Slicing Class Hierarchies in C++

Frank Tip Jong-Deok Choi John Field G. Ramalingam

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
{tip, jdchoi, jfield, rama}@watson.ibm.com

Abstract

This paper describes an algorithm for *slicing* class hierarchies in C++ programs. Given a C++ class hierarchy (a collection of C++ classes and inheritance relations among them) and a program P that uses the hierarchy, the algorithm eliminates from the hierarchy those data members, member functions, classes, and inheritance relations that are unnecessary for ensuring that the semantics of P is maintained.

Class slicing is especially useful when the program P is generated from a larger program P' by a *statement slicing* algorithm. Such an algorithm eliminates statements that are irrelevant to a set of slicing *criteria*—program points of particular interest. There has been considerable previous work on statement slicing, and it will not be the concern of this paper. However, the combination of statement slicing and class slicing for C++ has two principal applications: First, class slicing can enhance statement slicing's utility in program debugging and understanding applications, by eliminating both executable *and* declarative program components irrelevant to the slicing criteria. Second, the combination of the two slicing algorithms can be used to decrease the space requirements of programs that do not use all the components of a class hierarchy. Such a situation is particularly common in programs that use class libraries.

Appeared in the Proceedings of the 11th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96), San Jose, October 1996. ACM SIGPLAN Notices 30 (10), pp. 179–197.

1 Introduction

1.1 Overview

Program slicing is a technique for isolating computational threads in programs [30]. A program slice is generally defined as the set of statements that either contribute to, or are affected by, the values computed at some designated point of interest in the program. The combination of a program point and a set of variables of interest at that point is referred to as the *slicing criterion*.

Almost all previous work on slicing has addressed the question of determining which *executable* statements should be included in a slice, i.e., obtaining what we will call a *statement slice*. By contrast, we are concerned in this paper with eliminating unnecessary components from the *declarative* parts of C++ programs. In abstract form, our algorithm takes as input a C++ class hierarchy (a collection of C++ classes and and the inheritance relations among them) and a program that uses the hierarchy. It then eliminates from the hierarchy those data members, member functions, classes, and inheritance relations that are unnecessary for ensuring that the semantics of the program is maintained.

Our algorithm is specifically designed to accommodate the inheritance mechanism of C++. Due to the complexity of the semantics of multiple and virtual (i.e., shared) inheritance in C++, the task of obtaining class slices that are correct (in the sense that they do not alter the behavior of the criteria), yet not excessively conservative (in the sense that they do not include unnecessary class components), is far from trivial. Simple approaches to the problem are likely to suffer from subtle errors, or, if correct, to require excessive con-

servatism. We therefore take a formal approach to the problem that ensures that the semantics of the program are preserved by the slicing operation.

Statement slicing was originally proposed as a way to allow programming tools to assist the programmer in isolating the source of an error, or to focus attention on code that is relevant to a proposed program modification (see [29] for a survey of various slicing techniques and their numerous applications). For C++, our algorithm can complement statement slicing algorithms (e.g., that of Larsen and Harrold [20]) by eliminating irrelevant declarations as well as irrelevant statements.

In addition, our algorithm has the benefit of allowing unused components of classes to be eliminated in applications that do not use those components. This application is similar to the work of Agesen and Ungar [4, 3] for the dynamically typed language Self. However, the complex static inheritance mechanisms of C++ require a different approach.

Although class slicing can be used to eliminate unneeded components from any program, it is of particular value when used in conjunction with statement slicing to eliminate extraneous components from a class library. In this scenario, statement slicing would be used to eliminate library code irrelevant to a particular client application. Class slicing would then be used to eliminate unnecessary data members and inheritance links. The space savings accruing from elimination of such components, especially for programs that make extensive use of libraries, can be considerable.

1.2 Statement Slicing

To understand the traditional notion of statement slicing, consider the example program in Figure 1(a). This program computes the sum and the product of the first `n` integers. Figure 1(b) shows a slice with respect to the value of `prod` at the last line of the program. Observe that the statements that concern the computation of `sum` have been eliminated. Thus a programmer concerned with an error in the computation of `prod` need not consider the omitted statements when tracking down the error's source.

```
#include <iostream.h>

void main(){
  int n;
  cin >> n;
  int sum = 0;
  int prod = 1;
  for (int i = 1;
       i <= n;
       i++){
    sum += i;
    prod *= i;
  }
  cout << sum << endl;
  cout << prod << endl;
}
```

(a)

```
#include <iostream.h>

void main(){
  int n;
  cin >> n;
  [          ]
  int prod = 1;
  for (int i = 1;
       i <= n;
       i++){
    [      ]
    prod *= i;
  }
  [                  ]
  cout << prod << endl;
}
```

(b)

Figure 1: **(a)** Example program. **(b)** Slice of the program w.r.t. the value of `prod` at the last line of the program; constructs irrelevant to the final value of `prod` have been eliminated and replaced by boxes.

