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

Abstract

Class libraries are generally designed with an emphasis on versatility and extensibility. Applications that use a library typically exercise only part of the library's functionality. As a result, objects created by the application may contain unused members. We present an algorithm that *specializes* 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. Class hierarchy specialization reduces object size, and is hence primarily a space optimization. However, execution time may also be reduced through reduced object creation/destruction time, and caching/paging effects.

1 Introduction

Class libraries are generally designed with an emphasis on versatility and extensibility. An application that uses a class library typically exercises only part of the library's functionality. Unfortunately, this leads to situations where the objects created by the program contain 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 effectively 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/paging effects.
- Specialization may create new opportunities for existing optimizations such as virtual function call resolution [4, 9, 3, 8, 5].
- Specialization may be of use in program understanding and debugging tools. For example, specialization can be used as a means to suppress the output of unused parts of objects during a debugging session.

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.

To appear in the proceedings of the 12th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97), October 5–8 1997, Atlanta, GA.

1.1 Scope of this paper

The motivation for this work is to reduce the overhead incurred by class library usage in C++ applications. In order to prevent our definitions and algorithms from becoming too unwieldy, the present paper 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 inheritance. We have omitted many C++ features from $\mathcal{L}$ because they would clutter the presentation of the algorithm. E.g., we only allow default constructors/destructors, and members are assumed to be accessible from anywhere in a program, thus ignoring all issues related to access rights. In addition, several features have been omitted from $\mathcal{L}$ because their treatment is future work (e.g., explicit type-cast operations, and nested structures). This being said, we believe that our techniques are in principle applicable to realistic languages such as C++ and Java, although much engineering work remains to be done—future work will be discussed in Section 6. 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

Figure 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

```
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 T_var(s1) { int m1; };
class T_var(s2) :  T_var(s1) { int m2; };
class T_var(s3) :  T_var(s2) { int m3; };

void main(){
  T_var(s1) s1; T_var(s2) s2; T_var(s3) 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.

requires that y's type be transitively derived from x's type [1]. The specialized class hierarchy of Figure 1(b) shows how this observation can be exploited, using inheritance relations between the types of `s1`, `s2`, and `s3`. Note that `s1` and `s2` now contain fewer members (the number of members of `s3` remains the same) while program behavior is preserved.

Figure 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 Figure 2(b). Note that the functions `A::f()`, `A::g()`, `B::g()`, and `C::f()` are dispersed over four classes $T_{var(\texttt{*ap})}$, $T_{var(\texttt{a})}$, $T_{var(\texttt{b})}$, and $T_{var(\texttt{c})}$, and that class $T_{var(\texttt{*ap})}$ only contains a *declaration*[2] of method `g()`. Observe that the use of a common base class $T_{var(\texttt{*ap})}$ with only virtual methods allows us to eliminate the `x` data member from `b` and `c`.

Since the size of an object is strongly implementation-dependent it is difficult to make general statements about the space savings of this operation. If we assume that $\mathcal{L}$ uses the same object model as the IBM `xlC` C++ compiler on the RS/6000/AIX 4.1 platform, the size of variable `a` would remain unchanged at 8 bytes, the size of `b` would be reduced from 12 to 8 bytes, and the size of `c` would be reduced from 16 to 12 bytes.

1.3 Overview of algorithm

The specialization algorithm consists of four distinct phases. In Phase I, discussed in Section 2, 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.

Phase II 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. Phase II is presented in Section 3.

In Phase III, the type constraints of Phase II are used to construct a new class hierarchy, and the variable declarations in the program are updated to take this new hierarchy into account. This is discussed in Section 4.

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. Section 5 discusses Phase IV.

1.4 Related work

A number of categories of related work can be distinguished.

The first category consists of techniques for eliminating unused components from objects or class hierarchies. Tip, et al. [23] present a class hierarchy slicing algorithm 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 by *any* instance of type C. 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.

Agesen and Ungar [2, 1] 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 algorithm computes, 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 different nature of Self and $\mathcal{L}$. 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, because Self is 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, such as methods, from object-oriented programs [4, 20, 15]. 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* [25, 22], which determines the set of executable statements that may affect the values computed at some designated point(s) of interest in a program. Redundant statements can be removed from a program by slicing w.r.t. all output values.

Another view on class hierarchy specialization is that of a *type inference* algorithm, which infers a set of nonstandard types for

[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 an Y-object.

[2] In $\mathcal{L}$, methods only need to be defined if they are invoked. This is not the case in C++.

```
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.

variables, and constructs a new class hierarchy reflecting these.

For a discussion of type inference for object-oriented languages, we refer the reader to the seminal work on object-oriented type systems by Palsberg and Schwartzbach [15]. There are some interesting connections between our work and that of [15]. Since Palsberg and Schwartzbach study a language with only single inheritance, they can express an expression's type as a set of classes. In the presence of multiple inheritance, this is not possible, and a more sophisticated mechanism such as Rossie and Friedman's subobject-based types is required [18]. Our notion of a type constraint is similar in spirit to Palsberg and Schwartzbach's, but due to fact that types cannot be expressed as sets of classes, type constraints cannot be expressed using the subset operator. There is also a difference in the way type constraints are used. Palsberg and Schwartzbach submit all type constraints to an inference engine, which infers a type for each program variable. In contrast, in our case, where initial types are known for each variable, type constraints are not solved but interpreted as a new class hierarchy for the program.

O'Callahan and Jackson [12] use type inference to determine statically where the structure of a C program requires sets of variables to share a common representation. Although they are primarily interested in program understanding applications such as finding abstraction violations, their algorithm also detects unused fields of data structures. Since C does not have inheritance, O'Callahan and Jackson do not address the complex issues related to subtyping that arise in our setting. We believe that, in principle, the nonstandard types inferred by our algorithm can be used for the same program understanding applications as those mentioned in [12].

A third 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 and Johnson [13, 14] 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, class hierarchy specialization requires no programmer intervention.

Moore [11] 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.

2 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 3) 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 in $\mathcal{P}$ whose type is a pointer to a class (we will henceforth use the word "variables" to refer to variables as well as method parameters). 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 type of expression x in program $\mathcal{P}$.

2.1 Variables

Definition 2.1 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. *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 2.1 *Let $\mathcal{P}$ be a program. Then, we define the sets of class-typed variables and the set of pointer-to-class-typed variables as follows:*

$$\begin{array}{l} \mathit{ClassVars}(\mathcal{P}) \triangleq \\ \quad \{\, v \mid \; v \text{ is a variable/parameter in } \mathcal{P}, \\ \qquad \mathit{TypeOf}(\mathcal{P}, v) = C, \text{ for some class } C \text{ in } \mathcal{P} \,\} \end{array}$$

$$\begin{array}{l} \mathit{ClassPtrVars}(\mathcal{P}) \triangleq \\ \quad \{\, p \mid \; p \text{ is a variable/parameter in } \mathcal{P}, \\ \qquad \mathit{TypeOf}(\mathcal{P}, *p) = C, \text{ for some class } C \text{ in } \mathcal{P} \,\} \end{array}$$

Example 2.2 *For program $\mathcal{P}_2$ of Figure 2, we have:*

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

□

2.2 Class members

For a given program $\mathcal{P}$, *Members*($\mathcal{P}$) denotes the set of unqualified names of the class members that occur in $\mathcal{P}$. In addition, the sets *DataMembers*($\mathcal{P}$), and *VirtualMethods*($\mathcal{P}$) contain the unqualified names of data members and virtual methods of $\mathcal{P}$, respectively. For convenience, we assume the intersection of *DataMembers*($\mathcal{P}$) and *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 2.3 *For program $\mathcal{P}_2$ of Figure 2, we have:*

