated in Phase III will contain a class corresponding to each equivalence class E, representing the type of the variables that occur in E.

Definition 3.10 *Let $\mathcal{P}$ be a program. Then, the equivalence relation '$\sim$' on the variables in $\mathcal{P}$ is defined as follows:*

$$x \sim y \text{ when } \langle x_0, x_1\rangle, \langle x_1, x_2\rangle, \cdots, \langle x_{m-1}, x_m\rangle, \\ \langle y_0, y_1\rangle, \langle y_1, y_2\rangle, \cdots, \langle y_{n-1}, y_n\rangle \in \mathit{Assignments}(\mathcal{P})$$

for some $x_0, \cdots, x_m, y_0, \cdots, y_n$ such that $x_0 = x$, $x_m = y$, $y_0 = y$, $y_n = x$.

Furthermore, for a given variable x, we will use $[x]$ to denote the equivalence class containing x.

Example 3.11 *For program $\mathcal{P}_2$ of Fig. 2, each variable (see Example 3.2) occurs in an equivalence class by itself.* □

In Section 6, we will extend '$\sim$' in order to prevent the occurrence of inheritance structures that cannot be represented using the inheritance mechanisms of C++.

4 Phase II: Computing Type Constraints

In Phase II of the specialization algorithm, a set of *type constraints* is determined. These constraints precisely characterize the subtype-relationships that must be preserved in the specialized class hierarchy.

4.1 Member lookup and subobject graphs

The subsequent definitions of type constraints must precisely reflect the semantics of member lookup. In the presence of multiple inheritance, an object may contain multiple subobjects of a given type C, and hence multiple members $C::m$. In order to distinguish correctly between subobjects and members with the same name, we need to keep track of the subobjects that are selected by executing member lookup and type cast operations. To this end, we use Rossie and Friedman's formalization of subobject graphs and member lookup [25, 33]. We will only introduce the notions essential for performing class hierarchy specialization here, and refer the reader to [33] for details. An efficient member lookup algorithm can be found in [24].

A subobject graph abstractly represents object layout. The subobject graph contains a distinct subgraph for each type in the class hierarchy; in what follows, we will ignore the distinction between the entire subobject graph (representing the layout of all objects), and the subgraph for a specific type. Fig. 4(a) depicts a class hierarchy in which a class `D` inherits nonvirtually (replicated) from classes `B` and `C`, and classes `B` and `C` both

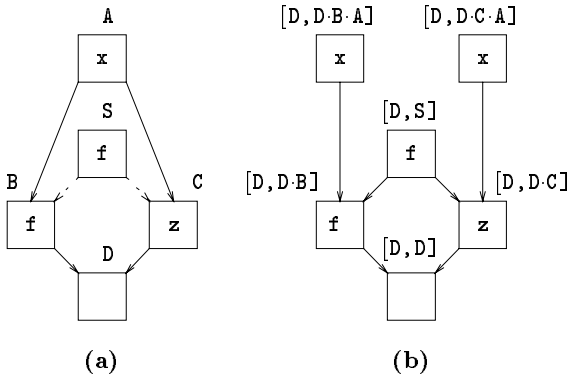


Figure 4: **(a)** Example class hierarchy graph. Solid edges indicate replicated (nonvirtual) inheritance. Dashed edges indicate virtual (i.e., shared) inheritance. **(b)** Subobject graph for type `D` in the class hierarchy of Fig. 4(a).

inherit virtually (shared) from class `S`, and nonvirtually (replicated) from class `A`. Class `A` contains a member `x`, `S` and `B` contain a member `f`, and `C` contains a member `z`.

Fig. 4(b) shows the subobject graph for `D`. The nodes in this graph are identified by a pair $[Y, X_1 \cdots X_n]$ where the first component, Y, indicates the most derived type of the subobject, and the second component is a sequence of class names, $X_1 \cdots X_n$ that encodes the sequence of inheritance relations from the least derived class X_n to its nearest virtual base class[8]. For a subobject $\sigma = [Y, X_1 \cdots X_n]$, $mdc(\sigma)$ denotes its most derived class Y, and $ldc(\sigma)$ denotes its least derived class X_n. We will say that a member m occurs in subobject σ if m occurs in its least derived class $ldc(\sigma)$. Edges in the subobject graph of Fig. 4(b) reflect the *containment* relation '$<$' between subobjects[9]. We will use '$\leq^*$' to denote the transitive and reflexive closure of '$<$'. In what follows, $\Sigma(\mathcal{P})$ denotes the set of all subobjects σ induced by the class hierarchy of program $\mathcal{P}$.

Example 4.1 *In the example of Fig. 4(b), subobject* `[D,D]` *indicates the "full"* `D` *object, and subobject* `[D,D·B]` *indicates the* `B` *subobject contained in* `[D,D]`*; in other words, we have that* `[D,D·B]` $<$ `[D,D]`*. Due to the presence of virtual inheritance,* `[D,D]` *contains a single shared* `S`*-subobject:* `[D,S]`*. By contrast, since* `B` *and* `C` *inherit nonvirtually (replicated) from* `A`*,* `[D,D]` *contains two distinct* `A`*-subobjects* `[D,D·B·A]` *and* `[D,D·C·A]`*, each containing a distinct* `x`*.*

Using the subobject graph, member lookup and type cast operations can be defined as a function from subobjects to subobjects. For example, a static lookup for a member m on a subobject σ can be expressed as follows:

$$\sigma' = \mathit{static\text{-}lookup}_{<}(\sigma, m)$$

[8] This is the minimal amount of information that is sufficient to uniquely denote a subobject.

[9] In the present paper, we define the contained subobject to be "less than" the containing subobject. We believe this notation to be more intuitive than that of [25, 33], where the contained subobject is "greater than" the containing subobject.

Here, subobject σ encodes both the static type and the run-time type of the object or pointer from which the member is accessed, and the "result" subobject σ' contains the definition of m that is selected by the lookup operation.

Static member lookup operations are used to model any member access that does not involve a dynamic dispatch, such as an access to a data member, or a call to a virtual method on a non-pointer variable. Static member lookup operations will also be used as a means to reason about the behavior of *dynamic* member lookups (corresponding to situations where a virtual method is called on a pointer). Specifically, we will model a call $p \rightarrow f()$, where $f()$ is a virtual method, using the following steps:

- A static lookup is performed to determine the method $f()$ that is statically selected. At least a declaration of $f()$ must be visible in the type of $*p$ in order for the call statement to be syntactically correct.
- Points-to information is used to determine the object v that p points to (see Definition 3.4).
- Another static lookup is performed to determine the definition of $f()$ that would be statically selected by a call $v.f()$.

The definition of $f()$ selected in the last step is the same method definition as the one selected by the dynamically dispatched call $p \rightarrow f()$.

Definition 4.2 below introduces a function *static-lookup* that models static member lookup. This function determines the "largest" subobject contained in σ that contains a definition of m. If such a unique largest subobject does not exist, the member access is ambiguous. It will be assumed that programs are type-correct, and in particular that they do not contain ambiguous member access or type cast operations.

Definition 4.2 *Let σ be a subobject in $\Sigma(\mathcal{P})$, let '$<$' be the ordering between subobjects, and let m be the name of a member. Then, the subobject that contains the accessed member m is defined as follows:*

$$\textit{static-lookup}_{<}(\sigma, m) \triangleq \max(\{\, \sigma' \mid \sigma' \leq^{*} \sigma, \textit{ldc}(\sigma') \text{ contains member } m \,\})$$

Example 4.3 *Consider a lookup* `d.z`*, where* `d` *is an object of type* `D`*. In this case, there is one definition of* `z` *in subobject* `[D,D]`*, which is located in in subobject* `[D,D·C]`*. Therefore, we have that: static-lookup*$_{<}$`([D,D],z)` = `[D,D·C]`. □

We have argued that a static member lookup operation in combination with points-to information is in principle sufficient to reason about dynamic member lookup operations. Nevertheless, it will be convenient to introduce another approach for reasoning about dynamic member lookup operations. This alternative approach relies on the fact that a subobject encodes both the static and the dynamic type of the object that a pointer points to, in the form of its least derived class and its most derived class. Definition 4.4 defines how a dynamic lookup for a member m corresponds to selecting the "largest" subobject contained in the "full object" $[\textit{mdc}(\sigma),\textit{mdc}(\sigma)]$ that contains a definition of m.

Definition 4.4 *Let σ be a subobject and let '$<$' be the ordering between subobjects. Then, the subobject that contains the dynamically accessed member m is defined as follows:*

$$\textit{dynamic-lookup}_{<}(\sigma, m) \triangleq \max(\{\sigma' \mid \sigma' \leq^{*} [\textit{mdc}(\sigma),\textit{mdc}(\sigma)], \textit{ldc}(\sigma') \text{ contains member } m\})$$

Example 4.5 *Using the class hierarchy of Fig. 4, consider a situation where we have a pointer* `cp` *of type* `C` *pointing to an object of type* `D`*, and suppose we have a virtual method call* `cp-> f()`.

This corresponds to a dynamic method lookup for `f()` *on subobject* [D,D·C]. *According to Definition 4.4, we have that:*

$$\begin{array}{l} \textit{dynamic-lookup}_{<}([\texttt{D,D·C}], \texttt{f}) = \\ \textit{dynamic-lookup}_{<}([\textit{mdc}([\texttt{D,D·C}]), \textit{mdc}([\texttt{D,D·C}])], \texttt{f}) = \\ \textit{dynamic-lookup}_{<}([\texttt{D,D}], \texttt{f}) = \\ [\texttt{D,D·B}] \end{array}$$

In other words, the method invocation `cp->f()` *will result in the invocation of* `B::f()`. □

Lemma 4.6 formally states the relationship between static and dynamic lookup operations that we informally discussed above.

Lemma 4.6 *Let σ be a subobject, m a member, and '$<$' an ordering among subobjects. Then, we have that:*

$$\textit{dynamic-lookup}_{<}(\sigma, m) = \textit{static-lookup}_{<}([\textit{mdc}(\sigma), \textit{mdc}(\sigma)], m)$$

Proof. Follows immediately from Definition 4.2 and Definition 4.4. □

We now turn our attention to type cast operations. For a cast from type X to type Y where Y is a transitive base class of X, the unique subobject $\sigma' \leq^* [X,X]$ such that $\textit{ldc}(\sigma') = Y$ is selected. If there is no unique σ' with least derived class Y, the type cast is ambiguous.

Definition 4.7 *Let σ be a subobject in $\Sigma(\mathcal{P})$, and let '$<$' be the ordering between subobjects. Then, the subobject selected by the type cast to class C is defined as follows:*

$$\textit{typecast}_{<}(\sigma, C) \triangleq \sigma' \text{ when } \sigma' \text{ is the unique subobject such that } \sigma' \leq^* \sigma,\ \textit{ldc}(\sigma') = C$$

Example 4.8 *Suppose that the program contains an assignment* `b = d`*, where* `b` *is of type* `B` *and* `d` *is of type* `D`*, respectively. For this assignment, the compiler generates a type cast from type* `D` *to type* `B`*. For this type cast, we have that:* $\textit{typecast}_{<}([\texttt{D,D}], \texttt{B}) = [\texttt{D,D·B}]$. *This implies that the assignment copies the* [D,D·B]*-subobject of* `d` *into* `b`. □

Section 10.2.2 will briefly discuss how down casts (cast operations where the target type is a derived class of the the type of the expression being casted) can be modeled.

We conclude the discussion of subobjects by introducing a composition operator '$\oplus$' on "compatible" subobjects (a subobject σ_1 and a subobject σ_2 are compatible if $\textit{ldc}(\sigma_1) = \textit{mdc}(\sigma_2)$. Intuitively, this operator determines a subobject $\sigma_3 \leq^* \sigma_1$ such that $\textit{ldc}(\sigma_3) = \textit{ldc}(\sigma_2)$. The definition below has two cases to reflect the fact that subobjects only record the inheritance path from a subobject's least derived class to its nearest virtual base.

Definition 4.9 *Let $\sigma_1 = [Z, Y_1 \cdots Y_m]$ and $\sigma_2 = [Y, X_1 \cdots X_n]$ be subobjects such that $Y = Y_m$. Then, the composition of σ_1 and σ_2 is defined as follows:*

$$\sigma_1 \oplus \sigma_2 \triangleq \begin{cases} [Z, Y_1 \cdots Y_{m-1} \cdot X_1 \cdots X_n] & \textit{when } Y = X_1 \\ [Z, X_1 \cdots X_n] & \textit{otherwise} \end{cases}$$

Example 4.10 *Using the example class hierarchy of Fig. 4, we have that* [D,D·B]$\oplus$[B,B·A] = [D,D·B·A] *and that* [D,D·B] $\oplus$ [B,S] = [D,S] □

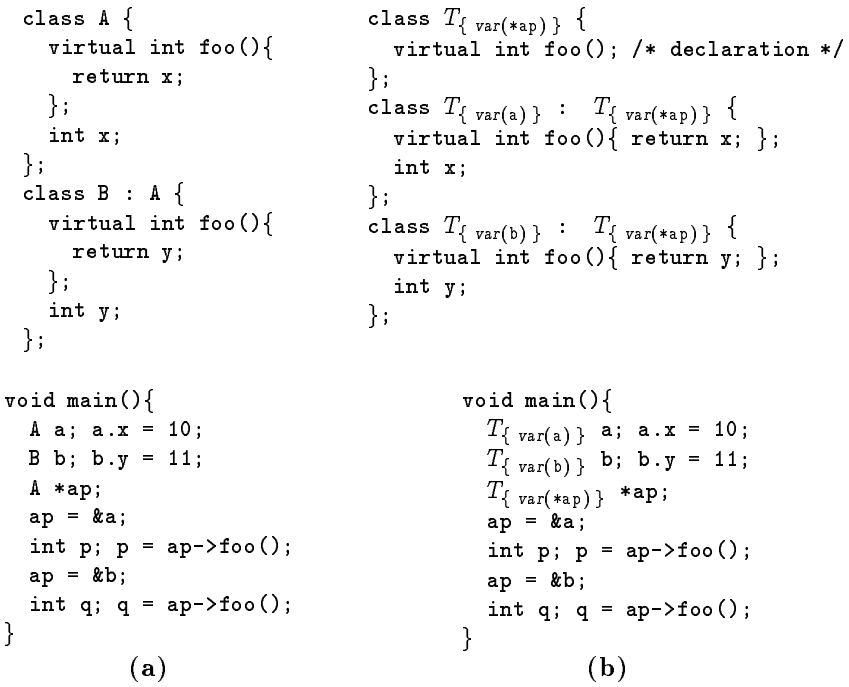


Figure 5: **(a)** Example program illustrating the purpose of distinguishing between method declarations and method definitions. **(b)** Specialized program and class hierarchy for the program of **(a)**.