1.3 Slicing Class Hierarchies

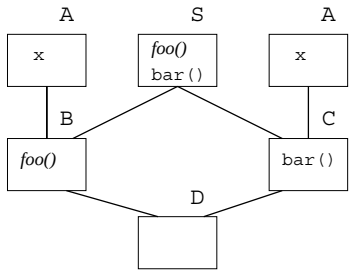
Consider now the more substantial example in Figure 2(a), which shows a class hierarchy that uses both multiple inheritance and virtual inheritance. This example is deliberately very contrived in order to illustrate a number of fine points.

According to the semantics of C++ [14], any object of type `D` contains two distinct subobjects of type `A` (inherited non-virtually through classes `B` and `C`, respectively), but only one subobject of type `S` (because `S` is inherited virtually through classes `B` and `C`). The subobject graph [27, 26] for this class hierarchy is depicted in Figure 2(b). For convenience, we will refer to the two `A` subobjects of `D` as [`D`, `D`·`B`·`A`] and [`D`, `D`·`C`·`A`], respectively. The control flow in the example is slightly counterintuitive. The call to method `bar()` in procedure `main` is resolved to `C::bar()`. The resolution process that is used here, relies on the notion of *dominance*[1]. In this case, both `C::bar()` and `S::bar()` are "visible from" class `D`, and the former dominates the latter. The call to method `foo()` in the body of `C::bar()` is resolved in a similar way. However, due to the fact that `foo()` is virtual, the *run-time* type is used

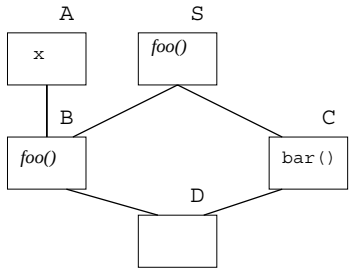
[1] A member name `f` in one subobject `B` dominates a member name `f` in subobject `A` if `A` is a base class subobject of `B` [1, Section 10.2].

```
class A {
public:
  int x;
};

class S {
public:
  virtual void foo();
  void bar();
};

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

class C : public A,
          public virtual S {
public:
  void bar();
};

class D : public B,
          public C {};

void S::foo(){};
void S::bar(){};
void B::foo(){ x++; };
void C::bar(){ this->foo(); };

void main(){
  D d;

  B* b = &d; b->x = 17;
  C* c = &d; c->x = 71;

  d.bar();

  int v = b->x;
  int w = c->x;
}
```

(a)



(b)



(d)

```
class A {
public:
  int x;
};

class S {
public:
  virtual void foo();
  [          ]
};

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

class C : [          ]
          public virtual S {
public:
  void bar();
};

class D : public B,
          public C {};

void S::foo(){};
[                    ]
void B::foo(){ x++; };
void C::bar(){ this->foo(); };

void main(){
  D d;

  B* b = &d; b->x = 17;
  [          ] [          ]

  d.bar();

  int v = b->x;
  [          ]
}
```

(c)

Figure 2: **(a)** Example program. **(b)** Subobject graph for the class hierarchy of the program. **(c)** Slice of the example program w.r.t. the final value of v; constructs irrelevant to the final value of v have been omitted and replaced by boxes. **(d)** Subobject graph for the class hierarchy of the slice.

in the resolution process. In this case, the run-time type of the `this` pointer in `C::bar()` is `D`, because `C::bar()` is called for an object of type `D` in procedure `main()`. From class `D`, two `foo()` methods are visible: `S::foo()` and `B::foo()`, where the latter dominates the former. Therefore, `B::foo()` is selected. Consequently, the program computes the value 18 for variable `v`, and the value 71 for variable `w`.

We are interested in determining the slice with respect to the final value of `v` in the main program. The value of `v` is equal to the value of data member `x` accessed via pointer `b`, which is the `x` in the [`D`, `D`·`B`·`A`] subobject. By carefully examining the program we can observe that:

1. The statements `C* c = &d` and `c->x = 71` in procedure `main` that manipulate the `x` in subobject [`D`, `D`·`C`·`A`] are irrelevant for `v`'s final value.

2. Neither `S::bar()` nor `S::foo()` are called, so they are irrelevant to `v`'s final value.

3. Although `S::foo()` is not called, it cannot be eliminated, because the absence of a statically "visible" definition of `foo()` from the body of `C::bar()` would prevent `B::foo()` from being called.

4. The inheritance relation between classes `B` and `S`, while "locally" superfluous, cannot be removed, since it affects the dominance relation between `B::foo()` and `S::foo()`. (Note that when a virtual inheritance relation *can* be removed safely, the representation of the base class can often be optimized by eliminating the need for an indirect access).