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

□

2.3 Points-to analysis

We will need for each pointer-to-class-typed variable a conservative approximation of the set of class-typed variables that it may point to in some execution of $\mathcal{P}$. Any of several existing algorithms [7, 6, 16, 21, 19]) can be used to compute this information, and we do not make assumptions about the particular algorithm used to compute points-to information. Definition 2.4 uses the information supplied by some points-to analysis algorithm to construct a set *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 2.4 *Let $\mathcal{P}$ be a program. Then, the points-to information for $\mathcal{P}$ is defined as follows:*

$$\begin{array}{l} \mathit{PointsTo}(\mathcal{P}) \triangleq \; \{\, \langle p, v\rangle \mid \; p \in \mathit{ClassPtrVars}(\mathcal{P}), \\ \qquad\qquad\qquad\qquad v \in \mathit{ClassVars}(\mathcal{P}), \\ \qquad\qquad\qquad\qquad p \text{ may point to } v \,\} \end{array}$$

Example 2.5 *We will use the following points-to information for program $\mathcal{P}_2$. Recall that $X::f$ denotes the* `this` *pointer of method $X::f()$.*

$$\begin{array}{l} \mathit{PointsTo}(\mathcal{P}_2) \equiv \{ \\ \quad \langle \mathtt{ap}, \mathtt{a}\rangle,\ \langle \mathtt{ap}, \mathtt{b}\rangle,\ \langle \mathtt{ap}, \mathtt{c}\rangle,\ \langle \mathtt{A::f}, \mathtt{a}\rangle,\ \langle \mathtt{A::f}, \mathtt{b}\rangle, \\ \quad \langle \mathtt{C::f}, \mathtt{c}\rangle,\ \langle \mathtt{A::g}, \mathtt{a}\rangle,\ \langle \mathtt{B::g}, \mathtt{b}\rangle,\ \langle \mathtt{B::g}, \mathtt{c}\rangle \,\} \end{array}$$

□

Note that the following simple algorithm suffices to compute the information of Example 2.5: 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.

2.4 Assignments

Definition 2.6 below defines a set *Assignments*($\mathcal{P}$) that contains a pair of objects $\langle x', y'\rangle$ for each assignment $x = y$ that occurs in $\mathcal{P}$. For a *direct* call[3], *Assignments*($\mathcal{P}$) contains elements that model parameter-passing between corresponding formal and actual parameters whose type is a (pointer to a) class. The return value of a method is treated as an additional parameter as well if the method's return type is a class. For an *indirect* call $p \rightarrow f(y_1, \ldots, y_n)$, *Assignments*($\mathcal{P}$) contains elements that model the parameter-passing in the direct call $x.f(y_1, \ldots, y_n)$, for each $\langle p, x\rangle \in \mathit{PointsTo}(\mathcal{P})$.

Definition 2.6 *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:*

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

Example 2.7 *For program $\mathcal{P}_2$ of Figure 2, we have:*

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

□

2.5 Member access operations

Definition 2.8 below defines a set *MemberAccess*($\mathcal{P}$) of all pairs $\langle m, x\rangle$ 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 $\langle f, x\rangle$ in *MemberAccess*($\mathcal{P}$) for each $\langle p, x\rangle \in \mathit{PointsTo}(\mathcal{P})$.

[3] A *direct* call is an invocation of a virtual method from a non-pointer typed variable. An *indirect* call is an invocation of a virtual method from a pointer, which requires the virtual dispatch mechanism to be invoked.

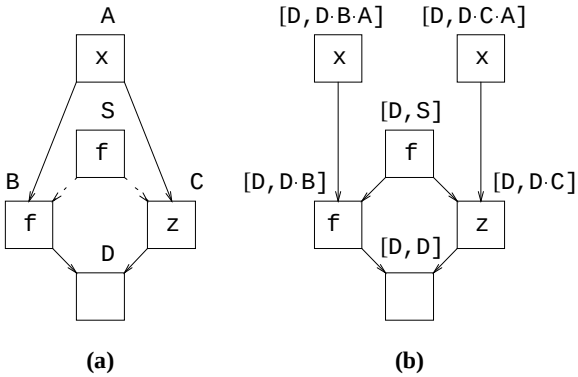


Figure 3: **(a)** Example class hierarchy graph. Solid edges indicate replicated (nonvirtual) inheritance. Dashed edges indicate virtual (i.e., shared) inheritance. **(b)** Subobject graph for type D in the class hierarchy of Figure 3(a).

Definition 2.8 *Let $\mathcal{P}$ be a program. Then, the set of member access operations in $\mathcal{P}$ is defined as follows:*

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

Example 2.9 *For program $\mathcal{P}_2$ of Figure 2, we have:*

$$\begin{aligned} &\mathit{MemberAccess}(\mathcal{P}_2) \equiv \\ &\quad \{\ \langle \mathsf{x}, \mathsf{*A::g}\rangle, \langle \mathsf{y}, \mathsf{*B::g}\rangle, \langle \mathsf{z}, \mathsf{*C::f}\rangle, \langle \mathsf{g}, \mathsf{*A::f}\rangle, \langle \mathsf{g}, \mathsf{*C::f}\rangle, \\ &\qquad \langle \mathsf{f}, \mathsf{*ap}\rangle, \langle \mathsf{f}, \mathsf{a}\rangle, \langle \mathsf{f}, \mathsf{b}\rangle, \langle \mathsf{f}, \mathsf{c}\rangle, \langle \mathsf{g}, \mathsf{a}\rangle, \langle \mathsf{g}, \mathsf{b}\rangle, \langle \mathsf{g}, \mathsf{c}\rangle\ \} \end{aligned}$$

□

3 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 between the types of variables that must be preserved in the specialized class hierarchy.

3.1 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 these subobjects and members with the same name, we need to keep track of the subobjects selected by the execution of member lookup and typecast operations. To this end, we use the formalization of subobject graphs and member lookup of [18, 23]. Here, we present this model informally; the reader is referred to [23] for details. An efficient algorithm for performing member lookups can be found in [17].

A subobject graph can be viewed as an abstract representation of the layout of an object. 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, and the subgraph for a specific type. Figure 3(a) depicts a class hierarchy in which a class D inherits nonvirtually (replicated) from classes B and C, and classes B and C both 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.

Figure 3(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 uniquely identifies a subobject of type X_n inside Y. For a subobject $\sigma \equiv [Y, X_1 \cdots X_n]$, $\mathit{mdc}(\sigma)$ denotes its most derived class Y, and $\mathit{ldc}(\sigma)$ 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 $\mathit{ldc}(\sigma)$. Edges in the subobject graph of Figure 3(b) reflect the *containment* relation '$<$' between subobjects[4]. In what follows, $\Sigma(\gamma)$ denotes the set of all subobjects σ for class hierarchy γ.

In the example of Figure 3(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:

$$\mathsf{[D,D{\cdot}B]} < \mathsf{[D,D]}$$

Due to the 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 typecast operations can be defined as functions from subobjects to subobjects. We will only discuss *static-lookup* and *typecast* here, which are functions that model static[5] member lookups and typecasts, respectively. Dynamic member lookups can be modeled similarly [23].

Static member lookups are modeled by way of a function *static-lookup* whose arguments are: (i) an initial subobject σ, (ii) a member m, and (iii) the containment relation '$<$' between subobjects. *static-lookup* returns the subobject σ' in which the accessed member is located. As an example, we study a static lookup for an expression $e.m$ or $e \rightarrow m$ that occurs in program $\mathcal{P}$. First, an *initial subobject* σ is associated with receiver e. For a static lookup, this initial subobject is simply $[T,T]$, where $T \equiv \mathit{TypeOf}(\mathcal{P}, e)$. Then, *static-lookup* determines the unique subobject σ' such that (i) $\sigma' \leq^* \sigma$, (ii) σ' contains member m, and (iii) if there is a $\sigma'' < \sigma$ that contains a member m, then $\sigma'' \leq^* \sigma'$ (here, '$\leq^*$' denotes the transitive and reflexive closure of '$<$').

As an example, suppose that there is a lookup d.z, where d is an object of type D. In this case, the initial subobject is [D,D], and there is one visible definition for z in subobject [D,D·C]. Consequently, the lookup function returns subobject [D,D·C] indicating that member z in the [D,D·C]-subobject of D is accessed. In other words, we have that:

$$\mathit{static\text{-}lookup}(\mathsf{[D,D]}, \mathsf{z}, \text{'}<\text{'}) \equiv \mathsf{[D,D{\cdot}C]}$$

Typecasts can be modeled as follows. For a cast from type X to type Y (in the present paper, we only allow the case where Y is a transitive base class of X), the unique subobject $\sigma' \leq^*$[X,X] for which we have that $\mathit{ldc}(\sigma') \equiv Y$ is selected. For example, 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

[4] 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 [18, 23], where the contained subobject is "greater than" the containing subobject.