4.2 Declarations vs. definitions of members

We will distinguish between declarations and definitions of members. A method's *definition* models its implementation, which has a `this` pointer from which other members may be accessed. The *declaration* of a method has the sole purpose of ensuring visibility. This distinction is important because it enables elimination of spurious dependences in the presence of virtual method calls.

Fig. 5 illustrates this issue by way of a simple program that uses two class-typed variables `a` and `b`, and a class-pointer-typed variable `ap` that points to `a` or `b`. We will now informally discuss the type constraints induced by this program, and how the distinction between declarations and definitions of methods can be exploited. For convenience, we will frequently write "member m must be visible/accessible[10] to variable x" instead of "member m must be visible/accessible from the type of variable x" in the sequel.

Clearly, the type of `*ap` must be a base class of the types of `a` and `b`. Otherwise, the assignments `ap = &a` and `ap = &b` would not be type-correct. Since virtual method `foo` is called from `ap`, a *declaration* of `foo` must be visible to `ap`. In addition, the *definition* of `A::foo()` must be visible to `a` because `ap` may point to `a`, and the *definition* of `B::foo()` must be visible to `b` because `ap` may point to `b`. Data member `x` must be visible to

[10] Since we ignore access rights of members and inheritance relations in the present paper, the notions of "visible" and "accessible" are equivalent.

`A::foo()` because it is accessed from `A::foo()`'s `this` pointer. Similarly, data member `y` must be visible to `B::foo()` because it is accessed from `B::foo()`'s `this` pointer. However, note that `A::foo()`'s *definition* need not be visible to `ap`. In fact, it is *undesirable* for `A::foo()`'s *definition* to be visible to `ap`, because that would force inclusion of `x` in `b`. The latter fact follows from the following observations in the above discussion: (i) member `x` must be visible to `A::foo()`, and (ii) that the type of `*ap` must be a base class of the type of `b`. Hence, making `A::foo()` a base class of the type of `*ap` would include `x` in `b`'s type due to the existence of a transitive inheritance relation between the type containing `x` and the type of `b`.

Fig. 5(b) shows the specialized program and class hierarchy for the example of Fig. 5(a). Note that, while the above constraints are met, `x` has been eliminated from `b`.

In the sequel, $def(A::m)$ denotes the definition of member $A::m$, whereas $dcl(A::m)$ denotes its declaration. As the example of Fig. 5 illustrates, it is useful to separate the declaration from the definition of virtual methods. Since a data member cannot access any other class members, we treat data members as if they only have declarations. (For nonvirtual methods, which are not treated in the present paper, distinguishing between declarations and definitions is not useful, and only a definition is required.)

4.3 Type constraints and constraint variables

Type constraints are of the form $\langle S, \sigma, T \rangle$, where σ is a subobject of the original class hierarchy, and S and T are sets of *constraint variables*, as defined by Definition 4.11 below.

Definition 4.11 *Let $\mathcal{P}$ be a program. Then, the set of constraint variables for $\mathcal{P}$ is defined as follows:*

$$\begin{aligned} CVars(\mathcal{P}) \triangleq\ & \{\ var(v) \mid v \in ClassVars(\mathcal{P})\ \} \cup \\ & \{\ var(*p) \mid p \in ClassPtrVars(\mathcal{P}),\ p \textit{ is not a method's } \texttt{this} \textit{ pointer}\} \cup \\ & \{\ dcl(X::m) \mid m \in Members(\mathcal{P}),\ m \textit{ occurs in class } X\ \} \cup \\ & \{\ def(X::m) \mid m \in VirtualMethods(\mathcal{P}),\ m \textit{ occurs in class } X\ \} \end{aligned}$$

Example 4.12 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$$\begin{aligned} CVars(\mathcal{P}_2) = \{\ & var(\texttt{a}),\ var(\texttt{b}),\ var(\texttt{c}),\ var(\texttt{*ap}),\ dcl(\texttt{A::x}),\ dcl(\texttt{B::y}),\ dcl(\texttt{C::z}), \\ & dcl(\texttt{A::f}),\ dcl(\texttt{A::g}),\ dcl(\texttt{B::g}),\ dcl(\texttt{C::f}),\ def(\texttt{A::f}),\ def(\texttt{A::g}), \\ & def(\texttt{B::g}),\ def(\texttt{C::f})\ \} \end{aligned}$$

□

Type constraints express subtype-relationships between constraint variables. For example, $\langle \{\ var(v)\ \}, \sigma, \{\ var(w)\ \} \rangle$ states that v has the same type as the σ-subobject of the type of w. Type constraints will also be used to express the "locations" of member declarations/definitions in objects. For example, the constraint $\langle \{\ dcl(A::m)\ \}, \sigma, \{\ var(w)\ \} \rangle$ expresses the fact that the declaration of member $A::m$ occurs in the σ-subobject of the type of w.

For reasons we will discuss shortly in Section 4.6, `this` pointers of methods require somewhat special treatment. Definition 4.13 below maps a variable v in the program to a constraint variable $var(v)$ if v is not the `this` pointer of a method, and to $def(A::m)$ if v is the `this` pointer of some method $A::m()$.

Definition 4.13 *Let x be an expression such that $x = v$ for some $v \in ClassVars(\mathcal{P})$ or $x = *p$ for some $p \in ClassPtrVars(\mathcal{P})$. Then, a constraint variable in $CVars(\mathcal{P})$ will be associated with x as follows:*

$$CVarOf(x) \triangleq \begin{cases} def(X::f) & when\ x = *X::f,\ for\ some\ method\ X::f() \\ var(x) & otherwise \end{cases}$$

Example 4.14 *For program $\mathcal{P}_2$, we have $CVarOf(\texttt{a}) = var(\texttt{a})$ and $CVarOf(\texttt{*A::f}) = def(\texttt{A::f})$.* □

The equivalence relation $\sim$ on variables of definition Definition 3.10 is now extended to constraint variables as follows.

Definition 4.15 *Let $\mathcal{P}$ be a program, and let s and t be constraint variables in $CVars(\mathcal{P})$. Then:*

$$s \sim t \text{ if and only if } s = CVarOf(x),\ t = CVarOf(y),\ x \sim y, \text{ for some variables } x,\ y$$

Furthermore, for a given constraint variable s, we will use $[s]$ to denote the equivalence class containing s.

Remark 4.16 *In order to simplify notation, we will often identify a singleton equivalence class with the element that it contains, and simply write s instead of $\{ s \}$, where s is a constraint variable.*

Note that, according to Definition 4.15, a constraint variable of the form $dcl(C::m)$ will always occur in an equivalence class by itself.

Example 4.17 *For program $\mathcal{P}_2$ of Fig. 2, each constraint variable occurs in an equivalence class by itself (see Example 3.11).* □

4.4 Type constraints due to assignments

Consider an assignment $v = w$, wh v a r i a b l e

n i q d u u b o o b il j e c t i t e T J (T & & T f (& & - D (w T j (T & T f (& & = [

Example 4.19 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$AssignTC(\mathcal{P}_2) = \{$
$\langle var(\texttt{*ap}), [\texttt{A},\texttt{A}], var(\texttt{a})\rangle$, $\langle var(\texttt{*ap}), [\texttt{B},\texttt{B}\cdot\texttt{A}], var(\texttt{b})\rangle$, $\langle var(\texttt{*ap}), [\texttt{C},\texttt{C}\cdot\texttt{B}\cdot\texttt{A}], var(\texttt{c})\rangle$,
$\langle def(\texttt{A::f}), [\texttt{A},\texttt{A}], var(\texttt{a})\rangle$, $\langle def(\texttt{A::f}), [\texttt{B},\texttt{B}\cdot\texttt{A}], var(\texttt{b})\rangle$, $\langle def(\texttt{C::f}), [\texttt{C},\texttt{C}], var(\texttt{c})\rangle$,
$\langle def(\texttt{A::g}), [\texttt{A},\texttt{A}], var(\texttt{a})\rangle$, $\langle def(\texttt{B::g}), [\texttt{B},\texttt{B}], var(\texttt{b})\rangle$, $\langle def(\texttt{B::g}), [\texttt{C},\texttt{C}\cdot\texttt{B}], var(\texttt{c})\rangle$ $\}$

Note: For the sake of readability, we have replaced all singleton equivalence classes in this example by the sole element that they contain (see Remark 4.16). □

4.5 Type constraints due to member access

Definition 4.20 below defines the set of type constraints due to member access. The definition has two cases.

The first case deals with situations where only a method declaration is needed, i.e., when the accessed member m is a data member, or a virtual method that is invoked from a pointer p. For example, consider the case where a virtual method m is accessed from a pointer p of type $*Y$. Then, there must be a unique subobject $\sigma = [Y,\alpha\cdot X] = \textit{static-lookup}_<([Y,Y], m)$ such that X contains m. Since the virtual dispatch mechanism only requires that a *declaration* of m be present in class X, a constraint $\langle [dcl(X::m)], \sigma, [var(*p)]\rangle$ is constructed, expressing the fact that the σ-subobject of $*p$ must contain a declaration of method $X::m()$.

The second case of Definition 4.20 addresses the situation where m's definition is required, i.e., when a virtual method is invoked from a nonpointer variable v. For example, suppose that a virtual method m is accessed from a variable y of type Y. Then, there must be a unique subobject $\sigma = [Y,\alpha\cdot X] = \textit{static-lookup}_<([Y,Y], m)$ such that X contains a definition of m. Consequently, a constraint $\langle [def(X::m)], \sigma, [var(y)]\rangle$ is constructed, expressing the fact that the σ-subobject of y must contain a definition of method $X::m()$.

Definition 4.20 *Let $\mathcal{P}$ be a program. Then, the set of type constraints due to member access operations is defined as follows:*

$$MemberAccessTC(\mathcal{P}) \triangleq$$
$$\left\{ \langle S,\sigma,T\rangle \;\middle|\; \begin{array}{l} \langle y,m\rangle \in MemberAccess(\mathcal{P}),\ Y = TypeOf(\mathcal{P},y), \\ (y = *p \text{ for some } p \in ClassPtrVars(\mathcal{P}) \text{ or } m \in DataMembers(\mathcal{P})), \\ \sigma = [Y,\alpha\cdot X] = \textit{static-lookup}_<([Y,Y],m), \\ S = [dcl(X::m)],\ T = [CVarOf(y)] \end{array} \right\} \cup$$
$$\left\{ \langle S,\sigma,T\rangle \;\middle|\; \begin{array}{l} \langle y,m\rangle \in MemberAccess(\mathcal{P}),\ Y = TypeOf(\mathcal{P},y), \\ (y = v \text{ for some } v \in ClassVars(\mathcal{P}) \text{ and } m \in VirtualMethods(\mathcal{P})), \\ \sigma = [Y,\alpha\cdot X] = \textit{static-lookup}_<([Y,Y],m), \\ S = [def(X::m)],\ T = [CVarOf(y)],\ S \neq T \end{array} \right\}$$

Example 4.21 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$MemberAccessTC(\mathcal{P}_2) = \{$
$\langle dcl(\texttt{A::x}), [\texttt{A},\texttt{A}], def(\texttt{A::g})\rangle$, $\langle dcl(\texttt{B::y}), [\texttt{B},\texttt{B}], def(\texttt{B::g})\rangle$, $\langle dcl(\texttt{C::z}), [\texttt{C},\texttt{C}], def(\texttt{C::f})\rangle$,
$\langle dcl(\texttt{A::g}), [\texttt{A},\texttt{A}], def(\texttt{A::f})\rangle$, $\langle dcl(\texttt{B::g}), [\texttt{C},\texttt{C}\cdot\texttt{B}], def(\texttt{C::f})\rangle$, $\langle dcl(\texttt{A::f}), [\texttt{A},\texttt{A}], var(\texttt{*ap})\rangle$,
$\langle def(\texttt{A::f}), [\texttt{A},\texttt{A}], var(\texttt{a})\rangle$, $\langle def(\texttt{A::f}), [\texttt{B},\texttt{B}\cdot\texttt{A}], var(\texttt{b})\rangle$, $\langle def(\texttt{C::f}), [\texttt{C},\texttt{C}], var(\texttt{c})\rangle$,
$\langle def(\texttt{A::g}), [\texttt{A},\texttt{A}], var(\texttt{a})\rangle$, $\langle def(\texttt{B::g}), [\texttt{B},\texttt{B}], var(\texttt{b})\rangle$, $\langle def(\texttt{B::g}), [\texttt{C},\texttt{C}\cdot\texttt{B}], var(\texttt{c})\rangle$ $\}$

Note: For the sake of readability, we have replaced all singleton equivalence classes in this example by the sole element that they contain (see Remark 4.16). □

4.6 Treatment of `this` pointers

We now return to the issue of modeling `this` pointers of methods. The definitions presented above were designed with the following properties in mind:

- The treatment of `this` pointers is analogous to that of other (class-typed and pointer-to-class-typed) parameters. Both are modeled as assignments between corresponding formal and actual parameters.
- Method declarations and method definitions are modeled in similar ways.
- The type of a `this` pointer is not declared explicitly, but determined by the location of the associated method in the class hierarchy[11]. Therefore, any constraint involving the `this` pointer of some method is effectively a constraint on the location in the hierarchy of that method.

We obtain the desired properties by mapping `this` pointers to constraint variables for the associated method definitions (see Definition 4.13). As a result, assignments and member access operations involving `this` pointers give rise to constraints involving the associated method definition as follows:

- Accessing a member m from the `this` pointer of a method f yields a type constraint involving the type containing (the declaration or definition of) m, and the type containing f's definition.
- Assigning the `this` pointer to a variable v (either explicitly, or via parameter-passing) yields a type constraint involving v's type and the type containing the definition of f.