5. The instance variable `x` in subobject [`D`, `D`·`C`·`A`] does not affect the final value of `v`.

6. The inheritance relationship between `C` and `A` is not needed for determining the final value of `v`.

Using the algorithm we present in Section 5, the slice shown in Figure 2(c) can be computed. This example should make clear that there are a number of subtle issues that must be considered to ensure that a class slice is correct, yet not unnecessarily conservative.

1.4 Use of Type Inference Algorithms

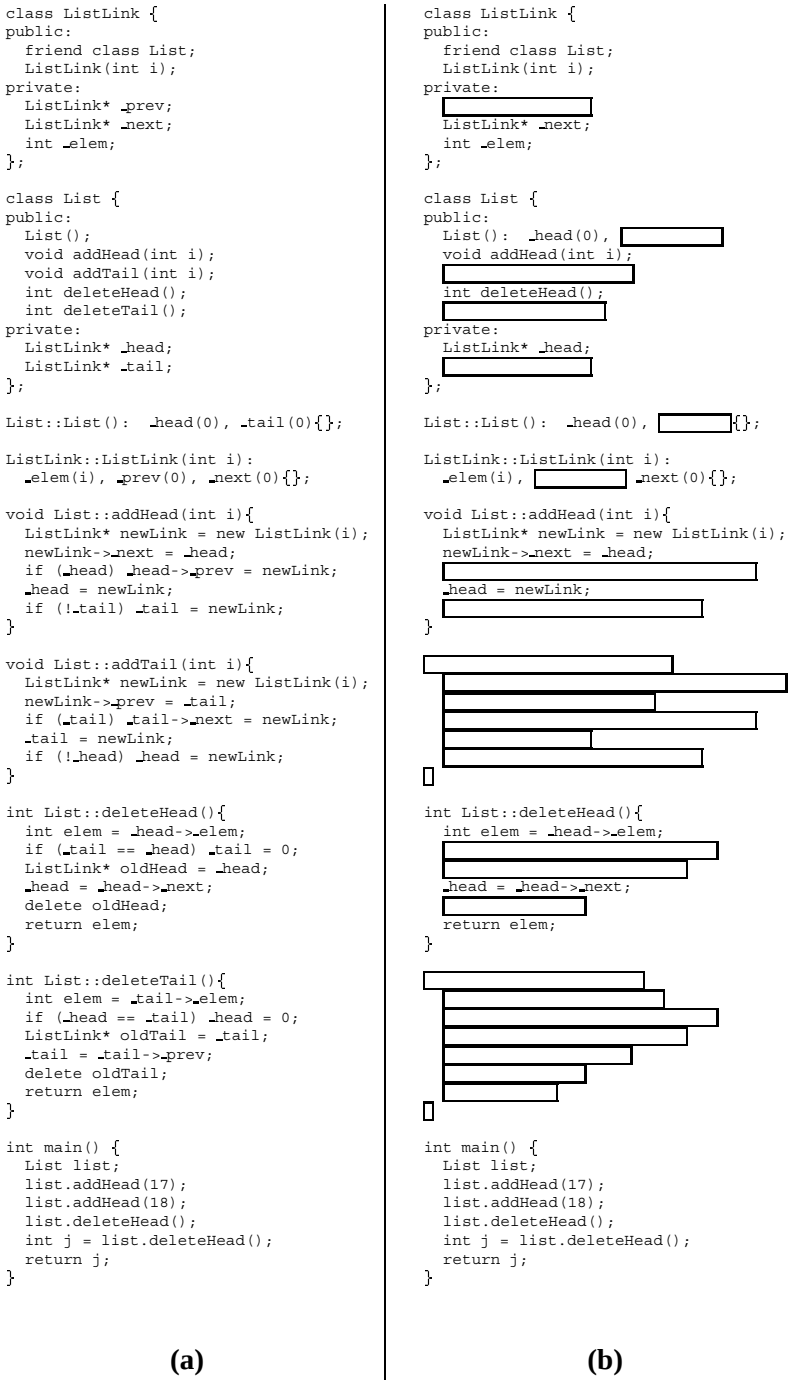
Our algorithm for computing class slices assumes as a prerequisite the existence of a *type inference* algorithm for C++. Such an algorithm is used to determine the set of potential run-time types of all receiver expressions of the form $e.f()$ or $e \rightarrow f()$, where $f()$ is a virtual function. By narrowing the range of potential run-time types for expressions, more accurate class slices may be computed. In the absence of a more sophisticated algorithm, the trivial algorithm which assumes that the receiver may have any subtype of its static type can be used. Section 6.2 reviews type inference algorithms in greater detail.