[5] We call a member lookup *static* if it corresponds to a data member access or a direct method call and *dynamic* if it corresponds to a method call that invokes the virtual dispatch mechanism.

```
class A {
  virtual void foo(){ this->x = 17; };
  int x;
};

void main(){
  A a1; A a2; A *ap;
  ap = &a1;
  *ap = a2;
  ap->foo();
  int result = a1.x;
}
```

(a)

```
class T_var(a2) {
  virtual void foo(); /* declaration */
};

class T_var(a1) :  T_var(a2) {
  virtual void foo(){ this->x = 17; };
  int x;
};

void main(){
  T_var(a1) a1; T_var(a2) a2; T_var(a2) *ap;
  ap = &a1;
  *ap = a2;
  ap->foo();
  int result = a1.x;
}
```

(b)

Figure 4: **(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)**.

generates a typecast from type `D` to type `B`. For this typecast, we have that:

$$\mathit{typecast}([\mathtt{D},\mathtt{D}], \mathtt{B}, \text{`}<\text{'}) \equiv [\mathtt{D},\mathtt{D}{\cdot}\mathtt{B}]$$

This implies that the assignment copies the `[D,D·B]`-subobject of `d` into `b`.

We conclude the discussion of subobjects by introducing a composition operator '$\oplus$'. Intuitively, this operator determines the subobject of a subobject. E.g., $[\mathtt{D},\mathtt{D}{\cdot}\mathtt{B}] \oplus [\mathtt{B},\mathtt{B}{\cdot}\mathtt{A}] \equiv [\mathtt{D},\mathtt{D}{\cdot}\mathtt{B}{\cdot}\mathtt{A}]$.

3.2 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.

Figure 4 illustrates this issue by way of a simple program that uses two class-typed variables `a1` and `a2`, and a class-pointer-typed variable `ap` that may point to `a1`, but not to `a2`. 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 to obtain a specialized class hierarchy. For convenience, we will frequently write "member m must be visible/accessible[6] 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 `a1` and `a2`. Otherwise, the assignments `ap = &a1` and `*ap = a2` would not be type-correct. Since virtual method `foo()` is called from `ap`, a *declaration* of `foo()` must be visible to `ap`, and since `ap` may point to `a1`, the *definition* of `A::foo()` must be visible to `a1`. Data member `x` must be visible to `A::foo` because it is accessed from `A::foo` (recall that `A::foo` denotes the `this` pointer of method `A::foo()`). However, note that `x` need not be visible to `a2`. In fact, it is *undesirable* for `A::foo`'s *definition* to be visible to `ap`, because that forces inclusion of `x` in `a2`.

Figure 4(b) shows the specialized program and class hierarchy for the example of Figure 4(a). Note that, while the above constraints are met, data member `x` has been eliminated from `a2`.

In the sequel, $\mathit{def}(A{::}m)$ denotes the definition of member $A{::}m$, whereas $\mathit{decl}(A{::}m)$ denotes its declaration. As the example of Figure 4 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 declaration and definition is not useful, and only a definition is required.)

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

3.3 Type constraints and constraint elements

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 *constraint elements*, as defined by Definition 3.1 below.

Definition 3.1 *Let $\mathcal{P}$ be a program. Then, the set of constraint elements for $\mathcal{P}$ is defined as follows:*

$$\begin{aligned}
\mathit{Elems}(\mathcal{P}) \triangleq\ & \{\ \mathit{var}(v) \mid v \in \mathit{ClassVars}(\mathcal{P})\ \} \cup \\
& \{\ \mathit{var}(*p) \mid\ p \in \mathit{ClassPtrVars}(\mathcal{P}), \\
& \qquad p \text{ is not a method's } \mathtt{this} \text{ pointer}\ \} \cup \\
& \{\ \mathit{decl}(X{::}m) \mid\ m \in \mathit{Members}(\mathcal{P}), \\
& \qquad m \text{ occurs in class } X\ \} \cup \\
& \{\ \mathit{def}(X{::}m) \mid\ m \in \mathit{VirtualMethods}(\mathcal{P}), \\
& \qquad m \text{ occurs in class } X\ \}
\end{aligned}$$

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

$$\begin{aligned}
\mathit{Elems}(\mathcal{P}_2) \equiv \{\ & \mathit{var}(\mathtt{a}),\ \mathit{var}(\mathtt{b}),\ \mathit{var}(\mathtt{c}),\ \mathit{var}(\mathtt{*ap}),\ \mathit{decl}(\mathtt{A{::}x}), \\
& \mathit{decl}(\mathtt{B{::}y}),\ \mathit{decl}(\mathtt{C{::}z}),\ \mathit{decl}(\mathtt{A{::}f}),\ \mathit{decl}(\mathtt{A{::}g}), \\
& \mathit{decl}(\mathtt{B{::}g}),\ \mathit{decl}(\mathtt{C{::}f}),\ \mathit{def}(\mathtt{A{::}f}),\ \mathit{def}(\mathtt{A{::}g}), \\
& \mathit{def}(\mathtt{B{::}g}),\ \mathit{def}(\mathtt{C{::}f})\ \}
\end{aligned}$$

□

Type constraints express subtype-relationships between constraint elements. For example, $\langle \mathit{var}(\mathrm{v}), \sigma, \mathit{var}(\mathrm{w})\rangle$ 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 $\langle \mathit{decl}(A{::}m), \sigma, \mathit{var}(w)\rangle$ 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, `this` pointers of methods will require somewhat special treatment. Definition 3.3 below maps a variable v in the program to a constraint element $\mathit{var}(v)$ if v is not the `this` pointer of a method, and to $\mathit{def}(A{::}m)$ if v is the `this` pointer of some method $A{::}m$.

Definition 3.3 *Let x be an expression such that $x \equiv v$ for some $v \in ClassVars(\mathcal{P})$ or $x \equiv *p$ for some $p \in ClassPtrVars(\mathcal{P})$. Then, the constraint element associated with x is defined as follows:*

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

Example 3.4 *For program $\mathcal{P}_2$, we have $Elem(\mathtt{a}) \equiv var(\mathtt{a})$ and $Elem(\mathtt{*A::f}) \equiv def(\mathtt{A::f})$.* □

3.4 Type constraints due to assignments

Consider an assignment $v = w$, where v is of class type V and w of class type W. This assignment is only type-correct if [W,W] contains a unique subobject $\sigma \equiv [W,\alpha \cdot V]$, where α is some (possibly empty) sequence of class names. This subobject σ can be defined as $\sigma \equiv typecast([W,W], V, \text{'<'})$. Definition 3.5 below defines the set of type constraints implied by assignments. These constraints are of the form $\langle var(x), \sigma, var(y)\rangle$, which can be interpreted as 'x must have the same type as the σ-subobject of y', and $\langle def(A::f), \sigma, var(y)\rangle$ which can be interpreted as 'the `this` pointer of method $A::f$ must have the same type as the σ-subobject of y'.