For example, the access to data member `x` from `A::g()`'s `this` pointer gives rise to the type constraint $\langle[dcl(\texttt{A::x})], [\texttt{A},\texttt{A}], [def(\texttt{A::g})]\rangle$, which can be interpreted as 'the declaration of `A::x` occurs in the [`A`,`A`]-subobject of the type containing the definition of method `A::g`'.

Modeling parameter-passing of `this` pointers as assignments is consistent with the treatment of other parameters, but has the slightly odd property that identical type constraints occur in $AssignTC(\mathcal{P})$ and $MemberAccessTC(\mathcal{P})$. For example, the constraint $\langle[def(\texttt{A::f})], [\texttt{A},\texttt{A}], [var(\texttt{a})]\rangle$ occurs in both $AssignTC(\mathcal{P}_2)$ and $MemberAccessTC(\mathcal{P}_2)$ (see Examples 4.19 and 4.21). Although it is possible to eliminate this duplication of type constraints by modifying the definitions slightly, we consider the present solution to be the most consistent approach. The presence of duplicate type constraints is harmless in the sense that it does not affect the specialized class hierarchy.

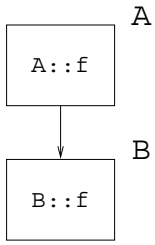
4.7 Type constraints for preserving dominance

We have now presented type constraints that express subtype-relationships between variables (Definition 4.18), and type constraints that express the visibility of members to

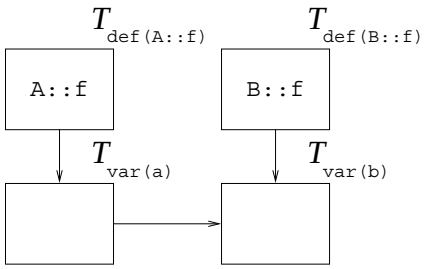
[11] Specifically, the `this` pointer of method $C::f$ has type C.

```
class A {
  virtual void f(){ ... };
};
class B {
  virtual void f(){ ... };
};

void main(){
  A a;
  a.f();
  B b;
  b.f();
  a = b;
}
```

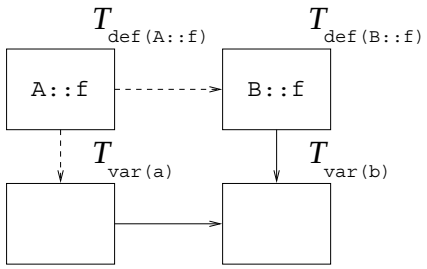
(a)



(b)



(c)



(e)

```
class A {
  virtual void f(){ ... };
};
class B {
  virtual void f(){ ... };
};

void main(){
  T var(a) a;
  a.f();
  T var(b) b;
  b.f();
  a = b;
}
```

(d)

Figure 6: **(a)** Example program. **(b)** Original class hierarchy of the program of Fig. 6(a). **(c)** Incorrect specialized class hierarchy obtained by ignoring the hiding relationships between method definitions. **(d)** Example program of Fig. 6(a) after updating variable declarations to reflect the class hierarchy of Fig. 6(c). **(e)** Correct specialized class hierarchy obtained by taking into account hiding relationships between method definitions.

variables (Definition 4.20). Together, these constraints capture all the information that is needed to construct a specialized class hierarchy in which each object contains the appropriate set of members. However, more information is needed to construct a specialized hierarchy that preserves program behavior.

Fig. 6(a) shows a simple example program, in which two objects `a` and `b` are created, with types `A` and `B`, respectively. The class hierarchy for this program is depicted in Fig. 6(b). A virtual method `f()` is defined in class `A` and overridden in class `B`. Observe that the program contains calls to method `f()` on both objects, and an assignment that copies the [B,B ·A]-subobject of object `b` into object `a`. According to Definitions 4.18 and 4.20, the following type constrains will be constructed for this program:

$$\{\ \langle \mathit{def}(\texttt{A::f}), [\texttt{A},\texttt{A}], \mathit{var}(\texttt{a})\rangle, \langle \mathit{def}(\texttt{B::f}), [\texttt{B},\texttt{B}], \mathit{var}(\texttt{b})\rangle, \langle \mathit{var}(\texttt{a}), [\texttt{B},\texttt{B}\cdot\texttt{A}], \mathit{var}(\texttt{b})\rangle\ \}$$

Fig. 6(c) shows the (incorrect) specialized class hierarchy that is obtained by simply interpreting these type constraints as subtype-relationships. Fig. 6(d) shows how the declarations of `a` and `b` in the program of Fig. 6(a) are updated to take into account the hierarchy of Fig. 6(c). Observe that method definitions $\mathit{def}(A::f)$ and $\mathit{def}(B::f)$ are *both* visible to object `b`. Since $\mathit{def}(B::f)$ does not hide or dominate $\mathit{def}(A::f)$, the call to `f()` on object `b` is ambiguous.

The ambiguity of member access `b.f()` in the above example is due to the fact that the "hiding" of method definition $\mathit{def}(A::f)$ by member definition $\mathit{def}(B::f)$ is not preserved. Our solution to this problem is to model hiding and dominance relations between members as type constraints as well. In the case of our example, a type constraint:

$$\langle \mathit{def}(\texttt{A::f}), [\texttt{B},\texttt{B}\cdot\texttt{A}], \mathit{def}(\texttt{B::f})\rangle$$

is generated to express that the type containing method definition $\mathit{def}(A::f)$ must be a base class of the type containing method definition $\mathit{def}(B::f)$. Fig. 6(e) shows the (correct) specialized class hierarchy that is constructed by taking this constraint into account. Although this example only illustrates the need for modeling hiding/dominance relations between definitions of methods, similar constraints are necessary to model hiding between definitions and declarations, and among declarations.

Definition 4.22 formally defines the set of *all* type constraints that model hiding and dominance relations between declarations and definitions of members with the same name. In Phase III, a correct specialized class hierarchy is generated by *selecting* from this set the minimal set of dominance constraints that suffices to preserve the non-ambiguity of accessed members.

Formally, Definition 4.22 states that if there are subobjects σ_1 with least derived class A and σ_2 with least derived class B such that σ_1 is contained in σ_2, and A and B both contain a declaration of member m, then a constraint $\langle [\mathit{dcl}(A::m)], \sigma, [\mathit{dcl}(B::m)]\rangle$, is constructed. Here, σ is the subobject such that $\sigma_1 = \sigma_2 \oplus \sigma$. Similar relationships are constructed for cases where A and B contain definitions of m.

Definition 4.22 *Let $\mathcal{P}$ be a program. Then, the set of type constraints that reflect the hiding/dominance relations between same-named members in the original hierarchy is*

defined as follows:

$$
\begin{aligned}
&DomTC(\mathcal{P}) \triangleq \\
&\left\{ \langle [dcl(A{::}m)], \sigma, [dcl(B{::}m)] \rangle \;\middle|\; \begin{array}{l} \sigma_1,\ \sigma_2 \text{ are subobjects in } \Sigma(\mathcal{P}),\ \sigma_1 \neq \sigma_2, \\ \sigma_1 = \sigma_2 \oplus \sigma,\ ldc(\sigma_1) = A,\ ldc(\sigma_2) = B, \\ \textit{class } A \textit{ contains a declaration of member } m, \\ \textit{class } B \textit{ contains a declaration of member } m \end{array} \right\} \cup \\
&\left\{ \langle [dcl(A{::}m)], \sigma, [def(B{::}m)] \rangle \;\middle|\; \begin{array}{l} \sigma_1,\ \sigma_2 \text{ are subobjects in } \Sigma(\mathcal{P}),\ \sigma_1 = \sigma_2 \oplus \sigma, \\ ldc(\sigma_1) = A,\ ldc(\sigma_2) = B, \\ \textit{class } A \textit{ contains a declaration of member } m, \\ \textit{class } B \textit{ contains a definition of member } m \end{array} \right\} \cup \\
&\left\{ \langle [def(A{::}m)], \sigma, [dcl(B{::}m)] \rangle \;\middle|\; \begin{array}{l} \sigma_1,\ \sigma_2 \text{ are subobjects in } \Sigma(\mathcal{P}),\ \sigma_1 \neq \sigma_2, \\ \sigma_1 = \sigma_2 \oplus \sigma,\ ldc(\sigma_1) = A,\ ldc(\sigma_2) = B, \\ \textit{class } A \textit{ contains a declaration of member } m, \\ \textit{class } B \textit{ contains a definition of member } m \end{array} \right\} \cup \\
&\left\{ \langle [def(A{::}m)], \sigma, [def(B{::}m)] \rangle \;\middle|\; \begin{array}{l} \sigma_1,\ \sigma_2 \text{ are subobjects in } \Sigma(\mathcal{P}),\ \sigma_1 \neq \sigma_2, \\ \sigma_1 = \sigma_2 \oplus \sigma,\ ldc(\sigma_1) = A,\ ldc(\sigma_2) = B, \\ \textit{class } A \textit{ contains a declaration of member } m, \\ \textit{class } B \textit{ contains a definition of member } m \} \end{array} \right\}
\end{aligned}
$$

Example 4.23 *For program $\mathcal{P}_2$ of Fig. 2, we have:*

$$
\begin{aligned}
&DomTC(\mathcal{P}_2) = \{ \\
&\quad \langle dcl(\texttt{A::f}), [\texttt{A},\texttt{A}], def(\texttt{A::f}) \rangle,\ \langle dcl(\texttt{A::f}), [\texttt{C},\texttt{C}\cdot\texttt{B}\cdot\texttt{A}], dcl(\texttt{C::f}) \rangle, \\
&\quad \langle dcl(\texttt{A::f}), [\texttt{C},\texttt{C}\cdot\texttt{B}\cdot\texttt{A}], def(\texttt{C::f}) \rangle,\ \langle def(\texttt{A::f}), [\texttt{C},\texttt{C}\cdot\texttt{B}\cdot\texttt{A}], dcl(\texttt{C::f}) \rangle, \\
&\quad \langle def(\texttt{A::f}), [\texttt{C},\texttt{C}\cdot\texttt{B}\cdot\texttt{A}], def(\texttt{C::f}) \rangle,\ \langle dcl(\texttt{C::f}), [\texttt{C},\texttt{C}], def(\texttt{C::f}) \rangle \ \} \\
&\quad \langle dcl(\texttt{A::g}), [\texttt{A},\texttt{A}], def(\texttt{A::g}) \rangle,\ \langle dcl(\texttt{A::g}), [\texttt{B},\texttt{B}\cdot\texttt{A}], dcl(\texttt{B::g}) \rangle, \\
&\quad \langle dcl(\texttt{A::g}), [\texttt{B},\texttt{B}\cdot\texttt{A}], def(\texttt{B::g}) \rangle,\ \langle def(\texttt{A::g}), [\texttt{B},\texttt{B}\cdot\texttt{A}], dcl(\texttt{B::g}) \rangle, \\
&\quad \langle def(\texttt{A::g}), [\texttt{B},\texttt{B}\cdot\texttt{A}], def(\texttt{B::g}) \rangle,\ \langle dcl(\texttt{B::g}), [\texttt{B},\texttt{B}], def(\texttt{B::g}) \rangle \ \}
\end{aligned}
$$

Note: For the sake of readability, we have replaced all singleton equivalence classes in this example by the sole element that they contain (see Remark 4.16). □

5 Phase III: Generating a Specialized Hierarchy

In Phase III, a subobject graph for the specialized class hierarchy is constructed. Then, the specialized hierarchy itself is derived from the new subobject graph, and variable declarations in the program are updated to take the new hierarchy into account.

5.1 Classes of the specialized hierarchy

The specialized hierarchy contains classes T_S, where S is an equivalence class of constraint variables, as was defined in Definition 4.15.

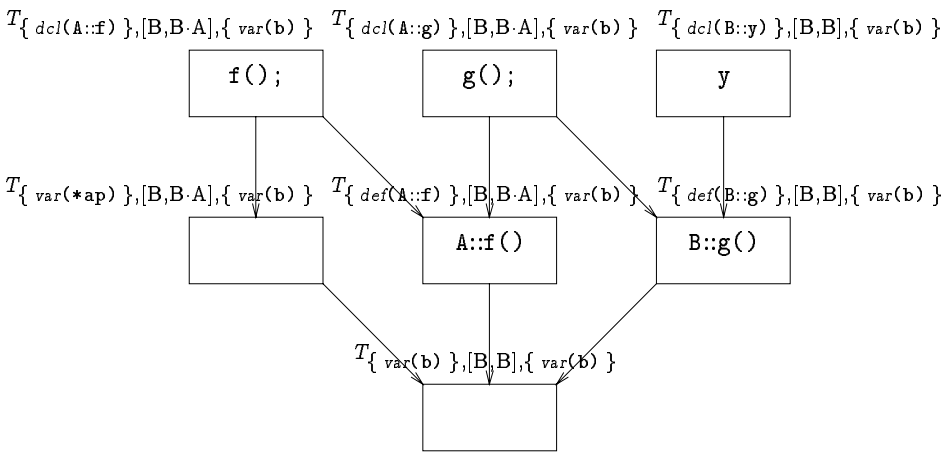


Figure 7: Specialized subobject graph for object b of example program $\mathcal{P}_2$ of Fig. 2.

Example 5.1 *For program $\mathcal{P}_2$ of Fig. 2, the specialized class hierarchy contains the following classes:*

$$\begin{aligned} NewClasses(\mathcal{P}_2) = \{\ & T_{\{\ var(\texttt{a})\ \}},\ T_{\{\ var(\texttt{b})\ \}},\ T_{\{\ var(\texttt{c})\ \}},\ T_{\{\ var(*\texttt{ap})\ \}},\ T_{\{\ dcl(\texttt{A::x})\ \}}, \\ & T_{\{\ dcl(\texttt{B::y})\ \}},\ T_{\{\ dcl(\texttt{C::z})\ \}},\ T_{\{\ dcl(\texttt{A::f})\ \}},\ T_{\{\ dcl(\texttt{A::g})\ \}},\ T_{\{\ dcl(\texttt{B::g})\ \}}, \\ & T_{\{\ dcl(\texttt{C::f})\ \}},\ T_{\{\ def(\texttt{A::f})\ \}},\ T_{\{\ def(\texttt{A::g})\ \}},\ T_{\{\ def(\texttt{B::g})\ \}},\ T_{\{\ def(\texttt{C::f})\ \}}\ \} \end{aligned}$$

□

5.2 The specialized subobject graph

Definitions 5.2 through 5.5 below together define the subobject graph $\langle N, \sqsubset \rangle$ of the specialized class hierarchy as a set of nodes N on which a containment ordering '$\sqsubset$' is defined. In the following definitions S, T, and U denote equivalence classes of constraint variables.