1.5 Scope of the Paper

In this paper, we will view a class hierarchy somewhat abstractly, treating it as a collection of classes with inheritance relations defined among them. Each class is then modeled as a simple collection of members. This view intentionally elides such issues as name scoping, access rights, and the genesis of the class definitions, e.g., whether they arise via preprocessing mechanisms such as templates and macros. By setting these issues aside, we are able to focus on the core technical problems of class slicing. For many applications of class slicing, particularly space optimization, the abstract view alone is sufficient. Slicing in the presence of preprocessing mechanisms is an important open issue (both for statement and for class slicing), which we leave for future work. Most of the remaining issues not covered by our abstract model are only relevant when a *compilable* program must be computed from a slice; however, a "smart" compiler that could use the results of our slicing algorithm directly would not need to address these problems.

2 Application to Program Optimization

To enable reusability, object oriented programming encourages the use of classes that incorporate a high degree of functionality. While reusability is very desirable, this approach has some drawbacks. Programs that



```
class ListLink {
public:
  friend class List;
  ListLink(int i);
private:
  ListLink* _prev;
  ListLink* _next;
  int _elem;
};

class List {
public:
  List();
  void addHead(int i);
  void addTail(int i);
  int deleteHead();
  int deleteTail();
private:
  ListLink* _head;
  ListLink* _tail;
};

List::List(): _head(0), _tail(0){};

ListLink::ListLink(int i):
  _elem(i), _prev(0), _next(0){};

void List::addHead(int i){
  ListLink* newLink = new ListLink(i);
  newLink->_next = _head;
  if (_head) _head->_prev = newLink;
  _head = newLink;
  if (!_tail) _tail = newLink;
}

void List::addTail(int i){
  ListLink* newLink = new ListLink(i);
  newLink->_prev = _tail;
  if (_tail) _tail->_next = newLink;
  _tail = newLink;
  if (!_head) _head = newLink;
}

int List::deleteHead(){
  int elem = _head->_elem;
  if (_tail == _head) _tail = 0;
  ListLink* oldHead = _head;
  _head = _head->_next;
  delete oldHead;
  return elem;
}

int List::deleteTail(){
  int elem = _tail->_elem;
  if (_head == _tail) _head = 0;
  ListLink* oldTail = _tail;
  _tail = _tail->_prev;
  delete oldTail;
  return elem;
}

int main() {
  List list;
  list.addHead(17);
  list.addHead(18);
  list.deleteHead();
  int j = list.deleteHead();
  return j;
}
```

(a)

```
class ListLink {
public:
  friend class List;
  ListLink(int i);
private:
  [redacted]
  ListLink* _next;
  int _elem;
};

class List {
public:
  List(): _head(0), [redacted]
  void addHead(int i);
  [redacted]
  int deleteHead();
  [redacted]
private:
  ListLink* _head;
  [redacted]
};

List::List(): _head(0), [redacted]{};

ListLink::ListLink(int i):
  _elem(i), [redacted] _next(0){};

void List::addHead(int i){
  ListLink* newLink = new ListLink(i);
  newLink->_next = _head;
  [redacted]
  _head = newLink;
  [redacted]
}

[redacted]

int List::deleteHead(){
  int elem = _head->_elem;
  [redacted]
  _head = _head->_next;
  [redacted]
  return elem;
}

[redacted]

int main() {
  List list;
  list.addHead(17);
  list.addHead(18);
  list.deleteHead();
  int j = list.deleteHead();
  return j;
}
```

(b)

Figure 3: **(a)** Example program. **(b)** Optimized program obtained by slicing with respect to the return expression of `main()`.

use feature-rich classes may pay a penalty for functions they do not use. The more obvious penalty is that of increased code size resulting from linking unused functions [28]. The less obvious penalty is that objects may contain unnecessary data members and subobjects. Larger objects can not only increase the space requirements of the program, but also decrease its execution speed, due to the extra time required for object construction and destruction, and the effects of paging and caching. These considerations force the class designer to choose carefully between features (which enable reusability) and performance.

One solution to this quandary is to design the class with as many features as needed, relying on the *compiler* to optimize applications by specializing classes to the requirements of the program. Statement and class slicing together can perform such an optimization by eliminating not only dead code and dead functions (as some "smart" linkers are also capable of doing), but also unused data members and subobjects.

Figure 3(a) illustrates this process using a program containing a class `List`. `List` defines doubly linked lists together with a number of associated operations; its auxiliary class `ListLink` represents a single list element. `ListLink` consists of a data member `_elem` containing an integer value, and pointers `_prev` and `_next`, which refer to the object's predecessor and successor, respectively. The crux of the example is that the procedure `main()` does not make use of the fact that the list is *doubly* linked.

Figure 3(b) shows an optimized version of the program, where in addition to the dead code in methods `addTail` and `deleteTail`, the `_prev` pointers in class `ListLink` have been removed. The program of Figure 3(b) can be obtained by first computing an interprocedural statement slice with respect to the return expression of procedure `main()`, then determining the components of the class hierarchy that are "needed" by the statements in the executable slice. The statement slice may be computed using a slightly enhanced version of the algorithm of Larsen and Harrold [20] (see Section 6.3 for further comments on statement slicing in C++).