Definition 3.5 *Let $\mathcal{P}$ be a program. Then, the set of type constraints due to assignments is defined as follows:*

$$\begin{aligned} AssignTC(\mathcal{P}) \triangleq \{ \quad & \langle Elem(x), \sigma, Elem(y)\rangle \mid \\ & \langle x, y\rangle \in Assignments(\mathcal{P}), \\ & X = TypeOf(\mathcal{P}, x),\ Y = TypeOf(\mathcal{P}, y), \\ & \sigma = typecast([Y,Y], X, \text{'<'}) \} \end{aligned}$$

Example 3.6 *For program $\mathcal{P}_2$ of Figure 2, we have:*

$$\begin{aligned} & AssignTC(\mathcal{P}_2) \equiv \{ \\ & \quad \langle var(\mathtt{*ap}), [\mathtt{A,A}], var(\mathtt{a})\rangle,\ \langle var(\mathtt{*ap}), [\mathtt{B,B{\cdot}A}], var(\mathtt{b})\rangle, \\ & \quad \langle var(\mathtt{*ap}), [\mathtt{C,C{\cdot}B{\cdot}A}], var(\mathtt{c})\rangle,\ \langle def(\mathtt{A::f}), [\mathtt{A,A}], var(\mathtt{a})\rangle, \\ & \quad \langle def(\mathtt{A::f}), [\mathtt{B,B{\cdot}A}], var(\mathtt{b})\rangle,\ \langle def(\mathtt{C::f}), [\mathtt{C,C}], var(\mathtt{c})\rangle, \\ & \quad \langle def(\mathtt{A::g}), [\mathtt{A,A}], var(\mathtt{a})\rangle,\ \langle def(\mathtt{B::g}), [\mathtt{B,B}], var(\mathtt{b})\rangle, \\ & \quad \langle def(\mathtt{B::g}), [\mathtt{C,C{\cdot}B}], var(\mathtt{c})\rangle \} \end{aligned}$$

□

3.5 Type constraints due to member access

Definition 3.7 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. The second case addresses the situation where m's definition is required, i.e., when a virtual method is invoked from a nonpointer variable v. As an example, consider the case where a virtual method m is accessed from a pointer p. Assuming that p has type $*Y$, there must be a unique subobject $\sigma \equiv [Y,\alpha \cdot X] \equiv static\text{-}lookup([Y,Y], m, \text{'<'})$ such that X contains m. Since the virtual dispatch mechanism only requires that a *declaration* of m be present in class X, a constraint $\langle decl(X::m), \sigma, var(*p)\rangle$ is constructed, expressing the fact that the σ-subobject of $*p$ must contain a declaration of method $X::m()$.

Definition 3.7 *Let $\mathcal{P}$ be a program. Then, the set of type constraints due to member access operations is defined as follows:*

$$\begin{aligned} & MemberAccessTC(\mathcal{P}) \triangleq \\ & \quad \{ \ \langle decl(X::m), \sigma, Elem(y)\rangle \mid Y \equiv TypeOf(\mathcal{P}, y), \\ & \qquad \langle m, y\rangle \in MemberAccess(\mathcal{P}), \\ & \qquad (y \equiv *p \text{ for some } p \in ClassPtrVars(\mathcal{P}) \\ & \qquad\quad \text{or } m \in DataMembers(\mathcal{P})), \\ & \qquad \sigma \equiv [Y,\alpha \cdot X] \equiv static\text{-}lookup([Y,Y], m, \text{'<'}) \} \cup \\ & \quad \{ \ \langle def(X::m), \sigma, var(y)\rangle \mid Y \equiv TypeOf(\mathcal{P}, y), \\ & \qquad \langle m, y\rangle \in MemberAccess(\mathcal{P}), \\ & \qquad (y \equiv v \text{ for some } v \in ClassVars(\mathcal{P}) \\ & \qquad\quad \text{and } m \in VirtualMethods(\mathcal{P})), \\ & \qquad \sigma \equiv [Y,\alpha \cdot X] \equiv static\text{-}lookup([Y,Y], m, \text{'<'}) \} \end{aligned}$$

Example 3.8 *For program $\mathcal{P}_2$ of Figure 2, we have:*

$$\begin{aligned} & MemberAccessTC(\mathcal{P}_2) \equiv \{ \\ & \quad \langle decl(\mathtt{A::x}), [\mathtt{A,A}], def(\mathtt{A::g})\rangle,\ \langle decl(\mathtt{B::y}), [\mathtt{B,B}], def(\mathtt{B::g})\rangle, \\ & \quad \langle decl(\mathtt{C::z}), [\mathtt{C,C}], def(\mathtt{C::f})\rangle,\ \langle decl(\mathtt{A::g}), [\mathtt{A,A}], def(\mathtt{A::f})\rangle, \\ & \quad \langle decl(\mathtt{B::g}), [\mathtt{C,C{\cdot}B}], def(\mathtt{C::f})\rangle,\ \langle decl(\mathtt{A::f}), [\mathtt{A,A}], var(\mathtt{*ap})\rangle, \\ & \quad \langle def(\mathtt{A::f}), [\mathtt{A,A}], var(\mathtt{a})\rangle,\ \langle def(\mathtt{A::f}), [\mathtt{B,B{\cdot}A}], var(\mathtt{b})\rangle, \\ & \quad \langle def(\mathtt{C::f}), [\mathtt{C,C}], var(\mathtt{c})\rangle,\ \langle def(\mathtt{A::g}), [\mathtt{A,A}], var(\mathtt{a})\rangle, \\ & \quad \langle def(\mathtt{B::g}), [\mathtt{B,B}], var(\mathtt{b})\rangle,\ \langle def(\mathtt{B::g}), [\mathtt{C,C{\cdot}B}], var(\mathtt{c})\rangle \} \end{aligned}$$

□

3.6 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.
- Method declarations and method definitions are modeled in similar ways.
- Since the type of a `this` pointer is determined by the location of the associated method in the class hierarchy, any constraint involving the `this` pointer of some method $C::f()$ is effectively a constraint on the location of $C::f()$ in the hierarchy.

We obtain the desired properties by mapping `this` pointers to constraint elements for the associated method definitions (see Definition 3.3). As a result, assignments and member access operations involving `this` pointers give rise to constraints involving the associated method definition. For example, the access to data member `x` from `A::g`'s `this` pointer gives rise to the type constraint $\langle decl(\mathtt{A::x}), [\mathtt{A,A}], def(\mathtt{A::g})\rangle$, 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 there is a slight drawback: identical type constraints occur in $AssignTC(\mathcal{P})$ and $MemberAccessTC(\mathcal{P})$. For example, the constraint $\langle def(\mathtt{A::f}), [\mathtt{A,A}], var(\mathtt{a})\rangle$ occurs in both $AssignTC(\mathcal{P}_2)$ and $MemberAccessTC(\mathcal{P}_2)$ (see Examples 3.6 and 3.8). 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 have an effect on the specialized class hierarchy.

3.7 Type constraints for preserving dominance

Definition 3.9 introduces a set of type constraints that model hiding/dominance relations between declarations and definitions of members with the same name. Informally, the definition states that if there are classes A and B in the original hierarchy that both define a member m, $A \neq B$, and A is a transitive base class of B then $def(A::m)$ must be contained in the σ''-subobject of $def(B::m)$. Similar relationships are constructed for cases where A and B both contain a declaration of a member, and where A contains a declaration and B a definition (in the latter case, a constraint is also constructed if $A = B$). These type constraints are used in Phase III to ensure that member lookup behavior is preserved in cases where multiple declarations/definitions of a member m are visible to a variable.

Definition 3.9 *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:*

$$
\begin{array}{l}
DominanceTC(\mathcal{P}) \triangleq \\
\quad \{ \ \langle decl(A::m), \sigma, def(B::m)\rangle \ | \\
\qquad A = B \text{ or } A \text{ is a transitive base class of } B, \\
\qquad \text{class } A \text{ contains a declaration of member } m, \\
\qquad \text{class } B \text{ contains a definition of member } m, \\
\qquad \sigma \equiv typecast([B,B], A, \text{`<'}) \ \} \cup \\
\quad \{ \ \langle decl(A::m), \sigma, decl(B::m)\rangle \ | \\
\qquad A \neq B, \ A \text{ is a transitive base class of } B, \\
\qquad \text{class } A \text{ contains a declaration of member } m, \\
\qquad \text{class } B \text{ contains a declaration of member } m, \\
\qquad \sigma \equiv typecast([B,B], A, \text{`<'}) \ \} \cup \\
\quad \{ \ \langle def(A::m), \sigma, def(B::m)\rangle \ | \\
\qquad A \neq B, \ A \text{ is a transitive base class of } B, \\
\qquad \text{class } A \text{ contains a definition of member } m, \\
\qquad \text{class } B \text{ contains a definition of member } m, \\
\qquad \sigma \equiv typecast([B,B], A, \text{`<'}) \ \}
\end{array}
$$