Definition 5.2 uses the type constraints in $AssignTC(\mathcal{P})$ and $MemberAccessTC(\mathcal{P})$ to construct the set of nodes N of the specialized subobject graph.

Definition 5.2 *Let $\mathcal{P}$ be a program. Then, the set of nodes N of the specialized subobject graph is inductively defined as follows:*

$$T_{[var(v)],\sigma,[var(v)]} \in N \quad when\ v \in ClassVars(\mathcal{P}),\ V = TypeOf(\mathcal{P}, v),\ \sigma = [V,V]$$

$$T_{Q,\sigma_2 \oplus \sigma_1,S} \in N \quad when \quad T_{R,\sigma_2,S} \in N,\ \langle Q, \sigma_1, R \rangle \in (AssignTC(\mathcal{P}) \cup MemberAccessTC(\mathcal{P}))$$

Definition 5.3 below defines the most derived class and the least derived class for nodes in N.

Definition 5.3 *Let $n = T_{Q,\sigma,S}$ be a node in N. Then, we define the most derived class $mdc(n)$ of n and the least derived class $ldc(n)$ of n as follows:*

$$\begin{array}{lcl} ldc(T_{Q,\sigma,S}) & \triangleq & T_Q \\ mdc(T_{Q,\sigma,S}) & \triangleq & T_S \end{array}$$

Definition 5.4 below defines a mapping from subobjects in the specialized class hierarchy to subobjects in the original class hierarchy.

Definition 5.4 *Let N be the set of nodes of the specialized subobject graph. Then, we define a function ψ that maps nodes in N to subobjects in the original subobject graph as follows:*

$$\psi(T_{Q,\sigma,S}) \triangleq \sigma$$

Definition 5.5 below defines a containment relation '$\sqsubset$' on subobjects in N. The '$\oplus$' operator used in this definition was introduced in Section 4.1. The $\sqsubset$-relationships (cf. subtype-relationships) between the nodes in N are determined by the constraints in $AssignTC(\mathcal{P})$, $MemberAccessTC(\mathcal{P})$, as well as those in $DomTC(\mathcal{P})$. This approach has the effect of selecting the appropriate subset of dominance relationships from $DomTC(\mathcal{P})$ needed to preserve the behavior of type casts and member lookups in $\mathcal{P}$.

Definition 5.5 *Let N be the set of nodes in the new subobject graph. Then, the containment ordering '$\sqsubset$' on subobjects in N is defined as follows: For nodes n, $n' \in N$ we have that:*

$$n \sqsubset n' \quad when \quad \left\{ \begin{array}{l} \psi(n) = \psi(n') \oplus \sigma, \\ ldc(n) = T_Q,\ ldc(n') = T_R, \\ mdc(n) = mdc(n'), \\ \langle Q, \sigma, R \rangle \in (AssignTC(\mathcal{P}) \cup MemberAccessTC(\mathcal{P}) \cup DomTC(\mathcal{P})) \end{array} \right.$$

Remark 5.6 *In principle, a $\sqsubset$-relationship that is due to constraints in $AssignTC(\mathcal{P})$ and $MemberAccessTC(\mathcal{P})$ could be constructed by modifying the inductive clause of Definition 5.2. However, as was discussed in Section 4.7, additional subtype-relationships are required in order to preserve hiding and dominance relationships between methods. Definition 5.5 provides a uniform approach for constructing all required subtype-relationships.*

Example 5.7 *Fig. 7 shows the specialized subobject graph for object* `b`. *Nodes in this graph correspond to subobjects in the specialized subobject graph, and edges in the graph reflect the '$\sqsubset$'-containment relation between nodes.*

In order to clarify the construction of the subobject graph of Fig. 7, we will study the construction of nodes $T_{\{\ \texttt{var(*ap)}\ \},[\texttt{B,B·A}],\{\ \texttt{var(b)}\ \}}$ *and* $T_{\{\ \texttt{var(b)}\ \},[\texttt{B,B}],\{\ \texttt{var(b)}\ \}}$ *in N, and the $\sqsubset$-relationship between these nodes in some detail.*

Node $T_{\{\ \texttt{var(b)}\ \},[\texttt{B,B}],\{\ \texttt{var(b)}\ \}}$ *is added to N by the first part of Definition 5.2, using $v =$* `b`, *$V =$* `B`, *and $\sigma =$* `[B,B]`. *Similarly, node* $T_{\{\ \texttt{var(*ap)}\ \},[\texttt{B,B·A}],\{\ \texttt{var(b)}\ \}}$ *is added to N by the second part of Definition 5.2, using* $T_{\{\ \texttt{var(b)}\ \},[\texttt{B,B}],\{\ \texttt{var(b)}\ \}} \in N$ *and* $\langle \{\ var(\texttt{*ap})\ \}, [\texttt{B,B·A}], \{\ var(\texttt{b})\ \} \rangle \in AssignTC(\mathcal{P}_2)$ *(see Example 4.19), such that* $Q = \{\ var(\texttt{*ap})\ \}$, $R = \{\ var(\texttt{b})\ \}$, $S = \{\ var(\texttt{b})\ \}$, $\sigma_1 =$ `[B,B·A]`, *and* $\sigma_2 =$ `[B,B]`.

The $\sqsubset$-relationship between these nodes is constructed by Definition 5.5 using $\langle \{ \mathit{var}(\texttt{*ap}) \}, [\texttt{B},\texttt{B}\cdot\texttt{A}], \{ \mathit{var}(\texttt{b}) \} \rangle \in \mathit{AssignTC}(\mathcal{P}_2)$, *where* $n = T_{\{ \mathit{var}(\texttt{*ap}) \},[\texttt{B},\texttt{B}\cdot\texttt{A}],\{ \mathit{var}(\texttt{b}) \}}$, $n' = T_{\{ \mathit{var}(\texttt{b}) \},[\texttt{B},\texttt{B}],\{ \mathit{var}(\texttt{b}) \}}$, $\mathit{ldc}(n) = T_{\{ \mathit{var}(\texttt{*ap}) \}}$, $\mathit{ldc}(n') = T_{\{ \mathit{var}(\texttt{b}) \}}$, $\mathit{mdc}(n) = \mathit{mdc}(n') = T_{\{ \mathit{var}(\texttt{b}) \}}$, $Q = \{ \mathit{var}(\texttt{*ap}) \}$, $R = \{ \mathit{var}(\texttt{b}) \}$, *and* $\sigma = [\texttt{B},\texttt{B}\cdot\texttt{A}]$.

□

5.3 The specialized class hierarchy

We are now in a position to construct the specialized class hierarchy, using the subobject graph $\langle N, \sqsubset \rangle$. Definition 5.8 defines how this hierarchy is constructed.

Definition 5.8 *The new class hierarchy contains a class T_S for each equivalence class of constraint variables S (see Definitions 3.10 and 4.15). Class T_S contains the following members:*

- *For each $\mathit{dcl}(X::m)$ in S, class T_S contains a declaration of member m, similar to the declaration of m in class X of the original hierarchy.*
- *For each $\mathit{def}(X::m)$ in S, class T_S contains a definition of member m, similar to the definition of m in class X of the original hierarchy.*

The inheritance relations of the specialized hierarchy are constructed as follows. For two subobjects $n, n' \in N$ such that $n \sqsubset n'$, class $\mathit{ldc}(n)$ is an immediate base class of class $\mathit{ldc}(n')$. This inheritance relation is virtual if all of the following hold:

1. *there is a node $n_1 \in N$ such that $\mathit{ldc}(n_1) = \mathit{ldc}(n)$,*
2. *$n_1 \sqsubset n_2$, for some $n_2 \in N$ with $\mathit{ldc}(n_2) = \mathit{ldc}(n')$,*
3. *$n_1 \sqsubset n_3$, for some $n_3 \in N$ such that $n_3 \neq n_2$, and*
4. *$n_2 \sqsubseteq^* n_4$ and $n_3 \sqsubseteq^* n_4$, for some n_4 in N.*

Otherwise, the inheritance relation between $\mathit{ldc}(n)$ and $\mathit{ldc}(n')$ is nonvirtual.

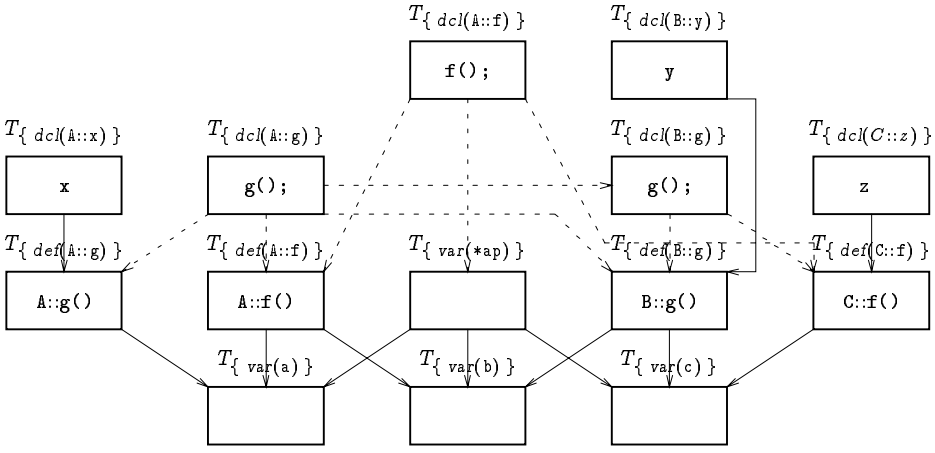
The final part of Phase III consists of updating the declarations in the program in order to reflect the new class hierarchy. This is accomplished by giving type $T_{[\mathit{var}(v)]}$ to each variable v in $\mathit{ClassVars}(\mathcal{P})$, and type $*T_{[\mathit{var}(*p)]}$ to each variable p in $\mathit{ClassPtrVars}(\mathcal{P})$ which is not the `this` pointer of a method. In C++ (and $\mathcal{L}$), `this` pointers are not declared explicitly, but the type of a `this` pointer is determined by the location of the associated method definition in the hierarchy. Hence, there are no declarations that need to be updated in this case.

Example 5.9 *Fig. 8 shows the new program and hierarchy constructed for program $\mathcal{P}_2$ of Fig. 2. The behavior of this program is identical to that of the original program, and the reader may verify that members have been eliminated from certain objects, e.g., objects* `b` *and* `c` *no longer contain member* `x`*. However, due to an abundance of virtual inheritance in the transformed hierarchy, the objects in the transformed program may have become larger than before the transformation (virtual inheritance increases member access time,*

```
class T{ dcl(A::x) } {
  int x;
};
class T{ dcl(A::f) } {
  virtual int f();
};
class T{ dcl(A::g) } {
  virtual int g();
};

class T{ dcl(B::y) } {
  int y;
};

class T{ dcl(B::g) } : virtual T{ dcl(A::g) } {
  virtual int g();
};

class T{ dcl(C::z) } {
  int z;
};

class T{ def(A::g) } : T{ dcl(A::x) },
                       virtual T{ dcl(A::g) } {
  virtual int g(){ return x; };
};

class T{ def(A::f) } : virtual T{ dcl(A::g) },
                       virtual T{ dcl(A::f) } {
  virtual int f(){ return g(); };
};

class T{ def(B::g) } : T{ dcl(B::y) },
    virtual T{ dcl(A::g) }, virtual T{ dcl(B::g) } {
  virtual int g(){ return y; };
};
```

```
class T{ def(C::f) } : T{ dcl(C::z) },
                       virtual T{ dcl(A::f) },
                       virtual T{ dcl(B::g) } {
  virtual int f(){ return g() + z; };
};

class T{ var(*ap) } : virtual T{ dcl(A::f) } { };

class T{ var(a) } :  T{ def(A::f) },
                     T{ def(A::g) },
                     T{ var(*ap) } {
};

class T{ var(b) } :  T{ def(A::f) },
                     T{ def(B::g) },
                     T{ var(*ap) } {
};

class T{ var(c) } :  T{ def(C::f) },
                     T{ def(B::g) },
                     T{ var(*ap) } {
};


void main(){
  T{ var(a) } a; T{ var(b) } b; T{ var(c) } c;
  T{ var(*ap) } *ap;
  if (...)  { ap = &a; }
  else if (...)  { ap = &b; }
  else { ap = &c; }
  ap->f();
}
```

(a)



(b)

Figure 8: **(a)** Class hierarchy and program generated by Phase III for program $\mathcal{P}_2$ of Fig. 2. **(b)** Pictorial view of the class hierarchy of Fig. 8.

and may increase object size). Using the object model of the IBM `xlC` *C++ compiler, object* `a` *now occupies 52 bytes (was 8), object* `b` *68 bytes (was 12), and object* `c` *76 bytes (was 16).* □

Phase IV of the algorithm addresses this problem by applying a set of transformation rules that simplify the class hierarchy, and reduce object size by eliminating virtual inheritance. These transformations are discussed in Section 8.

6 Representability issues

The purpose of the partitioning of variables into equivalence classes that was introduced in Definition 3.10 is to ensure that the generated class hierarchy can be expressed using the inheritance mechanisms of C++. In the absence of such a partitioning, a pair of assignments $x = y$; $y = x$ would lead to a situation where the type of x is a base class of the type of y, and the type of y is a base class of the type of x, and such cyclic inheritance class hierarchies are not valid in C++. The approach we follow, partitioning variables into equivalence classes and generating one type per equivalence class, prevents these problems.

Unfortunately, there is another situation that leads to irrepresentable inheritance structures in situations where the original class hierarchy contains classes X and Y such that a Y-object may contain multiple X-subobjects (due to multiple *nonvirtual* inheritance). Specialization may effectively transform each such X-subobject into a *shared* subobject. However, the virtual inheritance mechanism of C++ is not sufficiently powerful to model multiple, distinct shared subobjects of the same type.