The example above does not illustrate the full power of slicing-based optimization, since the benefits of the latter extend beyond the elimination of unnecessary data members. First, unused subobjects may be eliminated. Second, the removal of virtual inheritance links can enable further optimizations, even when subobjects cannot be eliminated. For instance, eliminating such a link from the dreaded diamond-shaped inheritance graph (created by virtual inheritance and multiple inheritance) breaks the diamond. Most compilers can then generate more efficient code as a result.

If one is interested only in eliminating unnecessary data members and not in these more involved optimizations, it is possible to use techniques that are simpler than the slicing algorithm presented in this paper. For instance, one could simply eliminate data members not mentioned in the statement slice.

3 The Rossie-Friedman Framework

We will begin by formalizing the notion of a class hierarchy, the set of subobjects for a given class in a hierarchy, and the selection of class members. The definition of a slice that will be presented subsequently is expressed in terms of these concepts.

The formalization presented in this section is based on that of Rossie and Friedman [26], although there are some differences. The main differences will be pointed out below.

3.1 Class Hierarchies and Subobjects

Let $\mathcal{C}$ denote the set of class names, and let $\mathcal{M}$ denote the set of member names. We will assume that class names are unique, or that some naming scheme is used if this is not the case. In addition, let $\mathcal{V}$ denote the set $\{$"non-virtual", "virtual"$\}$. A class hierarchy[2] can now be defined as follows:

Definition 3.1 (class hierarchy) *A class hierarchy γ is a quadruple $\langle \hat{C},\ \hat{M},\ \hat{P},\ \hat{S} \rangle$ where:*

[2] The main difference between this notion, and Rossie and Friedman's notion of a class context [26], is that our class hierarchies specify for each member whether it is virtual or not. Our class hierarchies are also subject to additional constraints in order to disallow overriding of a virtual function by a nonvirtual function with the same name.

(1) $\hat{C} \subseteq \mathcal{C}$, (2) $\hat{M} \subseteq \hat{C} \times \mathcal{M} \times \mathcal{V}$, (3) $\hat{S} \subseteq \hat{C} \times \hat{C}$, and (4) $\hat{P} \subseteq \hat{C} \times \hat{C}$

such that: (i) $\hat{S}$ and $\hat{P}$ are irreflexive, (ii) $\hat{S} \cap \hat{P} = \emptyset$, (iii) the transitive closure of $(\hat{S} \cup \hat{P})$ is antisymmetric, (iv) if $\langle X, m, f_1 \rangle \in \hat{M}$ and $\langle X, m, f_2 \rangle \in \hat{M}$ then $f_1 = f_2$, and (v) if $\langle X, m,$ "virtual"$\rangle \in \hat{M}$, $\langle Y, m, f \rangle \in \hat{M}$, and $\langle Y, X \rangle \in (\hat{S} \cup \hat{P})^*$ then $f =$ "virtual".

Given a class hierarchy γ, its components will be denoted by $\hat{C}(\gamma)$, $\hat{M}(\gamma)$, $\hat{S}(\gamma)$, and $\hat{P}(\gamma)$, respectively. The reflexive and transitive closure of $(\hat{S} \cup \hat{P})$ will be denoted by $\hat{I}(\gamma)$.

Here, $\hat{C}$ denotes the subset of class names that are used in this class hierarchy. $\hat{M}$ is a function that maps every class in $\hat{C}$ to a subset of the set of member names, and specifies for every class member if it is virtual or non-virtual; the function is stated in relational form because that will make it easier to specify operations on class hierarchies later. However, in cases where we are not interested in the virtuality of a member, we will often write '$m \in \hat{M}(C)$' instead of '$\langle C, m, v \rangle \in \hat{M}$ for some v'. $\hat{S}$ and $\hat{P}$ are relations indicating the proprietary (i.e., non-virtual) and the shared (i.e., virtual) inheritance relations between classes in C.

Constraints (i)–(v) encode the usual C++ constraints on inheritance relations that: (i) a class cannot be its own base class, (ii) a class cannot be a non-virtual *and* a virtual base class at the same time, (iii) cycles in the inheritance graph are not allowed, (iv) a member cannot be virtual and non-virtual at the same time, and (v) a virtual member cannot be overridden by a non-virtual member with the same name. For practical reasons, we will assume class hierarchies to be finite.