Example 3.10 *For program $\mathcal{P}_2$ of Figure 2, we have:*

$$
\begin{array}{l}
DominanceTC(\mathcal{P}_2) \equiv \{ \\
\quad \langle def(\mathsf{A::f}), [\mathsf{C,C{\cdot}B{\cdot}A}], def(\mathsf{C::f})\rangle, \\
\quad \langle def(\mathsf{A::g}), [\mathsf{B,B{\cdot}A}], def(\mathsf{B::g})\rangle, \\
\quad \langle decl(\mathsf{A::f}), [\mathsf{C,C{\cdot}B{\cdot}A}], decl(\mathsf{C::f})\rangle, \\
\quad \langle decl(\mathsf{A::g}), [\mathsf{B,B{\cdot}A}], decl(\mathsf{B::g})\rangle, \\
\quad \langle decl(\mathsf{A::f}), [\mathsf{A,A}], def(\mathsf{A::f})\rangle, \\
\quad \langle decl(\mathsf{A::f}), [\mathsf{C,C{\cdot}B{\cdot}A}], def(\mathsf{C::f})\rangle, \\
\quad \langle decl(\mathsf{A::g}), [\mathsf{A,A}], def(\mathsf{A::g})\rangle, \\
\quad \langle decl(\mathsf{A::g}), [\mathsf{B,B{\cdot}A}], def(\mathsf{B::g})\rangle, \\
\quad \langle decl(\mathsf{B::g}), [\mathsf{B,B}], def(\mathsf{B::g})\rangle, \\
\quad \langle decl(\mathsf{C::f}), [\mathsf{C,C}], def(\mathsf{C::f})\rangle \ \}
\end{array}
$$

□

4 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.

4.1 Classes of the specialized hierarchy

Definition 4.1 below defines the types of the specialized class hierarchy. The specialized hierarchy contains a class T_e, for each constraint element e in $Elems(\mathcal{P})$ (see Definition 3.1).

Definition 4.1 *Let $\mathcal{P}$ be a program. Then, the classes of the specialized class hierarchy are defined as follows:*

$$NewClasses(\mathcal{P}) \triangleq \{ \ T_e \ | \ e \in Elems(\mathcal{P}) \ \}$$

Example 4.2 *For program $\mathcal{P}_2$ of Figure 2, we have:*

$$
\begin{array}{l}
NewClasses(\mathcal{P}_2) \equiv \{ \\
\quad T_{var(\mathsf{a})}, \ T_{var(\mathsf{b})}, \ T_{var(\mathsf{c})}, \ T_{var(\mathsf{*ap})}, \ T_{decl(\mathsf{A::x})}, \ T_{decl(\mathsf{B::y})}, \\
\quad T_{decl(\mathsf{C::z})}, \ T_{decl(\mathsf{A::f})}, \ T_{decl(\mathsf{A::g})}, \ T_{decl(\mathsf{B::g})}, \ T_{decl(\mathsf{C::f})}, \\
\quad T_{def(\mathsf{A::f})}, \ T_{def(\mathsf{A::g})}, \ T_{def(\mathsf{B::g})}, \ T_{def(\mathsf{C::f})} \ \}
\end{array}
$$

□

4.2 The specialized subobject graph

Definitions 4.3 through 4.6 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 constraint elements in $Elems(\mathcal{P})$.

Definition 4.3 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 4.3 *Let $\mathcal{P}$ be a program. Then, the set of nodes N of the specialized subobject graph is inductively defined as follows:*

$$
\begin{array}{l}
T_{var(v),\sigma,var(v)} \in N \quad \text{when} \\
\quad v \in ClassVars(\mathcal{P}), \ V \equiv TypeOf(\mathcal{P}, v), \ \sigma \equiv [V,V]
\end{array}
$$

$$
\begin{array}{l}
T_{s,\sigma_2 \oplus \sigma_1, u} \in N \quad \text{when } T_{t,\sigma_2,u} \in N, \\
\quad \langle s, \sigma_1, t\rangle \in (AssignTC(\mathcal{P}) \cup MemberAccessTC(\mathcal{P}))
\end{array}
$$

Definition 4.4 below defines the most derived class and the least derived class for nodes in N.

Definition 4.4 *Let $n \equiv T_{s,\sigma,u}$ 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:*

$$ldc(T_{s,\sigma,u}) \triangleq T_s \quad mdc(T_{s,\sigma,u}) \triangleq T_u$$

Definition 4.5 below defines a mapping from subobjects in the specialized class hierarchy to subobjects in the original class hierarchy.

Definition 4.5 *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:*

$$\psi(T_{s,\sigma,u}) \triangleq \sigma, \ \text{where } T_{s,\sigma,u} \in N$$

Definition 4.6 below defines a relation '$\sqsubset$' on subobjects in N. The '$\oplus$' operator used in this definition was introduced in Section 3.1. The inclusion-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 $DominanceTC(\mathcal{P})$. This approach has the effect of selecting the appropriate subset of dominance relationships from $DominanceTC(\mathcal{P})$ that is needed to preserve the behavior of typecasts and member lookups in $\mathcal{P}$.

Definition 4.6 *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:*

$$
\begin{array}{lll}
n \sqsubset n' & \text{when} & \psi(n) = \psi(n') \oplus \sigma, \\
& & T_s \equiv ldc(n), \ T_t \equiv ldc(n'), \\
& & \langle s, \sigma, t\rangle \in (\ AssignTC(\mathcal{P}) \cup \\
& & \qquad MemberAccessTC(\mathcal{P}) \cup \\
& & \qquad DominanceTC(\mathcal{P}) \ )
\end{array}
$$

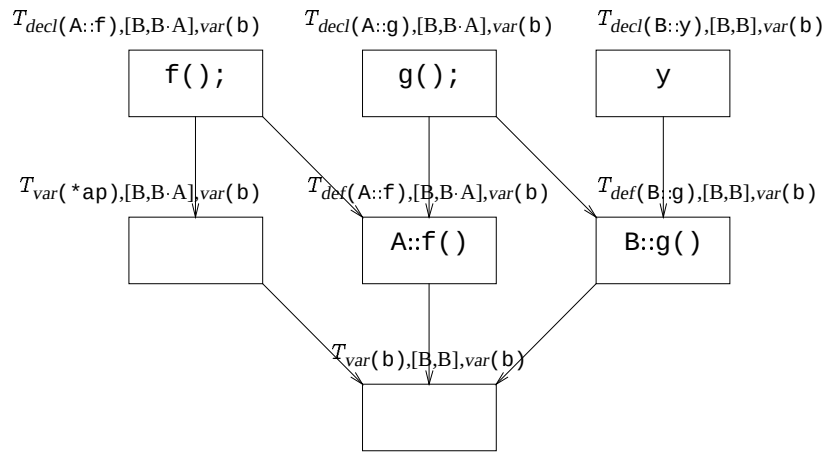


Figure 5: Specialized subobject graph for object b of example program $\mathcal{P}_2$ of Figure 2.

Figure 5 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.

4.3 The specialized class hierarchy

We already defined the set of classes of the specialized class hierarchy in Section 4.1. In this section, we complete the construction of the specialized hierarchy by defining the members of these classes, and the inheritance relationships among these classes, using the subobject graph $\langle N, \sqsubset \rangle$ that was constructed in Section 4.2:

1. Class $T_{var(x)}$ does not contain any members.
2. Class $T_{decl(X::m)}$ contains a declaration of member m, similar to the declaration of m in class X of the original hierarchy.
3. Class $T_{def(X::m)}$ contains a definition of member m, similar to the definition of m in class X of the original hierarchy.
4. For two subobjects $n,\ n' \in N$ such that $n \sqsubset n'$, class $ldc(n)$ is an immediate base class of class $ldc(n')$. This inheritance relation is virtual if: (i) there is a node $n_1 \in N$ such that $ldc(n_1) \equiv ldc(n)$, (ii) $n_1 \sqsubset n_2$, for some $n_2 \in N$ with $ldc(n_2) \equiv ldc(n')$, (iii) $n_1 \sqsubset n_3$, for some $n_3 \in N$ such that $n_3 \neq n_2$, and (iv) $n_2 \sqsubset^* n_4$ and $n_3 \sqsubset^* n_4$, for some n_4 in N. Otherwise, the inheritance relation is nonvirtual.