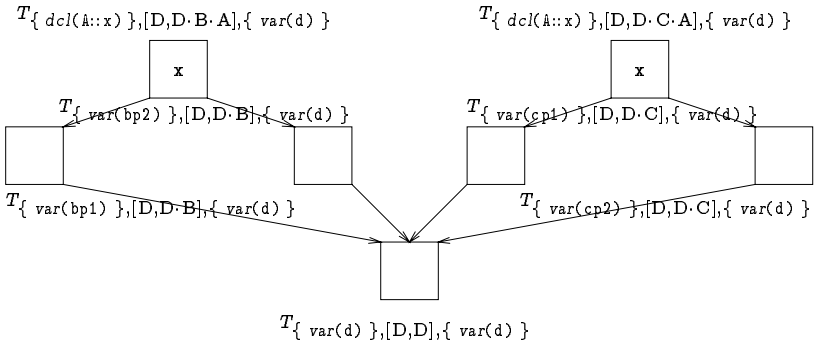
Fig. 9(a) shows a program that illustrates this situation. Note that the specialized subobject graph for this program, shown in Fig. 9(b), contains two distinct nodes $T_{\{\,dcl(\texttt{A::x})\,\},[\texttt{D,D·B·A}],\{\,var(\texttt{d})\,\}}$ and $T_{\{\,dcl(\texttt{A::x})\,\},[\texttt{D,D·C·A}],\{\,var(\texttt{d})\,\}}$ that have same least derived class: $T_{dcl(\texttt{A::x})}$. Unless countermeasures are taken, the algorithm of Section 5.3 will construct the incorrect specialized class hierarchy of Fig. 9(c). This hierarchy is incorrect because program behavior is not preserved: the program of Fig. 9(c) computes the value 50 for variable `result`, whereas the program of Fig. 9(a) computes the value 20.

The above problem only occurs in the presence of objects that contain multiple, distinct subobjects that have the same least derived class. Definition 6.1 formalizes the concept of a *replicated* class, which will be a key notion in our approach for avoiding irrepresentable inheritance structures.

Definition 6.1 *Let $\mathcal{P}$ be a program. Then, a class X in $\mathcal{P}$ is a replicated class if there is some class Y in $\mathcal{P}$ such that [Y,Y] contains multiple subobjects whose least derived class is X. We will use ReplClasses($\mathcal{P}$) to denote the set of all replicated classes in $\mathcal{P}$.*

We will use Definition 6.1 to modify the equivalence relation '$\sim$' on constraint variables in such a way that:

- The type $T_{[var(v)]}$ associated with a variable v whose type in the original hierarchy is a replicated class has at most one derived class in the specialized class hierarchy.



(b)

```
class A { int x; };
class B : A { };
class C : A { };
class D : B, C { };
void main(){
  D d;
  B *bp1;
  B *bp2;
  C *cp1;
  C *cp2;
  bp1 = &d;
  bp2 = &d;
  cp1 = &d;
  cp2 = &d;
  bp1->x = 10;
  bp2->x += 10;
  cp1->x = 40;
  cp2->x += 10;
  int result;
  result = bp1->x;
}
```

(a)

```
class T{ dcl(A::x) } { int x; };
class T{ var(*bp1) } :
  virtual T{ dcl(A::x) } { };
class T{ var(*bp2) } :
  virtual T{ dcl(A::x) } { };
class T{ var(*cp1) } :
  virtual T{ dcl(A::x) } { };
class T{ var(*cp2) } :
  virtual T{ dcl(A::x) } { };
class T{ var(d) } :
  T{ var(*bp1) }, T{ var(*bp2) },
  T{ var(*cp1) }, T{ var(*cp2) }
  { };
void main(){
  T{ var(d) } d;
  T{ var(*bp1) } *bp1;
  T{ var(*bp2) } *bp2;
  T{ var(*cp1) } *cp1;
  T{ var(*cp2) } *cp2;
  bp1 = &d; bp2 = &d;
  cp1 = &d; cp2 = &d;
  bp1->x = 10;
  bp2->x += 10;
  cp1->x = 40;
  cp2->x += 10;
  int result;
  result = bp1->x;
}
```

(c)

```
class T{ dcl(A::x) } { int x; };
class T{ var(*bp1), var(*bp2) } :
  T{ dcl(A::x) } { };
class T{ var(*cp1), var(*cp2) } :
  T{ dcl(A::x) } { };
class T{ var(d) } :
  T{ var(*bp1), var(*bp2) },
  T{ var(*cp1), var(*cp2) } { };
void main(){
  T{ var(d) } d;
  T{ var(*bp1), var(*bp2) } *bp1;
  T{ var(*bp1), var(*bp2) } *bp2;
  T{ var(*cp1), var(*cp2) } *cp1;
  T{ var(*cp1), var(*cp2) } *cp2;
  bp1 = &d; bp2 = &d;
  cp1 = &d; cp2 = &d;
  bp1->x = 10;
  bp2->x += 10;
  cp1->x = 40;
  cp2->x += 10;
  int result;
  result = bp1->x;
}
```

(d)

Figure 9: **(a)** Example program. **(b)** Specialized subobject graph (irrepresentable). **(c)** Specialized class hierarchy and program (incorrect). **(d)** Correct specialized class hierarchy and program obtained using the equivalence relation of Definition 6.2.

- The type $T_{[dcl(m)]}$ or $T_{[def(m)]}$ associated with a member m that occurs in a replicated class in the original hierarchy has at most one derived class in the specialized class hierarchy.

Since this implies that elements of the specialized subobject graph corresponding to "replicated" subobjects in the original hierarchy are no longer shared, it is guaranteed that no irrepresentable inheritance structures can occur[12]. Definition 6.2 shows the modified definition of '$\sim$'.

Definition 6.2 *Let $\mathcal{P}$ be a program. Then, the equivalence relation '$\sim$' on the variables in $\mathcal{P}$ is defined as follows:*

$$x \sim y \quad \text{when} \quad \langle x, y_1\rangle, \cdots, \langle y_n, y\rangle, \langle y, x_1\rangle, \cdots, \langle x_m, x\rangle \in \mathit{Assignments}(\mathcal{P})$$

$$x \sim y \quad \text{when} \quad \begin{cases} \langle v, x\rangle, \langle v, y\rangle \in \mathit{Assignments}(\mathcal{P}), \\ \mathit{TypeOf}(\mathcal{P}, v) \in \mathit{ReplClasses}(\mathcal{P}), \\ \mathit{TypeOf}(\mathcal{P}, x) = \mathit{TypeOf}(\mathcal{P}, y) \end{cases}$$

$$x \sim y \quad \text{when} \quad \begin{cases} \langle x, m\rangle, \langle y, m\rangle \in \mathit{MemberAccess}(\mathcal{P}), \\ X = \mathit{TypeOf}(\mathcal{P}, x) = \mathit{TypeOf}(\mathcal{P}, y), \\ \mathit{static\text{-}lookup}_{<}([X,X], m) = \mathrm{n}, \\ \mathit{ldc}(n) \in \mathit{ReplClasses}(\mathcal{P}) \end{cases}$$

for some $x_1, \cdots, x_m, y_1, \cdots, y_n$ in $(\mathit{ClassVars}(\mathcal{P}) \cup \mathit{ClassPtrVars}(\mathcal{P}))$.

The first clause of this definition is the same as before. The second clause states that if a variable v whose type is a replicated class is assigned two other variables, x and y, then the types of x and y are merged. The third clause states that if a member in a replicated class is accessed from two variables x and y, the types of these variables must be merged. The effect of the additional equivalence rules is that any replicated class in the specialized class hierarchy has no more than one derived class. As a result, such a class will never be required to be a virtual base class of another class.

This scheme is sufficient to prevent the representability problem mentioned above, provided that the following requirements are met:

- If the program contains an assignment $x = z$, and the type of x is a replicated class X, and the type of z is Z, then $X = Z$, or X is an immediate base class of Z.
- If the program contains a member access $v.m$ or $v \rightarrow m$ that statically resolves to a member m in a replicated class X, then v's type is X.

These assumptions are nonrestrictive: any $\mathcal{L}$-program that does not conforms to these assumptions can be trivially transformed into an equivalent $\mathcal{L}$-program that meets our requirements.

Returning to the example of Fig. 9(a), Fig. 9(d) shows the specialized class hierarchy and program obtained using the modified definition of '$\sim$' of Definition 6.2. Variables

[12] An alternative approach for avoiding irrepresentable structures might be to make the access to multiple shared subobjects with the same least derived class explicit, by introducing a data member that contains a pointer to the subobject under consideration.

bp1 and bp2, and variables cp1 and cp2 now occur in the same equivalence class, causing their types to be merged in the specialized hierarchy. As a result, the inheritance relation between these "merged" types, and type $T_{\{\ dcl(\texttt{A::x})\ \}}$ is now non-virtual.

Representability issues become a much more prominent issue for object-oriented languages such as Java [15] that have more limited facilities for expressing inheritance than $\mathcal{L}$. The inheritance structures that result from class hierarchy specialization are derived from the member access and assignment operations in a program, and do not conform "naturally" to a language's limitations on inheritance. For example, multiple inheritance arises naturally in the generated subobject graphs because any variable from which n members are accessed may have up to n base classes (the exact number of base classes depends on how many of these members occur in the same equivalence class). If a language for example does not support multiple inheritance, types of variables must be merged until all use of multiple inheritance is eliminated.

7 Justification

In this section, we demonstrate that class hierarchy specialization is a semantics-preserving program transformation. Since only the class hierarchy and the declarations of variables are affected by the transformation, it suffices to show that the behaviors of member lookup and type cast operations are preserved. In order to do so, we need to reason about "corresponding" subobjects in the original and specialized class hierarchy, and "corresponding" lookup and type cast operations that are performed on the original and the specialized subobject graphs. To this end, we use the ψ mapping of Section 5.2: Informally, a subobject n in N *corresponds to* a subobject $\sigma \in \Sigma(\mathcal{P})$ if $\psi(n) = \sigma$.

In order to uniformly refer to the types of variables, member declarations, and member definitions, we extend *TypeOf* to constraint variables as follows:

Definition 7.1 *Let $\mathcal{P}$ be a program, and let e be a constraint variable in $CVars(\mathcal{P})$. Then:*

$$TypeOf(\mathcal{P}, e) \triangleq \begin{cases} TypeOf(\mathcal{P}, x) & when \quad e = var(x) \\ C & when \quad e = dcl(C::m) \\ C & when \quad e = def(C::m) \end{cases}$$

Lemma 7.2 states that all constraint variables in an equivalence class have the same type.

Lemma 7.2 *Let e and f be constraint variables such that $e \sim f$. Then:*

$$TypeOf(\mathcal{P}, e) = TypeOf(\mathcal{P}, f)$$

Proof. Follows directly from Definition 6.2. □

Lemma 7.3 establishes a relationship between the types of the constraint variables in S, and the least derived class of a subobject σ, for a given type constraint $\langle S, \sigma, T \rangle$ that is due to an assignment or member access.

Lemma 7.3 *Let $\mathcal{P}$ be a program and let $\langle S, \sigma, T \rangle$ be a type constraint in $AssignTC(\mathcal{P})$ or $MemberAccessTC(\mathcal{P})$. Then, for each constraint variable e in S, we have that $ldc(\sigma) = TypeOf(\mathcal{P}, e)$.*

Proof. Follows directly from Definitions 4.18 and 4.20, and Lemma 7.2. □

Lemma 7.4 establishes a relationship between the least derived class of a subobject, and the subobject composition operator.

Lemma 7.4 *Let σ_1 and σ_2 be subobjects such that $mdc(\sigma_1) = ldc(\sigma_2)$. Then, we have that $ldc(\sigma_2 \oplus \sigma_1) = ldc(\sigma_1)$.*

Proof. Follows directly from Definition 4.9. □

Lemma 7.5 states that for any subobject n in N with least derived class $T_{[e]}$, the least derived class of $\psi(n)$ is the same as the type of e in the original class hierarchy.

Lemma 7.5 *Let n be a subobject in $\langle N, \text{`}<\text{'} \rangle$ such that $ldc(n) = T_{[e]}$. Then:*

$$ldc(\psi(n)) = TypeOf(\mathcal{P}, e)$$

Proof. This can be shown inductively, by showing that the property holds for any node added to N in Definition 5.2.

The base case consists of nodes n such that $n = T_{[\text{var}(v)],\sigma,[\text{var}(v)]}$, for some $v \in ClassVars(\mathcal{P})$, $V = TypeOf(\mathcal{P}, v)$, and $\sigma = [V,V]$. The property follows trivially for v, and from Lemma 7.2 it follows that the property holds for all elements of $[\text{var}(v)]$.

For the inductive case, assume that the property holds for a node $n' \in N$. Let n be a node that is added by the inductive clause of Definition 5.2 such that $n' = T_{T,\sigma_2,U}$, and $\langle S, \sigma_1, T \rangle \in (AssignTC(\mathcal{P}) \cup MemberAccessTC(\mathcal{P}))$. Then, we have that $n = T_{S,\sigma_2 \oplus \sigma_1,U}$. The property follows from Lemma 7.3 and Lemma 7.4. □

Lemma 7.6 *Let n and n' be subobjects in N such that $n' \sqsubset n$. Then $\psi(n') \leq^* \psi(n)$ and $mdc(n') = mdc(n)$.*

Proof. Follows directly from Definitions 5.2 and 5.5. □

Lemma 7.7 states that casting a subobject σ to its least derived class results in selection of σ itself.

Lemma 7.7 *Let σ be a subobject. Then, we have that:*

$$typecast_{<}(\sigma, ldc(\sigma)) = \sigma$$

Proof. Follows immediately from Definition 4.7. □

Theorem 7.8 states that assignment behavior is preserved. Specifically, we demonstrate that if (i) there is an assignment $\langle x, y \rangle \in Assignments(\mathcal{P})$, (ii) σ and n are corresponding subobjects in $\Sigma(\mathcal{P})$ and N, respectively, and (iii) the least derived classes of σ and n both correspond to the type of object y, then execution of the assignment will result in the selection of corresponding subobjects in $\Sigma(\mathcal{P})$ and N.

Theorem 7.8 *Let $\mathcal{P}$ be a program with initial subobject graph $\langle \Sigma(\mathcal{P}), \text{`}<\text{'} \rangle$ and specialized subobject graph $\langle N, \text{`}\sqsubset\text{'} \rangle$. Let n be a subobject in N such that $ldc(n) = T_{[CVarOf(y)]}$, and let $\langle x, y \rangle \in Assignments(\mathcal{P})$. Then:*

$$\psi(typecast_{\sqsubset}(n, T_{[CVarOf(x)]})) = typecast_{<}(\psi(n), TypeOf(\mathcal{P}, x))$$

Proof. We distinguish two cases:

1. $CVarOf(x) \sim CVarOf(y)$. We will demonstrate that the left-hand side and right-hand side of the equation reduce to the same subobject.