Example 3.2 *For the example in Figure 2, the class hierarchy γ_{ex} is given by:*

$$\begin{aligned}
\hat{C}(\gamma_{ex}) &= \{\, \texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{S} \,\} \\
\hat{M}(\gamma_{ex}) &= \{\ \langle \texttt{A}, \texttt{x}, \text{“non-virtual”} \rangle, \\
&\qquad \langle \texttt{S}, \texttt{foo}, \text{“virtual”} \rangle, \\
&\qquad \langle \texttt{S}, \texttt{bar}, \text{“non-virtual”} \rangle, \\
&\qquad \langle \texttt{B}, \texttt{foo}, \text{“virtual”} \rangle, \\
&\qquad \langle \texttt{C}, \texttt{bar}, \text{“non-virtual”} \rangle \,\} \\
\hat{S}(\gamma_{ex}) &= \{\, \langle \texttt{B}, \texttt{S} \rangle, \langle \texttt{C}, \texttt{S} \rangle \,\} \\
\hat{P}(\gamma_{ex}) &= \{\, \langle \texttt{D}, \texttt{C} \rangle, \langle \texttt{D}, \texttt{B} \rangle, \langle \texttt{B}, \texttt{A} \rangle, \langle \texttt{C}, \texttt{A} \rangle \,\}
\end{aligned}$$

□

Rossie and Friedman use the term *subobject* not to refer to a "part of an object," but rather to denote what is in essence a refined notion of type derived from the combination of the "static" (declared) and "run-time" (as allocated) types of an object. Therefore, when we wish to refer to a "part of an object," we will use the term *subobject instance*.

Intuitively, a subobject identifies the type (i.e., class) C of the "full object instance" in which a subobject instance is embedded as well as the type (class) D of the subobject instance itself. However, defining a subobject as a pair $\langle C, D \rangle$ would be insufficient, because a C object instance may contain more than one subcomponent of type D in the presence of multiple inheritance. We will identify a subobject by a pair $[C, C_1 \cdot \cdots \cdot C_n]$, where C denotes the type of the "full object instance", and $C_1 \cdot \cdots \cdot C_n$ is a sequence of class names encoding the transitive inheritance relation between C_1 to C_n. There are two cases here: For *non-shared subobject instances* we have that $C_1 = C$, and for *shared subobject instances* we have that C_1 is the least derived (smallest) shared base class of C that contains the subobject. This scheme is sufficient because shared base classes are unique. In Definition 3.3 below, the set of subobjects[3] of a class C in a hierarchy γ is defined. We use α to denote a possibly empty sequence of class names $C_1 \cdot \cdots \cdot C_n$.

Definition 3.3 (subobject) *For a class hierarchy γ, and a class $C \in \hat{C}(\gamma)$, the set of subobjects of C, denoted $\Sigma(\gamma, C)$, is inductively defined as follows:*

1. $[C, C] \in \Sigma(\gamma, C)$
2. $[C, D] \in \Sigma(\gamma, C)$ *if there exists an X such that $\langle C, X \rangle \in \hat{I}(\gamma)$, and $\langle X, D \rangle \in \hat{S}(\gamma)$*
3. $[C, \alpha \cdot X \cdot Y] \in \Sigma(\gamma, C)$ *if* $[C, \alpha \cdot X] \in \Sigma(\gamma, C)$ *and $\langle X, Y \rangle \in \hat{P}(\gamma)$.*

We will use $\Sigma(\gamma)$ to denote the set containing all subobjects of all classes in γ.

[3] Our notion of a subobject is equivalent to Rossie and Friedman's [26], although our notation and definition are different.

Example 3.4 *For the class hierarchy γ_{ex} of Example 3.2, it follows from Definition 3.3 that the set of subobjects of class* `D`*,* $\Sigma(\gamma_{ex}, \texttt{D})$ *consists of:*

$\{$ [D, D], [D, D·B], [D, D·C], [D, D·B·A], [D, D·C·A], [D, S] $\}$

□

Example 3.5 *Consider the following fragment of code with respect to the class hierarchy of Figure 2 and Example 3.2:*

```
[1]        D d1, d2;
[2]        B *b1 = &d1;
[3]        B *b2 = &d2;
```

Using informal conventional terminology, one might say that `d1` and `d2` in the above example are (two different) objects of class `D`, while `b1` and `b2` point to the `B` subobject of `d1` and `d2` respectively. In our terminology, we would say that `d1` and `d2` are (two different) instances of the subobject $[D, D]$, while `b1` and `b2` point to (two different) instances of the subobject $[D, D{\cdot}B]$.

We will now define the notions of a subobject's *most derived class* and a subobject's *least derived class*, respectively[4]. Intuitively, a subobject's most derived class is the class with respect to which references to virtual methods are resolved. A subobject's least derived class is the class with respect to which references to data members, and references to non-virtual methods are resolved. In a more operational view, the least derived class of an object corresponds to the *declared type* of an object (i.e., the type as it appears in the object's declaration), whereas the most derived class of an object corresponds to the *run-time type* of the object.