4.4 Updating variable declarations

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 $T_{var(v)}$ to each variable v in *ClassVars*$(\mathcal{P})$, and type $*T_{var(p)}$ to each variable p in *ClassPtrVars*$(\mathcal{P})$ which is not the this pointer of a method. The type of a this pointer is determined by the location of the associated method definition in the hierarchy; no declaration needs to be updated in this case.

4.5 Example

Figure 6 shows the new program and hierarchy that are constructed for program $\mathcal{P}_2$ of Figure 2. The behavior of this program is identical to that of the original program, and the reader may verify that certain 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, and may increase object size). Assuming that the object model of the IBM xlC C++ compiler would be used, object a would now be 52 bytes (was 8), object b would be 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 5.

4.6 Representability issues

There are two kinds of situations in which the specialization algorithm generates inheritance structures that cannot be represented directly in terms of the inheritance mechanisms of $\mathcal{L}$ (or C++).

The first kind of situation that gives rise to an irrepresentable inheritance structure arises in the presence of a set of assignments of the form:

$$x_1 = x_2;\ x_2 = x_3;\ \ldots;\ x_{n-1} = x_n;\ x_n = x_1$$

(the same situation arises in the presence of recursive method calls). Each assignment $x_i = x_{i+1}$ implies that the type of x_i is a base class of the type of x_{i+1}. Since the assignment $x_n = x_1$ implies that the type of x_n is a base of the type of x_1, there will be a cycle in the inheritance graph unless precautions are taken.

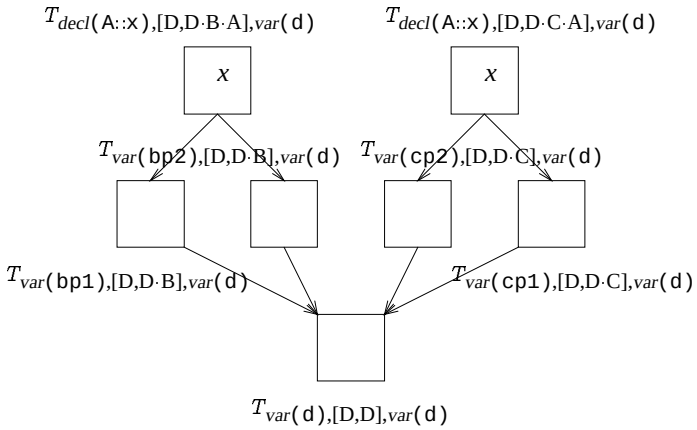
Another situation that leads to irrepresentable inheritance structures has to do with the fact that the specialization may effectively transform any replicated subobject into a shared subobject. In the presence of nonvirtual multiple inheritance, this may give rise to multiple, distinct shared subobjects of the same type—something that cannot be expressed in terms of the inheritance mechanisms of $\mathcal{L}$ (or C++). Figure 7(a) shows a program that illustrates this situation. Note that the specialized subobject graph for this program, shown in Figure 7(b), contains two distinct nodes T_{decl}(A::x),[D,D·B·A],var(d) and T_{decl}(A::x),[D,D·C·A],var(d)

```
class T_decl(A::x) {
  int x;
};

class T_decl(A::f) {
  virtual int f();
};

class T_decl(A::g) {
  virtual int g();
};

class T_decl(B::y) {
  int y;
};

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

class T_decl(C::z) {
  int z;
};

class T_def(A::g) :  T_decl(A::x), virtual T_decl(A::g) {
  virtual int g(){ return x; };
};

class T_def(A::f) :  virtual T_decl(A::g),
                     virtual T_decl(A::f) {
  virtual int f(){ return g(); };
};

class T_def(B::g) :  T_decl(B::y),
                     virtual T_decl(A::g),
                     virtual T_decl(B::g) {
  virtual int g(){ return y; };
};

class T_def(C::f) :  T_decl(C::z),
                     virtual T_decl(A::f),
                     virtual T_decl(B::g) {
  virtual int f(){ return g() + z; };
};

class T_var(*ap) :  virtual T_decl(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();
}
```