   For the left-hand side, we have that:

   $$\psi(typecast_{\sqsubset}(n, T_{[CVarOf(x)]})) = \psi(typecast_{\sqsubset}(n, T_{[CVarOf(y)]})) = \psi(n) \triangleq \sigma$$

   using $[CVarOf(x)] = [CVarOf(y)]$ for the first step, and $ldc(n) = T_{[CVarOf(y)]}$ and Definition 4.7 for the second step).

   For the right-hand side, we have that:

   $$typecast_{<}(\psi(n), TypeOf(\mathcal{P}, x)) = typecast_{<}(\sigma, TypeOf(\mathcal{P}, y)) = \sigma$$

   using $\sigma = \psi(n)$, $CVarOf(x) \sim CVarOf(y)$, and Lemma 7.2 for the first step. The second step relies on Lemmas 7.5 and 7.7 to demonstrate that $ldc(\sigma) = TypeOf(\mathcal{P}, y)$, and hence that $typecast_{<}(\sigma, TypeOf(\mathcal{P}, y)) = \sigma$.

2. $CVarOf(x) \not\sim CVarOf(y)$. Let $n = T_{[CVarOf(y)],\sigma_2,T}$. From $\langle x, y \rangle \in Assignments(\mathcal{P})$ and Definition 4.18, it follows that there exists a type constraint $\langle [T_{CVarOf(x)}], \sigma_1, [T_{CVarOf(y)}] \rangle$ in $AssignTC(\mathcal{P})$, for some σ_1. From Definitions 5.2 and 5.5, it follows that $n' = T_{[CVarOf(x)],\sigma_2 \oplus \sigma_1,T} \in N$, and that $n' \sqsubset n$.

   This demonstrates that n contains a subobject n' whose least derived class is of the correct type (the target type of the cast operation).

   What remains to be demonstrated is that n does not contain *another* subobject n'' with the same least derived class that would render the cast operation ambiguous. Formally speaking, we will show by contradiction that there is no n'' in N such that $n'' \sqsubseteq^* n$, $n'' \neq n'$, and $ldc(n'') = T_{[CVarOf(x)]}$. Assume there is such an n''. Then, from Lemma 7.6 it follows that $\sigma_1 \leq^* \sigma$, $ldc(\sigma_1) = X$ $\sigma_2 \leq^* \sigma$, $ldc(\sigma_2) = X$, and $\sigma_1 \neq \sigma_2$ where $X = TypeOf(\mathcal{P}, x)$, $\sigma = \psi(n)$, $\sigma_1 = \psi(n_1)$, and $\sigma_2 = \psi(n_2)$. From Definition 4.7, it follows that the type cast to type X in the original class hierarchy is ambiguous. Since we assume the program to be type-correct, this is impossible. Therefore, the property also holds in the case where $CVarOf(x) \not\sim CVarOf(y)$.

□

The following lemma is crucial in proving that the behavior of static lookup operations is preserved. Informally speaking, it states that the declaration/definition of a member m that is accessed in a lookup operation dominates all other visible declarations and definitions of m.

Lemma 7.9 *Let n_1, n_2, and n_3 be nodes in N such that $n_1 = T_{S,\sigma_1,T}$, $n_2 = T_{[\beta(Y::m)],\sigma_1 \oplus \sigma_2,T}$, and $n_3 = T_{[\alpha(X::m)],\sigma_1 \oplus \sigma_3,T}$, $n_2 \sqsubset n_1$, $n_3 \sqsubseteq^* n_1$, $\langle [\beta(Y::m)], \sigma_2, S \rangle \in MemberAccessTC(\mathcal{P})$, and α, β in $\{$ dcl, def$\}$. Then $n_3 \sqsubset n_2$.*

Proof. Let $X = TypeOf(\mathcal{P}, \alpha(X::m))$ and $Y = TypeOf(\mathcal{P}, \beta(Y::m))$. From Definition 5.8, it follows that classes X and Y both contain a declaration/definition of m.

From Lemma 7.6, $n_2 \sqsubset n_1$, and $n_3 \sqsubseteq^* n_1$, it follows that $\sigma_1 \oplus \sigma_2 \leq^* \sigma_1$ and $\sigma_1 \oplus \sigma_3 \leq^* \sigma_1$.

From the above information, Definition 4.20, and Definition 4.2, it follows that $\sigma_1 \oplus \sigma_3 \leq^* \sigma_1 \oplus \sigma_2$. It can easily be seen that this implies that $\sigma_3 \leq^* \sigma_2$.

We have now demonstrated that the occurrences of m in subobjects n_2 and n_3 are both visible in subobject n_1, and that there exists a containment relationship between corresponding subobjects in the original hierarchy, σ_2 and σ_3. Informally speaking, this containment relation implies that the m in subobject σ_2 hides or dominates the m in subobject σ_3. The dominance type constraints of Definition 4.22 were introduced to capture the appropriate hiding/dominance relations so they can be retained in the specialized subobject graph.

Formally, Definition 4.22 states that there is a constraint $\langle [\alpha(X::m)], \sigma, [\beta(Y::m)] \rangle$ in $DomTC(\mathcal{P})$, where $\sigma_2 \oplus \sigma = \sigma_3$. Hence, Definition 5.5 implies that $n_3 \sqsubset n_2$. □

Theorem 7.10 states that the behavior of static lookup operations is preserved. Informally, the theorem states that if (i) member m is accessed from object y, (ii) σ and n are corresponding subobjects in $\Sigma(\mathcal{P})$ and N, respectively, and (iii) the least derived class of σ and n correspond to the type of object y then the static lookup operation will select corresponding subobjects in $\Sigma(\mathcal{P})$ and N.

Theorem 7.10 *Let $\mathcal{P}$ be a program with initial subobject graph $\langle \Sigma(\mathcal{P}), \text{`}< \text{'} \rangle$ and specialized subobject graph $\langle N, \text{`}\sqsubset \text{'} \rangle$. Let n be a subobject in N such that $ldc(n) = T_{[CVarOf(y)]}$, and let $\langle m, y \rangle \in MemberAccess(\mathcal{P})$. Then:*

$$\psi(\textit{static-lookup}_{\sqsubset}(n, m)) = \textit{static-lookup}_{<}(\psi(n), m)$$

Proof. Let $n = T_{[CVarOf(y)],\sigma_2,T}$. There are two cases:

1. $y \in ClassPtrVars(\mathcal{P})$ or $m \in DataMembers(\mathcal{P})$. According to Definition 4.20 there is a type constraint $\langle [dcl(X::m)], \sigma_1, [CVarOf(y)] \rangle \in MemberAccessTC(\mathcal{P})$, where $Y = TypeOf(\mathcal{P}, y)$, and $\sigma_2 \oplus \sigma_1 = [Y, \alpha \cdot X] = \textit{static-lookup}_{<}([Y, Y], m)$. From Definition 5.2 and 5.5, it follows that $n' = T_{[dcl(X::m)],\sigma_2 \oplus \sigma_1,T} \in N$, and that $n' \sqsubset n$. From Lemma 7.9 it follows that for every n'' in N $ldc(n'') = T_{[\alpha(W::m)]}$ for some W and some $x \in \{ dcl, def \}$, and $n'' \sqsubseteq^* n$, we have that $n'' \sqsubseteq^* n'$. From Definition 4.2 it follows that $\textit{static-lookup}_{\sqsubset}(n, m) = \text{n}'$, and hence that $\psi(\textit{static-lookup}_{\sqsubset}(n, m)) = \textit{static-lookup}_{<}(\psi(n), m)$.

2. $y \in ClassVars(\mathcal{P})$ and $m \in VirtualMethods(\mathcal{P})$. According to Definition 4.20 there is a type constraint $\langle [def(X::m)], \sigma, [CVarOf(y)] \rangle \in MemberAccessTC(\mathcal{P})$, where $Y = TypeOf(\mathcal{P}, y)$, and $\sigma = [Y, \alpha \cdot X] = \textit{static-lookup}_{<}([Y, Y], m)$. From Definition 5.2 and 5.5, it follows that $n' = T_{[def(X::m)],\sigma_2 \oplus \sigma_1,T} \in N$, and that $n' \sqsubset n$. From Lemma 7.9 it follows that for every n'' in N $ldc(n'') = T_{[\alpha(W::m)]}$ for some W and some $x \in \{ dcl, def \}$ and $n'' \sqsubseteq^* n$, we have that $n'' \sqsubseteq^* n'$. From Definition 4.2 it follows that $\textit{static-lookup}_{\sqsubset}(n, m) = \text{n}'$, and hence that $\psi(\textit{static-lookup}_{\sqsubset}(n, m)) = \textit{static-lookup}_{<}(\psi(n), m)$.

□

Theorem 7.10 states a correspondence between static lookup operations in the original and specialized class hierarchies. However, in order to argue that program behavior is preserved, it is necessary to make a similar claim about *dynamic* member lookup

operations that arise from dynamically dispatched method calls. We first introduce another lemma.

Lemma 7.11 establishes a relationship between a subobject's most derived class, and the subobject mapping of Definition 5.4.

Lemma 7.11 *Let n be a subobject in the specialized class hierarchy. Then, we have that:*

$$\psi([mdc(n),mdc(n)]) = [mdc(\psi(n)),mdc(\psi(n))]$$

Proof. Follows directly from Definitions 5.3 and 5.4. □

Theorem 7.12 uses Lemma 7.11 to demonstrate that dynamic lookup behavior is preserved.

Theorem 7.12 *Let $\mathcal{P}$ be a program with initial subobject graph $\langle \Sigma(\mathcal{P}), \text{`}<\text{'} \rangle$ and specialized subobject graph $\langle N, \text{`}\sqsubset\text{'} \rangle$. Let n be a subobject in N such that $ldc(n) = T_{[CVarOf(y)]}$, and let $\langle m, y \rangle \in MemberAccess(\mathcal{P})$. Then:*

$$\psi(dynamic\text{-}lookup_{\sqsubset}(n, m)) = dynamic\text{-}lookup_{<}(\psi(n), m)$$

Proof. Using Lemma 4.6, we have that

$$\psi(dynamic\text{-}lookup_{\sqsubset}(n, m)) = \psi(static\text{-}lookup_{\sqsubset}([mdc(n),mdc(n)], m))$$

Using Theorem 7.10, this can be restated as:

$$static\text{-}lookup_{<}(\psi([mdc(n),mdc(n)]), m)$$

According to Lemma 7.11, this can be rewritten to:

$$static\text{-}lookup_{<}([mdc(\psi(n)),mdc(\psi(n))], m)$$

According to Definition 4.4, this is the same as:

$$dynamic\text{-}lookup_{<}(\psi(n), m)$$

□

8 Phase IV: Simplification

Phase IV of the algorithm consists of the application of a set of semantics-preserving transformation rules to the specialized class hierarchy[13]. These rules simplify the (virtual) inheritance structures of the class hierarchy in order to reduce the number of compiler-generated fields in objects, and consequently reducing member access time and/or object size. It is important to realize that the number of explicit (i.e., user-defined) members contained in each object is *not* affected by the transformations, with the exception that a member's declaration and definition may be merged.

[13] Alternatively, the set of type constraints could be simplified before the specialized class hierarchy is generated. However, since these transformations are of interest in their own right (e.g., as an optimization performed subsequent to class hierarchy slicing [33] or application extraction [34]), we have chosen to present them as general transformations that may be applied to *any* class hierarchy.

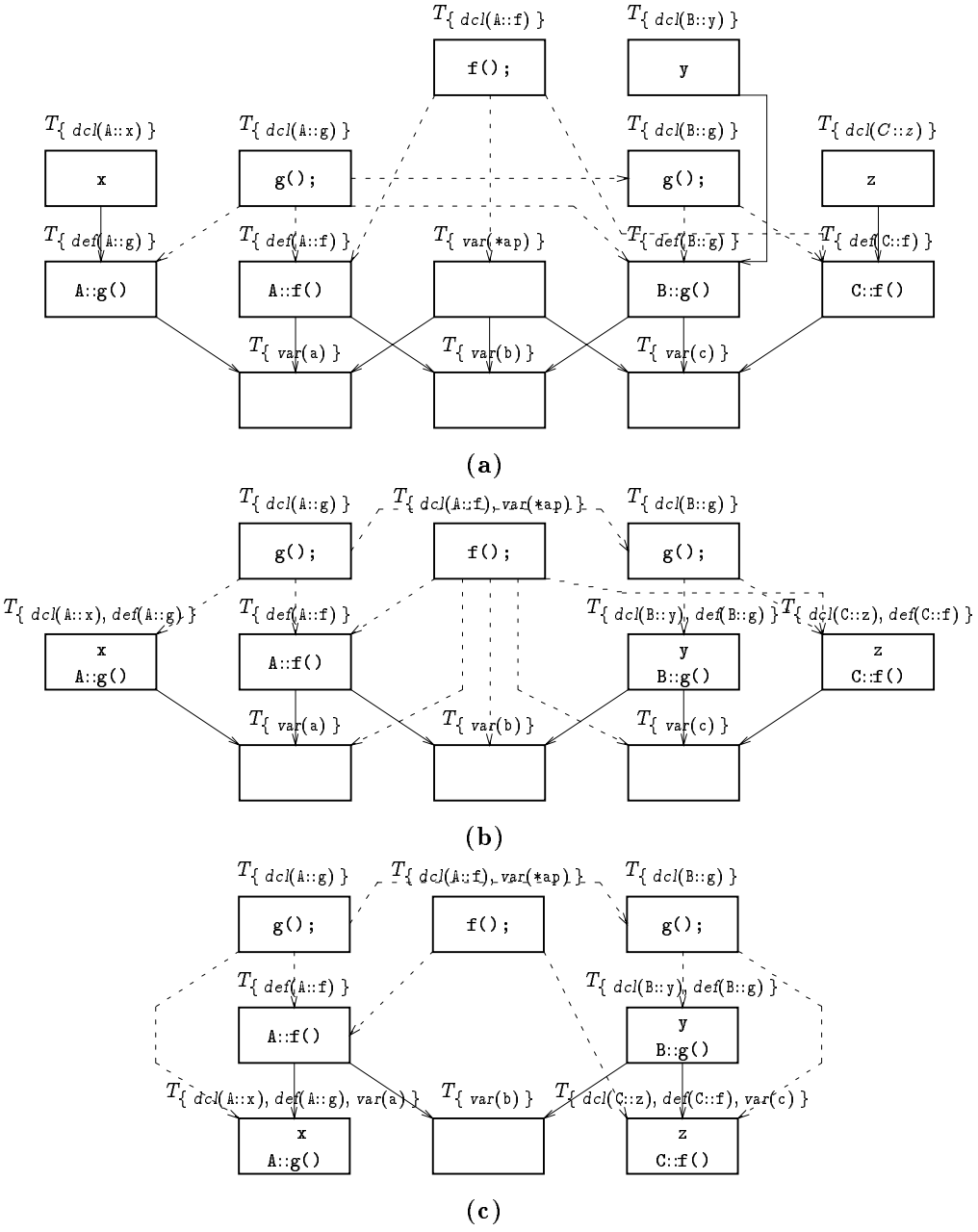


Figure 10: Illustration of the class hierarchies that result from applying the simplification rules of Section 8 to the specialized class hierarchy of Fig. 8. In the figure, boxes indicate classes, solid arrows indicate nonvirtual (replicated) inheritance, and dashed arrows indicate virtual (shared) inheritance. An unqualified member name inside a box (e.g., `f();`) indicates that a declaration of that member occurs in the class. A qualified member name (e.g., `A::g()`) indicates a member definition and the class in the original hierarchy from where it originated (`A`).