Definition 3.6 (most/least derived class) *Let* $[C, \alpha{\cdot}X] \in \Sigma(\gamma)$*. Then:*

$$\begin{array}{lcl} \mathit{mdc}([C, \alpha{\cdot}X]) & \triangleq & C \\ \mathit{ldc}([C, \alpha{\cdot}X]) & \triangleq & X \end{array}$$

[4] These concepts are equivalent to Rossie and Friedman's *actual class* and *effective class*, respectively.

3.2 Subobject Ordering and Member Lookup

Though defined differently, Definition 3.7 below is equivalent to Rossie and Friedman's partial ordering '$<$' on the subobjects in $\Sigma(\gamma, C)$. This ordering models the effect of hiding, as well as the dominance rule [26].

Definition 3.7 ($<$) *Let γ be a class hierarchy, $C \in \hat{C}(\gamma)$. Then:*

$$\begin{array}{lcll} [C, \alpha] & <_{p,\gamma} & [C, \alpha{\cdot}X] & \\ [C, \alpha{\cdot}X] & <_{s,\gamma} & [C, Y] & \textit{if} \quad \langle X, Y\rangle \in \hat{S}(\gamma) \end{array}$$

Furthermore, let '$<_\gamma$' be the union of '$<_{p,\gamma}$' and '$<_{s,\gamma}$', and let $\leq^*_\gamma$ be the transitive and reflexive closure of $<_\gamma$. Moreover, for any $\Sigma \subseteq \Sigma(\gamma, C)$, let $\min_\gamma(\Sigma)$ denote the least element of Σ w.r.t. $\leq^*_\gamma$, when it exists. We will drop the subscript γ if the class hierarchy is obvious from the context.

Example 3.8 *From Definition 3.7 it follows that we have the following relationships between the subobjects in* $\Sigma(\gamma_{ex}, \texttt{D})$ *(see Example 3.4) of class hierarchy γ_{ex} (see Example 3.2):*

$$\begin{array}{ll} [\texttt{D, D}] < [\texttt{D, D·B}] & [\texttt{D, D}] < [\texttt{D, D·C}] \\ [\texttt{D, D·B}] < [\texttt{D, D·B·A}] & [\texttt{D, D·B}] < [\texttt{D, S}] \\ [\texttt{D, D·C}] < [\texttt{D, D·C·A}] & [\texttt{D, D·C}] < [\texttt{D, S}] \end{array}$$

These relationships were rendered pictorially in Figure 2(b). □

We will now define *VisibleDefs*(γ, σ, m), the set of all subobjects that have a member m that is "visible" in a subobject σ[5].

Definition 3.9 (*VisibleDefs*) *Let γ be a class hierarchy, $C \in \hat{C}(\gamma)$, $\sigma \in \Sigma(\gamma, C)$, and let m be a member name in $\mathcal{M}$. Then the set of all subobjects of C that have a*

[5] Rossie and Friedman do not define *VisibleDefs*, but have a similar notion, **fam**, which does not take hiding into account. This does not make a difference for the purpose of defining member lookup, but *VisibleDefs* will turn out to be essential for the slicing algorithm of Section 5.

member m that is visible in σ is:

$$\mathit{VisibleDefs}(\gamma,\ \sigma,\ m) \triangleq \begin{cases} \{\,\sigma\,\} \\ \textit{if } m \in \hat{M}(\mathit{ldc}(\sigma)) \\ \\ \bigcup_{\sigma<\sigma'} \mathit{VisibleDefs}(\gamma,\ \sigma',\ m) \\ \textit{if } m \notin \hat{M}(\mathit{ldc}(\sigma)) \end{cases}$$

Example 3.10 *The subobjects of class* `D` *in the class hierarchy γ_{ex} of Example 3.2 were shown in Example 3.4. From Definition 3.9, it follows that*

$$\mathit{VisibleDefs}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D}],\ \mathtt{bar}) = \{\,[\mathtt{D},\ \mathtt{S}],\ [\mathtt{D},\ \mathtt{D{\cdot}C}]\,\}$$
$$\mathit{VisibleDefs}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D{\cdot}C}],\ \mathtt{foo}) = \{\,[\mathtt{D},\ \mathtt{S}]\,\}$$
$$\mathit{VisibleDefs}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D{\cdot}B}],\ \mathtt{x}) = \{\,[\mathtt{D},\ \mathtt{D{\cdot}B{\cdot}A}]\,\}$$

□

The key property of a class hierarchy slice that will be presented in Section 4 is *preservation of member selection*. Informally stated, this means that a component of the class hierarchy is irrelevant so long as removing that component does not affect the selection of members (i.e., data members and virtual and non-virtual methods). In this section, we will formalize the process of member selection.

Using Definitions 3.7 and 3.9, we can now formally define which subobject is selected for a static or dynamic access to a member m in a subobject σ. The functions *static-lookup* and *dynamic-lookup* defined below use *VisibleDefs* and the subobject ordering to determine which subobject contains the member that is accessed. Functions *static-lookup* and *dynamic-lookup* are both defined as a mapping from subobjects to subobjects.