Figure 6: Result of Phase III for program $\mathcal{P}_2$ of Figure 2.

```
class A { int x; };
class B : A { ··· };
class C : A { ··· };
class D : B, C { ··· };

void main(){
  D d;
  B *bp1, *bp2;
  C *cp1, *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)



(b)

Figure 7: **(a)** Example program. **(b)** Generated specialized subobject graph.

with the same least derived class: $T_{decl(\texttt{A::x})}$. Unless countermeasures are taken, the algorithm of Section 4.3 will construct a specialized hierarchy where class $T_{decl(\texttt{A::x})}$ is a shared base class of classes $T_{var(\texttt{*bp1})}$, $T_{var(\texttt{*bp2})}$, $T_{var(\texttt{*cp1})}$, and $T_{var(\texttt{*cp2})}$. However, this implies that there would be a single subobject of type $T_{decl(\texttt{A::x})}$ inside an object of type $T_{var(\texttt{d})}$, and therefore, program behavior would not be preserved: the specialized version of the program would compute the value 50 for variable `result`, whereas the original program would compute the value 20.

Lack of space prevents us from presenting our approach to these problems in detail, but the essence of our solution consists of merging the types of variables in such a way that no irrepresentable structures can arise (this mechanism will be discussed in [24]). To this end, we add a phase to the algorithm where constraint elements are partitioned into equivalence classes; elements that occur in the same equivalence class must have the same type. Roughly speaking, two variables x and y occur in the same equivalence class if (i) $\mathcal{P}$ contains assignments such that x is transitively assigned to y, and vice versa, or (ii) if a (member in) a replicated class A (see Appendix A) is accessed from both x and y. In addition, the type constraints of Section 3 need to be modified slightly to take equivalence classes into account.

Representability issues such as the ones discussed above become much more prominent for object-oriented languages with more limited facilities for expressing inheritance than $\mathcal{L}$, such as Java [10]. The inheritance structures that result from specialization are derived from the member access and assignment operations that occur in a program, and do not "naturally" conform to a language's limitations on inheritance. This implies that for a language without, e.g., support for multiple inheritance, classes in the specialized inheritance structures must be merged until all use of multiple inheritance is eliminated.

4.7 Justification

In this section, we argue that class hierarchy specialization is a semantics-preserving program transformation. Due to space limitations, we only state the essential properties here, without proofs (details may be found in [24]).

In order to show that behavior is preserved, we need to reason about "corresponding" lookup and typecast operations in the original and the specialized subobject graphs. To this end, we use the ψ mapping that was introduced in Section 4.2: we will say that a subobject n in N *corresponds to* a subobject $\sigma \in \Sigma(\gamma)$ if $\psi(n) \equiv \sigma$.

The following theorem states that assignment behavior is preserved. Informally, the theorem states that if (i) σ and n are corresponding subobjects in $\Sigma(\gamma)$ and N, respectively, (ii) the least derived class of σ and n correspond to the type of object y, and (iii) there is an assignment $\langle x, y\rangle \in \mathit{Assignments}(\mathcal{P})$, then execution of the assignment will result in the selection of corresponding subobjects in $\Sigma(\gamma)$ and N.

Theorem 4.7 (preservation of assignment behavior) *Let $\mathcal{P}$ be a program with initial subobject graph $\langle\Sigma(\gamma), `<'\rangle$ and specialized subobject graph $\langle N, `\sqsubset'\rangle$. Let n be a subobject in N such that $\mathit{ldc}(n) = T_{[\mathit{var}(y)]}$, and let $\langle x, y\rangle \in \mathit{Assignments}(\mathcal{P})$. Then:*

$$\phi(\mathit{typecast}(n, T_{[\mathit{var}(x)]}, `\sqsubset')) = \mathit{typecast}(\phi(n), \mathit{TypeOf}(\mathcal{P}, x), `<')$$

Here, $[\mathit{var}(x)]$ denotes the equivalence class for variable x, as was alluded to in Section 4.6.

The following theorem states that lookup behavior is preserved. Informally, the theorem states that if (i) σ and n are corresponding subobjects in $\Sigma(\gamma)$ and N, respectively, (ii) the least derived class of σ and n correspond to the type of object y, and (iii) member m is accessed from object y, then execution of the lookup will result in the selection of corresponding subobjects in $\Sigma(\gamma)$ and N.

Theorem 4.8 (preservation of lookup behavior) *Let $\mathcal{P}$ be a program with initial subobject graph $\langle\Sigma(\gamma), `<'\rangle$ and specialized subobject graph $\langle N, `\sqsubset'\rangle$. Let n be a subobject in N such that $\mathit{ldc}(n) = T_{[\mathit{var}(y)]}$, and let $\langle y, m\rangle \in \mathit{MemberAccess}(\mathcal{P})$. Then:*

$$\phi(\mathit{lookup}(n, m, `\sqsubset')) = \mathit{lookup}(\phi(n), m, `<')$$

5 Phase IV: Simplification

Phase IV of the algorithm consists of the application of a set of simple transformation rules to the specialized class hierarchy[7]. The effect of these transformations is a simplification of the inheritance structure, which may result in a reduction in the number of compiler-generated fields in objects, and hence in a reduction of 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.

In the rules below, the "merging" of two classes X and Y (where X is a base class of Y) involves the following steps:

1. A new class Z is created that (virtually, nonvirtually) inherits from all (virtual, nonvirtual) base classes of X and Y, and that contains all members of X and Y,
2. 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 Y and Z' is virtual; otherwise it is nonvirtual.
3. All variables of type X and Y are given type Z, and all variables of type $X*$ and $Y*$ are given type $Z*$.
4. Classes X and Y are removed from the hierarchy.

The conditions of the rules presented below are designed in such a way that classes are only merged if no object in the program becomes larger (i.e., contains more members) as a result of the merge.

Rule 1: Merge a base class X with a derived class Y if:

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. 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.

[7] Alternatively, the set of type constraints could be simplified before the specialized class hierarchy is generated. However, since we believe that these transformations are of interest in their own right (e.g., as an optimization performed subsequent to class hierarchy slicing [23]), we have chosen to present them as general transformations that may be applied to *any* class hierarchy.

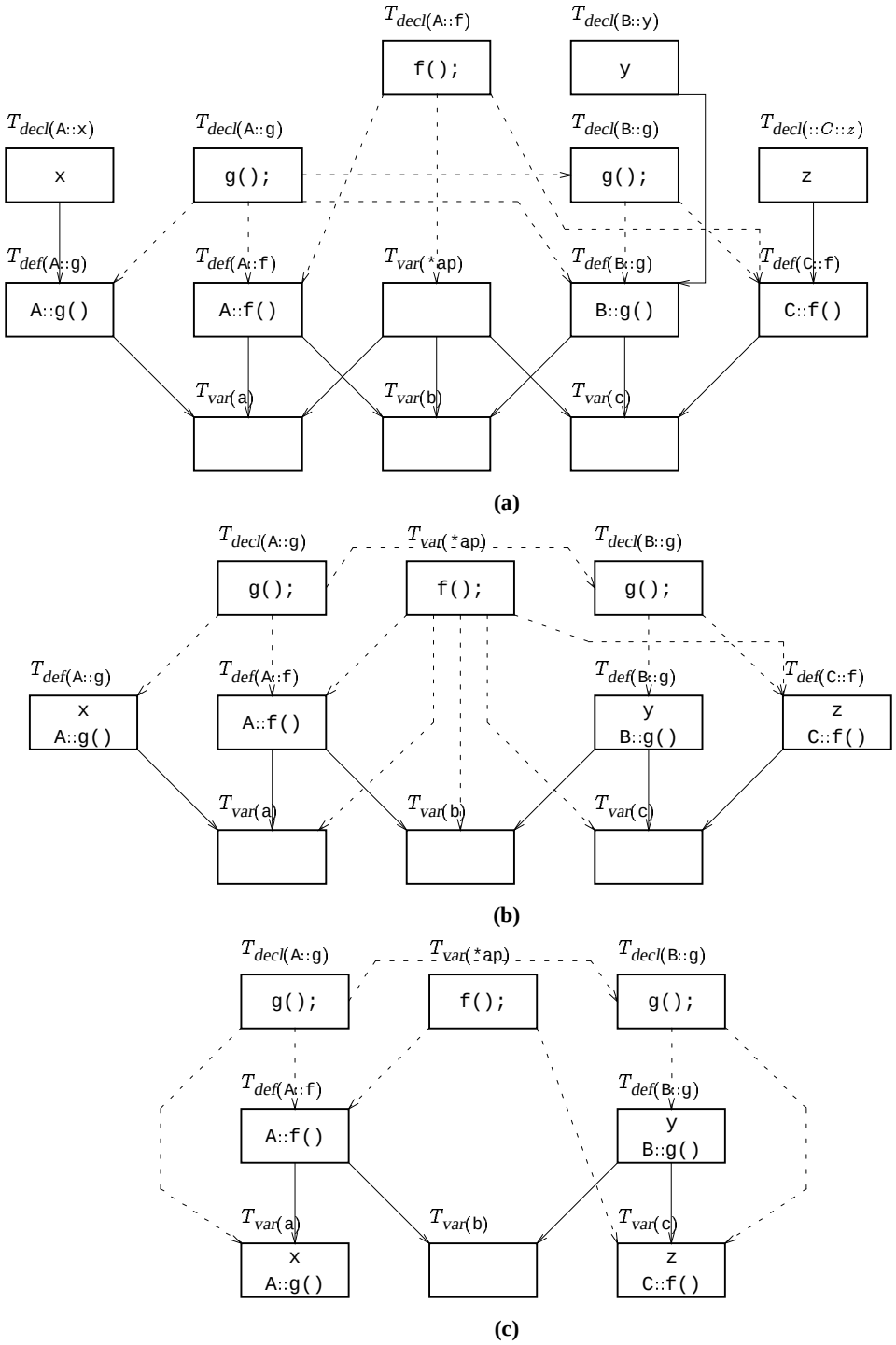


Figure 8: Illustration of the class hierarchies that result from applying the simplification rules of Section 5 to the specialized class hierarchy of Figure 6. In the figure, boxes indicate classes, solid arrows indicate nonvirtual (replicated) inheritance, and dashed arrows indicate virtual (shared) 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`).

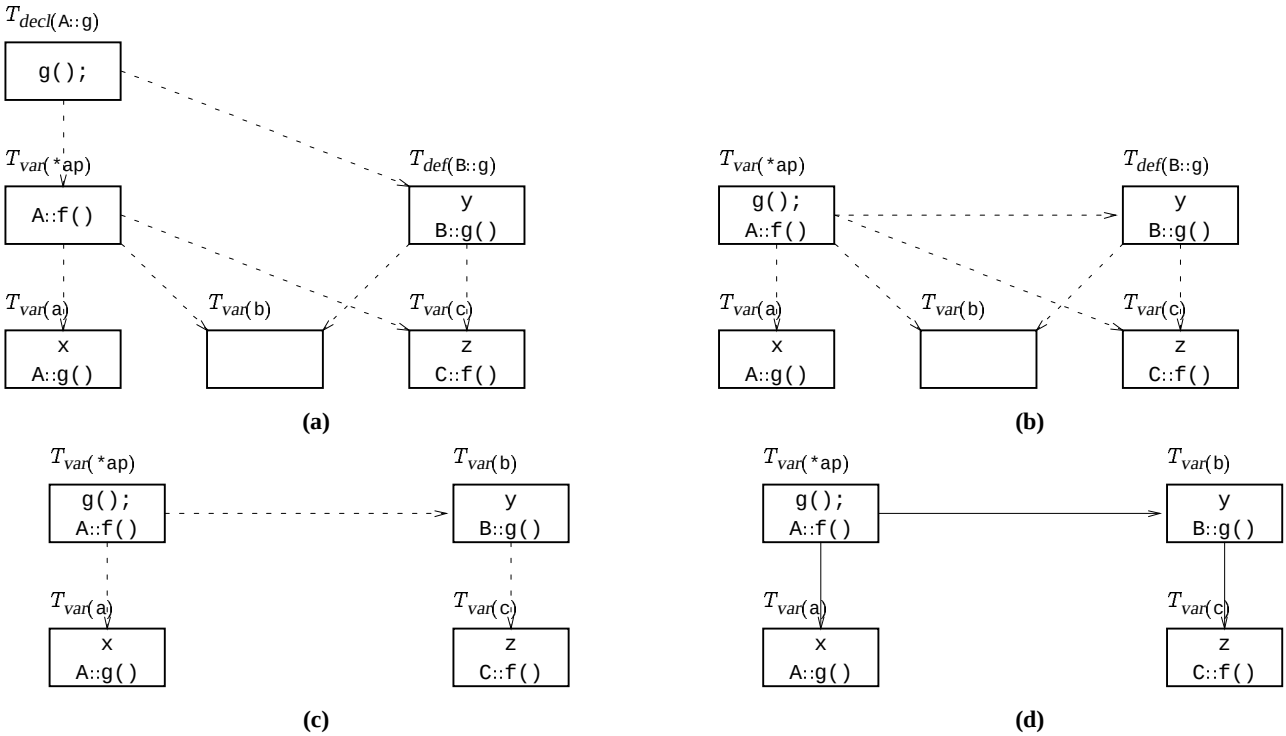


Figure 9: Illustration of the class hierarchies that result from applying the simplification rules of Section 5 to the specialized class hierarchy of Figure 6 (continuation of Figure 8).

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, 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)–(3) of Rule 1 ensure that the class hierarchy is still valid after the merge, whereas conditions (4) and (5) ensure that no object becomes larger as a result of the merge.

Rule 2: Remove the virtual inheritance relation between classes X and Z when:

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

Rule 3: Replace virtual inheritance between classes X and Y by nonvirtual inheritance 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'.

As an example, we will study the simplification of the specialized class hierarchy that was shown in Figure 6. Figure 8(a) depicts this class hierarchy before any simplifications have been performed. In Figure 8(b), the class hierarchy is shown after merging class $T_{decl(\texttt{A::x})}$ with class $T_{def(\texttt{A::g})}$ (Rule 1), merging class $T_{decl(\texttt{B::y})}$ with class $T_{def(\texttt{B::g})}$ (Rule 1), merging class $T_{decl(\texttt{C::z})}$ with class $T_{def(\texttt{C::f})}$ (Rule 1), eliminating the inheritance relation between class $T_{decl(\texttt{A::g})}$ and class $T_{def(\texttt{B::g})}$ (Rule 2), and merging class $T_{decl(\texttt{A::f})}$ with class $T_{var(\texttt{*ap})}$ (Rule 1). Figure 8(c) depicts the class hierarchy after eliminating the inheritance relation between class $T_{var(\texttt{*ap})}$ and class $T_{var(\texttt{a})}$ (Rule 2), eliminating the inheritance relation between class $T_{var(\texttt{*ap})}$ and class $T_{var(\texttt{b})}$ (Rule 2), eliminating the inheritance relation between class $T_{var(\texttt{*ap})}$ and class $T_{var(\texttt{c})}$ (Rule 2), merging class $T_{def(\texttt{A::g})}$ with class $T_{var(\texttt{a})}$ (Rule 1), and merging class $T_{def(\texttt{C::f})}$ with class $T_{var(\texttt{c})}$ (Rule 1).

Figure 9(a) shows the hierarchy after eliminating the inheritance relation between class $T_{decl(\texttt{A::g})}$ and class $T_{var(\texttt{a})}$ (Rule 2), eliminating the inheritance relation between class $T_{decl(\texttt{B::g})}$ and class $T_{var(\texttt{c})}$ (Rule 2), merging class $T_{decl(\texttt{B::g})}$ and class $T_{def(\texttt{B::g})}$ (Rule 1), and merging class $T_{var(\texttt{*ap})}$ and class $T_{def(\texttt{A::f})}$ (Rule 1). Figure 9(b) shows the hierarchy after merging class $T_{decl(\texttt{A::g})}$ and class $T_{var(\texttt{*ap})}$ (Rule 1). Figure 9(c) shows the hierarchy after eliminating the inheritance relation between class $T_{var(\texttt{*ap})}$ and class $T_{var(\texttt{b})}$ (Rule 2), eliminating the inheritance relation between class $T_{var(\texttt{*ap})}$ and class $T_{var(\texttt{c})}$ (Rule 2), and merging class $T_{def(\texttt{B::g})}$ and class $T_{var(\texttt{b})}$ (Rule 1). The final result, shown in Figure 9(d) is obtained by replacing all virtual inheritance relations by nonvirtual inheritance relations (three applications of Rule 3). This is the same hierarchy that was shown earlier in Figure 2.

While the simplification rules introduced above are sufficient for the examples presented in this paper, further research is needed to determine if more simplification rules are needed to cover 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 an unused data member is added to a certain class. Other issues related to simplification rules such as proofs of correctness, termination

behavior, and completeness also require further research.

6 Conclusions and Future Work

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 (a feature that increases member access time, and that may increase 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, we believe that specialization may create new opportunities for existing optimizations such as virtual function call resolution.

We have presented our techniques for a small object-oriented language, but our intentions are ultimately to implement class hierarchy specialization for real languages such as C++ and Java. Before this is feasible, a number of language features need to be modeled, including:

- Nested structures, i.e., class members whose type is a (pointer to a) class. Recursive types are an interesting special case of nested structures.
- Features that bypass the standard typing rules such as explicit typecasts, and method calls that use the '::' operator.
- 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 [17], 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.

Handling these, and the other features that we currently do not treat will be a major effort, but we foresee no fundamental obstacles. Since the algorithm only inspects the code, we believe that it can be implemented efficiently. However, an in-depth complexity analysis of the algorithm is future work.

In addition to extending the algorithm to accommodate various language features, we intend to study the following:

- Although the simplification rules of Section 5 are sufficient for our current examples, more simplification rules may be needed in general. We intend to develop a complete system of simplification rules that allows for tradeoffs based on the compiler's object model.
- Since $\mathcal{L}$ has a very rich inheritance structure, expressing the inheritance structures that result from specialization in terms of a valid $\mathcal{L}$-hierarchy is relatively easy. For languages with more restricted forms of inheritance such as Java, this step will require more work.
- The primary application we have in mind for specialization is that of space/time optimization. It would be interesting to investigate whether specialization could be used as a basis for class hierarchy design/maintenance tools.
- In order to make class hierarchy specialization practical for real applications, several pragmatic issues related to separate compilation and the use of class libraries for which only object code is available need to be addressed.