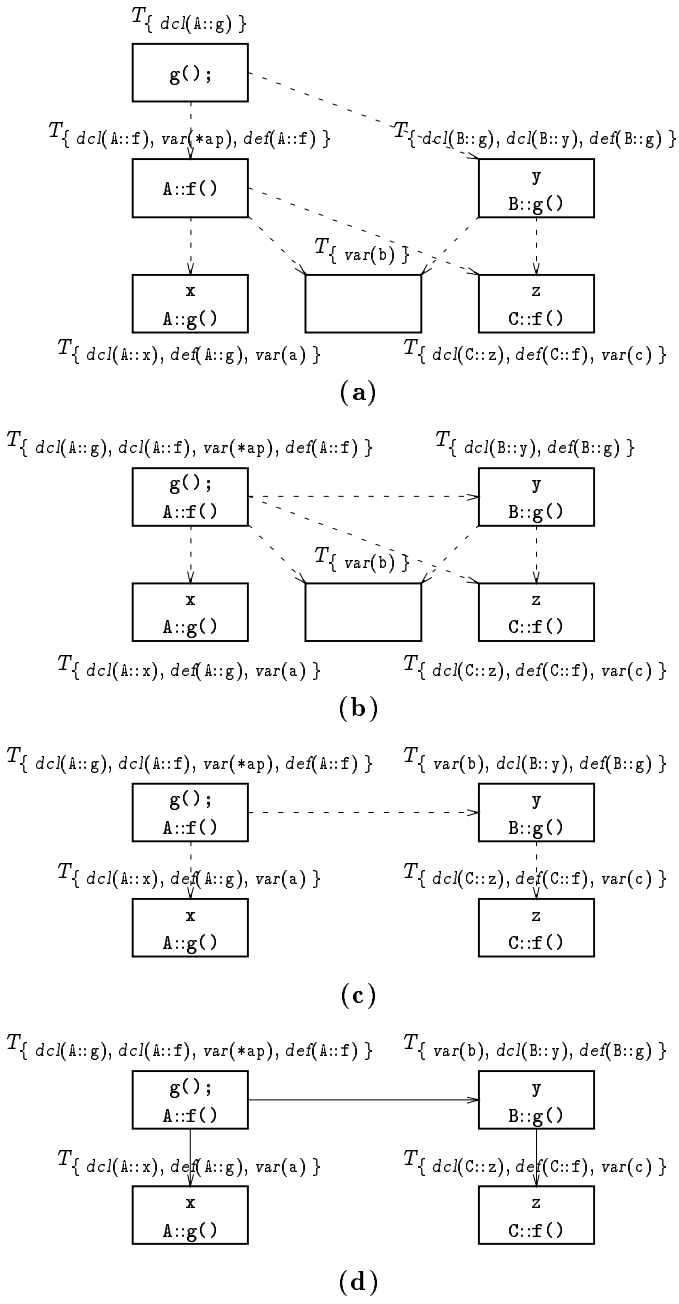


Figure 11: Illustration of the class hierarchies that result from applying the simplification rules of Section 8 to the specialized class hierarchy of Fig. 8 (continuation of Fig. 10).

8.1 The R-Rule: Removal of redundant inheritance relations

The R-Rule states that a virtual inheritance relation between classes X and Z can be removed if there exists a class Y such that:

1. X is an immediate virtual base class of Y,
2. X is an immediate virtual base class of Z, and
3. Y is a (direct or indirect) base class of Z.

8.2 The D-Rule: De-virtualizing an inheritance relation

The D-Rule[14] states that the virtual inheritance between classes X and Y can be replaced by a nonvirtual inheritance relation when:

1. X is an immediate virtual base class of Y, and
2. there is no class $Y' \neq Y$ such that (i) X is an immediate virtual base class of Y', and (ii) there is a class Z that directly or indirectly inherits from both Y and Y', and
3. there is no type W such that subobject [W,W] contains multiple, distinct subobjects with least derived class X.

8.3 The M-Rule: Merging two classes

In the description of the rule below, the "merging" of two classes X and Y (where X is a base class of Y) involves the creation of a new class Z that (virtually) inherits from each (virtual) base class of X and Y, and which contains all members of X and Y. In addition, each class Z' that inherits from X or Y is made to inherit from Z instead. This inheritance relation is virtual if the inheritance relation between X and Y or the inheritance relation between X and Z' or Y and Z' is virtual; otherwise it is nonvirtual. All variables of type X and Y are given type Z, and all variables of type $X*$ and $Y*$ are given type $Z*$. The final part of the merge operation consists of the removal of classes X and Y from the hierarchy.

The M-Rule states that we merge a base class X with a derived class Y if all of the following conditions hold:

1. X and Y have no members in common, except for the fact that for any member m, X may contain a declaration of m, and Y a definition of m.
2. There is no class Z which is a direct nonvirtual base class of both X and Y.
3. If there is a direct base class $X' \neq X$ of Y, and a direct derived class $Y' \neq Y$ of X, then X' is an indirect base class of Y'.
4. Y is not a replicated class.

[14] The original formulation of this rule in [35] contained an error.

5. If there are any variables in the program whose type is X, or any type $Y' \neq Y$ directly or indirectly derived from X, then neither Y nor any direct or indirect base class $X' \neq X$ of Y contains any data members.

6. If there are any variables in the program whose type is X, or any type $Y' \neq Y$ directly or indirectly derived from X, and if Y or any direct or indirect base class $X' \neq X$ of Y contains a declaration/definition of a virtual method, then X contains a declaration/definition of a virtual method.

Conditions (1)–(4) ensure that the class hierarchy is still valid after the merge and that member lookup behavior is preserved. Condition (5) ensures that no object becomes larger due to the addition of a data member of method as a result of the merge, and condition (6) ensures that no object becomes larger due to the addition of a virtual function table pointer[15].

8.4 Example

As an example, we will study the simplification of the specialized class hierarchy that was shown in Fig. 8.

Fig. 10(a) depicts this class hierarchy before any simplifications have been performed. In Fig. 10(b), the class hierarchy is shown after merging class $T_{\{\ dcl(\texttt{A::x})\ \}}$ with class $T_{\{\ def(\texttt{A::g})\ \}}$ (M), merging $T_{\{\ dcl(\texttt{B::y})\ \}}$ and $T_{\{\ def(\texttt{B::g})\ \}}$ (M), merging $T_{\{\ dcl(C::z)\ \}}$ and $T_{\{\ def(\texttt{C::f})\ \}}$ (M), eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::g})\ \}}$ and $T_{\{\ def(\texttt{B::g})\ \}}$ (R), and merging $T_{\{\ dcl(\texttt{A::f})\ \}}$ and $T_{\{\ var(*\texttt{ap})\ \}}$ (M). Fig. 10(c) depicts the class hierarchy after eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::f}),\ var(*\texttt{ap})\ \}}$ and $T_{\{\ var(\texttt{a})\ \}}$ (R), eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::f}),\ var(*\texttt{ap})\ \}}$ and $T_{\{\ var(\texttt{b})\ \}}$ (R), eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::f}),\ var(*\texttt{ap})\ \}}$ and $T_{\{\ var(\texttt{c})\ \}}$ (R), merging $T_{\{\ dcl(\texttt{A::x}),\ def(\texttt{A::g})\ \}}$ and $T_{\{\ var(\texttt{a})\ \}}$ (M), and merging $T_{\{\ dcl(\texttt{C::z}),\ def(\texttt{C::f})\ \}}$ and $T_{\{\ var(\texttt{c})\ \}}$ (M).

Fig. 11(a) shows the hierarchy after eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::g})\ \}}$ and $T_{\{\ dcl(\texttt{A::x}),\ def(\texttt{A::g}),\ var(\texttt{a})\ \}}$ (R), eliminating the inheritance relation between $T_{\{\ dcl(\texttt{B::g})\ \}}$ and $T_{\{\ dcl(\texttt{C::z}),\ def(\texttt{C::f}),\ var(\texttt{c})\ \}}$ (R), merging $T_{\{\ dcl(\texttt{B::g})\ \}}$ and $T_{\{\ dcl(\texttt{B::y}),\ def(\texttt{B::g})\ \}}$ (M), and merging $T_{\{\ dcl(\texttt{A::f}),\ var(*\texttt{ap})\ \}}$ and $T_{\{\ def(\texttt{A::f})\ \}}$ (M). Note that merging $T_{\{\ dcl(\texttt{A::f}),\ var(*\texttt{ap})\ \}}$ with its other derived class, $T_{\{\ dcl(\texttt{C::z}),\ def(\texttt{C::f}),\ var(\texttt{c})\ \}}$, is not permitted because that would violate condition (5) of the M-Rule. Another point to note is that, as a result of the merge, the inheritance relations between the newly created "merged" classes and their derived classes have become virtual. Fig. 11(b) shows the hierarchy after merging $T_{\{\ dcl(\texttt{A::g})\ \}}$ and $T_{\{\ dcl(\texttt{A::f}),\ var(*\texttt{ap}),\ def(\texttt{A::f})\ \}}$ (M). Fig. 11(c) shows the hierarchy after eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::g}),\ dcl(\texttt{A::f}),\ var(*\texttt{ap}),\ def(\texttt{A::f})\ \}}$ and $T_{\{\ var(\texttt{b})\ \}}$ (R), eliminating the inheritance relation between $T_{\{\ dcl(\texttt{A::g}),\ dcl(\texttt{A::f}),\ var(*\texttt{ap}),\ def(\texttt{A::f})\ \}}$ and $T_{\{\ dcl(\texttt{C::z}),\ def(\texttt{C::f}),\ var(\texttt{c})\ \}}$ (R), and merging $T_{\{\ dcl(\texttt{B::y}),\ def(\texttt{B::g})\ \}}$ and $T_{\{\ var(\texttt{b})\ \}}$ (M). The final result, shown in Fig. 11(d) is obtained by replacing all virtual inheritance relations by nonvirtual inheritance relations (three applications of the D-Rule). This is the same hierarchy that was shown earlier in Fig. 2.

[15] Condition (6) is dependent on the object model. This condition may require modification if a different object model is used.

9 Interaction with other Optimizations

Class hierarchy specialization may interact with a number of existing program optimizations and transformations in interesting ways. In Section 9.1, we discuss a number of program transformations that may improve the results when applied *before* specialization. Section 9.2 discusses optimizations that may be enabled by specialization.

9.1 Optimizations to be performed before specialization

Removing dead or useless code may improve the result of specialization. In particular, eliminating assignments and member access expressions may reduce the number of inheritance relations in the specialized hierarchy, and eliminating declarations of variables reduces the number of classes. Various techniques for eliminating useless code may be used, including elimination of unreachable methods [29], dead code elimination [17], and program slicing [36, 32].

Sometimes programmers reuse variables in order to save space. This situation is illustrated by Fig. 12(a), where variable `ap` is declared once, and used in two different, unrelated contexts—note that the second assignment to `ap` "kills" the previous value. Reusing variables may adversely affect specialization because the different "uses" of the variable access different members, and be involved in different subtype-relationships with other variables. The result of specializing the program is shown in Fig. 12(b). Note that a better result can be obtained by first "splitting" variable `ap` (see Fig. 12(c)), followed by specialization (see Fig. 12(d)).

We conclude this discussion by mentioning that, in certain cases, a better specialization result can be achieved by transforming nonvirtual methods into virtual methods. This is the case because virtual methods are more "flexible" than nonvirtual methods in the sense that the definition of the method need not be visible to the caller.

9.2 Optimizations to be performed after specialization

The example of Fig. 3 illustrates how class hierarchy specialization may enable the transformation of virtual methods into nonvirtual methods. This may in turn create opportunities for inlining methods, and various intraprocedural optimizations.

10 Conclusions and Future Work

10.1 Discussion

We have presented an algorithm that computes a new class hierarchy for a program, and updates the declarations of variables in the program accordingly. This transformation may remove unnecessary members from objects, and it may eliminate virtual (shared) inheritance (which decreases member access time, and which may decrease object size). The advantages of specialization are reduced space requirements at run-time, and reduced time requirements through the reduced cost of object creation/destruction, and indirectly through caching/paging effects. In addition, specialization may create additional opportunities for existing optimizations such as virtual function call resolution.

```
class A {
  virtual int f(){ return x; };
  int x;
  int z;
};
class B : A {
  virtual int f(){ return y; };
  int y;
};

void main(){
  A a;
  B b;
  A *ap;
  ...
  ap = &a;
  ap->z = 10;
  int p; p = ap->f();
  ...
  ap = &b;
  int q; q = ap->f();
}
```

(a)

```
class T{ var(*ap) } {
  virtual int f();
  int z; };
class T{ var(a) } : T{ var(*ap) } {
  virtual int f(){ return x; };
  int x;
  int z;
};
class T{ var(b) } : T{ var(*ap) } {
  virtual int f(){ return y; };
  int y;
};

void main(){
  T{ var(a) } a;
  T{ var(b) } b;
  T{ var(*ap) } *ap;
  ...
  ap = &a;
  ap->z = 10;
  int p; p = ap->f();
  ...
  ap = &b;
  int q; q = ap->f();
}
```

(b)

```
class A {
  virtual int f(){ return x; };
  int x;
  int z;
};
class B : A {
  virtual int f(){ return y; };
  int y;
};

void main(){
  A a;
  B b;
  A *ap1;
  ...
  ap1 = &a;
  ap1->z = 10;
  int p; p = ap1->f();
  ...
  A *ap2;
  ap2 = &b;
  int q; q = ap2->f();
}
```

(c)

```
class T{ var(a) } {
  virtual int f(){ return x; };
  int x;
  int z;
};
class T{ var(b) } {
  virtual int f(){ return y; };
  int y;
};

void main(){
  T{ var(a), var(*ap1) } a;
  T{ var(b), var(*ap2) } b;
  T{ var(a), var(*ap1) } *ap1;
  ...
  ap = &a;
  ap->z = 10;
  int p; p = ap->f();
  ...
  T{ var(b), var(*ap2) } *ap2;
  ap2 = &b;
  int q; q = ap2->f();
}
```

(d)

Figure 12: **(a)** Example program. **(b)** Specialized program class hierarchy.

Much of the complexity of the formalization of class hierarchy specialization is due to the complexity of multiple *non-virtual* inheritance. In the presence of single inheritance, and multiple *virtual* inheritance, each object can be characterized as a *set* of members, because an object always contains at most one subobject of any given type. In the presence of *non-virtual* multiple inheritance, this is no longer the case, and subobject information needs to be encoded in type constraints. Virtual inheritance does not pose many problems by itself, because even for languages with only *single* inheritance, the hierarchies generated by Phase III naturally exhibit virtual multiple inheritance. However, as we mentioned in Section 6, additional work would be involved in transforming these intermediate results into hierarchies with only single inheritance.

While we do not have empirical data of the space savings due to class hierarchy specialization, less sophisticated member elimination techniques [31, 34] have shown to be highly effective in reducing the number of class members. Specifically, [31] reports an average of 12.5% dead data members in C++ applications, and [34] reports an average of 49.7% dead fields, and an average of 34.4% dead methods in Java applications. We believe that the better results in the context of Java are due to the fact that Java applications are written in a more object-oriented style and tend to rely more on class libraries, but also because in the approach of [34] dead methods are removed prior to the transformations of the class hierarchy. Being a more precise analysis, class hierarchy specialization should produce better results. How much better the results would be in practice is a topic for future research.

10.2 Accommodating other language features

We have presented our definitions and formalism for a small object-oriented language in order to prevent our definitions from becoming too unwieldy. However, the application of class hierarchy specialization to a real language such as C++ or Java requires that a number of additional language features be modeled.

10.2.1 Nested structures

Nested structures arise when the type of a data member is a class, or a pointer to a class. Applying specialization to such structures affects a data member $C::m$ of type D in two "orthogonal" dimensions:

- The "*location*" of m in the class hierarchy is affected by changing the number of objects that contain this field. This is no different from data members of built-in types.
- The *type* of m is replaced by a specialized version of D, containing a subset of D's members. This is no different from the way we treat variables.

Consequently, data members of class-based types should be modeled as built-in data members *and* as variables. This is accomplished by introducing constraint variables $dcl(C::m)$ and $var(C::m)$, which represent the "data member" view and the "variable view" of m, respectively. Constructing the type constraints involving these constraint variables is completely analogous to the case with only data members of built-in types.

10.2.2 Down casts and type-test operations

Down casts are type cast operations where the "target" type T of the cast operation is a derived class of the static type S of the casted expression. Down casting is generally discouraged because a run-time error or exception occurs if the run-time type of the expression is not a class transitively derived from T. However, many realistic programs, especially languages such as Java that lack parametric polymorphism, use downcasting heavily. Type test operations are closely related to down casts, and allow a user to test if a pointer or reference is a (subtype of) a specified type, and compute a boolean or integer value indicating the result. For example, Java allows expressions of the form `e instanceof X`, to test if the object pointed to by `e` is a subtype of `X`. Both down casts and type test operations can be transformed into virtual method calls[16], allowing us to simply rely on the previously discussed mechanisms.

10.2.3 Miscellaneous other features

Other language features that need to be modeled include:

- User-defined constructors and destructors. Typically, a constructor initializes all members of a class. The algorithm presented in the present paper would not be able to omit any members accessed from a constructor's `this` pointer.
- Static members. Although member lookup works somewhat differently for static members [24], we do not think that there are any conceptual difficulties here. From a space savings point of view, static members are not very interesting because there is only one such member per class.
- Reflection and dynamic loading. Reflective features allow one to access an object's class, and from such a class-object members in that class can be inspected or accessed. Since it is in general impossible to determine using static analysis which members may be accessed using reflection, additional user input would be required to perform class hierarchy specialization on programs that use reflection (the approach taken in [34]).

Other pragmatic issues that need to be addressed in order to make class hierarchy specialization practical are separate compilation and the use of class libraries for which only object code is available.

10.3 Simplification rules

While the simplification rules of Section 8 are sufficient for the examples presented in this paper, further research is needed to determine if additional rules are required in other cases. In addition, simplification rules would ideally allow for certain time/space tradeoffs. For example, one might think of a situation where a virtual inheritance relation can be eliminated if a data member m is added to a certain object that does not need m. We intend to investigate whether rewriting systems [7, 16] can be used as a formal means to reason about class hierarchy simplifications.

[16] This transformation was proposed by M. Streckenbach, and is presented in detail in [28].

10.4 Implementation plans

We have started work on an implementation of class hierarchy specialization in the context of *Jax* [34], an application extraction tool for Java which is currently being developed at IBM Research[17]. The main goal of *Jax* is to reduce the time required to download applications over the internet by reducing application size. In [34], *Jax* is evaluated on a number of realistic benchmark applications, and an average ZIP file size reduction of 51.7% is reported. *Jax* incorporates a number of recently developed whole-program analysis techniques such as Rapid Type Analysis [6, 5] and the dead data member detection algorithm of [31].

We have adapted the class hierarchy simplification rules of Section 8 to Java, and implemented them in the context of *Jax*. Java provides a limited form of multiple inheritance for interface classes, and does not make an explicit distinction between virtual and non-virtual inheritance[18]. Therefore, only the M-Rule and the R-Rule of Section 8 have been implemented. For the benchmark applications of [34], the simplification rules reduce the number of classes by an average of 33.8%. This has a nontrivial impact on application size because in the Java class file representation, each class has a local copy of the literal values it refers to, and merging classes reduces the duplication of constants in different classes.

A Language $\mathcal{L}$

Language $\mathcal{L}$ is a small C++-like language with virtual (shared) and nonvirtual (replicated) multiple inheritance. We omitted many C++ features from $\mathcal{L}$, including user-specified constructors and destructors, nonvirtual methods, pure virtual methods and abstract base classes, access rights (for members and inheritance relations; members and subobjects are accessible from anywhere within an $\mathcal{L}$-program), multi-level pointers, functions, operators, overloading, dynamic allocation, pointer arithmetic, pointers-to-members, the '::' direct method call operator, explicit casts, typedefs, templates, exception handling constructs. Furthermore, we assume that data members are of a built-in type. For convenience, we allow classes to contain the declaration of a method without an accompanying definition if the method under consideration is not called. All variable/parameter types are either `int` or a class, data members are always of type `int`, and members may only be accessed from a variable. Fig. 13 shows a BNF grammar for $\mathcal{L}$.

Without loss of generality we assume that the program does not contain variables, parameters, members, and classes with the same name (if this is not the case, some name-mangling scheme can be applied). The only exception to this rule is that we allow a virtual method to override another virtual method with the same name.

[17] More information about *Jax* can be found at `www.research.ibm.com/jax`. A free evaluation copy can be downloaded from `www.alphaWorks.ibm.com/tech/jax`.

[18] Since interfaces cannot contain non-static fields, and all of the declarations of a method in different interfaces refer to the same method, virtual and non-virtual inheritance would have exactly the same semantics anyway.

```
Program      ::=  Hierarchy  void main() { S_List }
Hierarchy    ::=  ClassDef  | ClassDef  Hierarchy
ClassDef     ::=  class  Id  [ : I_List ]  {  M_List };
I_List       ::=  [ virtual ]  Id  |  [ virtual ]  Id, I_List
M_List       ::=  Member; |  Member; M_List
Member       ::=  virtual int  Id( [ D_List ] ) [ { S_List } ] |
                  virtual Id  Id( [ D_List ] ) [ { S_List } ] | int  Id
S_List       ::=  Stat; |  Stat;  S_List
Stat         ::=  Decl  |  IfStat  |  AssignStat  |  ReturnStat  |  CallStat
Decl         ::=  int Id  |  Id [ * ] Id
D_List       ::=  Decl  |  Decl, D_List
IfStat       ::=  if (Id) { S_List }  [ else { S_List } ]
AssignStat   ::=  [ * ] Id  =  Exp  |  Id M_Op Id = Exp
ReturnStat   ::=  return  Exp
CallStat     ::=  CallExp
Exp          ::=  IntConst  |  Id  |  *Id  |  &Id  |  Exp  +  Id  |  CallExp
CallExp      ::=  Id M_Op Id( [ Exp_List ]) |  Id M_Op Id |  Id ( [ Exp_List ])
Exp_List     ::=  Exp  |  Exp, Exp_List
IntConst     ::=  ... | -1 | 0 | 1 | ...
M_Op         ::=  .  |  ->
```

Figure 13: BNF grammar for $\mathcal{L}$.

References

[1] Accredited Standards Committee X3, I. P. S. Working paper for draft proposed international standard for information systems—programming language C++. Doc. No. X3J16/97-0108. Draft of 25 November 1997.

[2] Agesen, O. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, December 1995. Appeared as Sun Microsystems Laboratories Technical Report SMLI TR-96-52.

[3] Agesen, O., and Ungar, D. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)* (Portland, OR, 1994), pp. 355–370. *SIGPLAN Notices* 29(10).

[4] Aigner, G., and Hölzle, U. Eliminating virtual function calls in C++ programs. In *Proceedings of the Tenth European Conference on Object-Oriented Programming (ECOOP'96)* (Linz, Austria, July 1996), vol. 1098 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 142–166.

[5] Bacon, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[6] Bacon, D. F., and Sweeney, P. F. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 324–341. *SIGPLAN Notices* 31(10).

[7] Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., and Sleep, M. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages* (Eindhoven, The Netherlands, 1987), vol. 259 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 141–158.

[8] Calder, B., and Grunwald, D. Reducing indirect function call overhead in C++ programs. *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (January 1994), 397–408.

[9] Carini, P. R., Hind, M., and Srinivasan, H. Flow-sensitive type analysis for C++. Tech. Rep. RC 20267, IBM T.J. Watson Research Center, 1995.

[10] Choi, J.-D., Burke, M., and Carini, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (1993), ACM, pp. 232–245.

[11] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[12] Diwan, A., Moss, J. E. B., and McKinley, K. S. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 292–305. *SIGPLAN Notices* 31(10).

[13] Godin, R., and Mili, H. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)* (Washington, DC, 1993), pp. 394–410. *ACM SIGPLAN Notices* 28(10).

[14] Godin, R., Mili, H., Mineau, G. W., Missaoui, R., Arfi, A., and Chau, T.-T. Design of class hierarchies based on concept (galois) lattices. *Theory and Practice of Object Systems 4*, 2 (1998), 117–134.

[15] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison-Wesley, 1996.

[16] Klop, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.

[17] Liu, Y. A., and Stoller, S. D. Dead code elimination using program-based regular tree grammars. Tech. Rep. TR498, Indiana University, November 1997.

[18] Moore, I. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 235–250. *SIGPLAN Notices* 31(10).

[19] O'Callahan, R., and Jackson, D. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering Programming Systems, Languages, and Applications (ICSE'96)* (Boston, MA, May 1997).

[20] Opdyke, W., and Johnson, R. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference* (1993).

[21] Opdyke, W. F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University Of Illinois at Urbana-Champaign, 1992.

[22] Palsberg, J., and Schwartzbach, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[23] Pande, H. D., and Ryder, B. G. Static type determination and aliasing for C++. Report LCSR-TR-250-A, Rutgers University, October 1995.

[24] Ramalingam, G., and Srinivasan, H. A member lookup algorithm for C++. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation* (Las Vegas, NV, 1997), pp. 18–30.

[25] Rossie, J. G., and Friedman, D. P. An algebraic semantics of subobjects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)* (Austin, TX, 1995), pp. 187–199. *SIGPLAN Notices* 30(10).

[26] Shapiro, M., and Horwitz, S. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages* (Paris, France, 1997), pp. 1–14.

[27] Snelting, G., and Tip, F. Reengineering class hierarchies using concept analysis. In *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6)* (Lake Buena Vista, FL, November 1998), pp. 99–110.

[28] Snelting, G., and Tip, F. Reengineering class hierarchies using concept analysis. Tech. rep., IBM T.J. Watson Research Center, December 1999. Forthcoming.

[29] Srivastava, A. Unreachable procedures in object oriented programming. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 355–364.

[30] Steensgaard, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[31] Sweeney, P. F., and Tip, F. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation* (Montreal, Canada, June 1998), pp. 324–332.

[32] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[33] Tip, F., Choi, J.-D., Field, J., and Ramalingam, G. Slicing class hierarchies in C++. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, 1996), pp. 179–197. *SIGPLAN Notices* 31(10).

[34] Tip, F., Laffra, C., Sweeney, P. F., and Streeter, D. Practical experience with an application extractor for java. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming, Languages, and Applications (OOPSLA'99)* (November 1999), pp. 292–305. *SIGPLAN Notices* 34(10).

[35] Tip, F., and Sweeney, P. F. Class hierarchy specialization. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. ACM SIGPLAN Notices 32(10).

[36] Weiser, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

Acknowledgements

John Field, Yossi Gil, G. Ramalingam, and Gregor Snelting made many useful suggestions. The constructive feedback by the anonymous referees is also much appreciated.