```
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 10: BNF grammar for $\mathcal{L}$.

Acknowledgements

We are grateful to John Field, Yossi Gil, G. Ramalingam, and the anonymous referees for their many invaluable suggestions.

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 a variable. Figure 10 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.

We make two assumptions concerning the member lookup expressions and typecasts in $\mathcal{L}$-programs. If a class hierarchy contains classes X and Y such that a Y-object contains multiple X-subobjects, we call X a *replicated* class. We will assume that:

- If the program contains an (implicit) typecast from a class Z to a replicated class X, then X is an immediate base 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 conform to these assumptions can be trivially transformed into an equivalent $\mathcal{L}$-program that meets our requirements. The reason for imposing these restrictions is to avoid the generation of irrepresentable inheritance structures, as is discussed in Section 4.6.

References

[1] 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.

[2] 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).

[3] 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.

[4] 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).

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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).

[10] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. Addison-Wesley, 1996.

[11] 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).

[12] 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).

[13] OPDYKE, W., AND JOHNSON, R. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference* (1993).

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

[15] PALSBERG, J., AND SCHWARTZBACH, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[16] PANDE, H. D., AND RYDER, B. G. Static type determination and aliasing for C++. Report LCSR-TR-250-A, Rutgers University, October 1995.

[17] 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.

[18] 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).

[19] 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.

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

[21] 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.

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

[23] 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).

[24] TIP, F., AND SWEENEY, P. F. Class hierarchy specialization. Technical report, IBM, 1997. Forthcoming.

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