Definition 3.11 (static lookup and dynamic lookup) *Let γ be a class hierarchy, $C \in \hat{C}(\gamma)$, $\sigma \in \Sigma(\gamma, C)$, and let $m \in \mathcal{M}$ be the name of a member. Then:*

$$\mathit{static\text{-}lookup}(\gamma,\ \sigma,\ m) \triangleq \min(\mathit{VisibleDefs}(\gamma,\ \sigma,\ m))$$
$$\mathit{dynamic\text{-}lookup}(\gamma,\ \sigma,\ m) \triangleq \min(\mathit{VisibleDefs}(\gamma,\ [\mathit{mdc}(\sigma),\ \mathit{mdc}(\sigma)],\ \mathrm{m}))$$

The lookup is undefined in cases where a minimum element w.r.t. $\leq^*$ does not exist and this denotes that the reference is ambiguous.

Definition 3.11 defines the selection of a subobject given a member m and a subobject σ. However, the question whether the lookup for m should be static or a dynamic is encoded in the $\hat{M}$ component of the class hierarchy, and depends on m and the static class of subobject σ. In function *lookup*[6] (Definition 3.12 below), a static lookup for member m is performed first. Then, the virtuality of m is determined, and a dynamic lookup is performed if m is found to be virtual. This "double" lookup scheme is designed in such a way to simplify the subsequent definitions and descriptions of algorithms; there are obviously more efficient ways of implementing lookups for virtual members.

Definition 3.12 (*lookup*) *Let γ be a class hierarchy, $C \in \hat{C}(\gamma)$, $\sigma \in \Sigma(\gamma, C)$, and let $m \in \mathcal{M}$ be the name of a member. Let $\sigma' = \mathit{static\text{-}lookup}(\gamma, \sigma, m)$, and let $\langle \mathit{ldc}(\sigma'), m, v\rangle \in \hat{M}(\gamma)$. Then:*

$$\mathit{lookup}(\gamma,\ \sigma,\ m) \triangleq \begin{cases} \sigma' \\ \text{if } v = \text{“non-virtual”} \\ \\ \mathit{dynamic\text{-}lookup}(\gamma,\ \sigma,\ m) \\ \text{if } v = \text{“virtual”} \end{cases}$$

Example 3.13 *From Example 3.2 and Definition 3.12, we can obtain the following for the call to* `bar()` *in procedure* `main()` *of the program of Figure 2.*

$$\begin{aligned} &\mathit{static\text{-}lookup}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D}],\ \mathtt{bar}) = \\ &\min(\mathit{VisibleDefs}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D}],\ \mathtt{bar})) = \\ &\min(\{\,[\mathtt{D},\ \mathtt{S}],\ [\mathtt{D},\ \mathtt{D{\cdot}C}]\,\}) = \\ &[\mathtt{D},\ \mathtt{D{\cdot}C}] \end{aligned}$$

Because of $\langle$`C`, `bar`, “non-virtual”$\rangle \in \hat{M}(\gamma_{ex})$ *we have that*

$$\begin{aligned} &\mathit{lookup}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D}],\ \mathtt{bar}) = \\ &\mathit{static\text{-}lookup}(\gamma_{ex},\ [\mathtt{D},\ \mathtt{D}],\ \mathtt{bar}) = \\ &[\mathtt{D},\ \mathtt{D{\cdot}C}] \end{aligned}$$

[6] *static-lookup* and *dynamic-lookup* are equivalent to Rossie and Friedman's **stat** and **dyn**, though defined differently. In [26], Rossie and Friedman do not address the problem of determining whether a lookup should be static or dynamic. In particular, they do not define a function *lookup* similar to Definition 3.12 below.

For the call to `foo()` *in* `C::bar()` *we have the following:*

$$\begin{aligned}&\textit{static-lookup}(\gamma_{ex}, [\texttt{D}, \texttt{D·C}], \texttt{foo}) =\\ &\min(\textit{VisibleDefs}(\gamma, [\texttt{D}, \texttt{D·C}], \texttt{foo})) =\\ &\min(\{ [\texttt{D}, \texttt{S}] \}) =\\ &[\texttt{D}, \texttt{S}]\end{aligned}$$

Since we have that $\langle \texttt{S}, \texttt{foo}, \text{“virtual”} \rangle \in \hat{M}(\gamma_{ex})$ *we have that*

$$\begin{aligned}&\textit{lookup}(\gamma_{ex}, [\texttt{D}, \texttt{D·C}], \texttt{foo}) =\\ &\textit{dynamic-lookup}(\gamma_{ex}, [\texttt{D}, \texttt{D·C}], \texttt{foo}) =\\ &\min(\textit{VisibleDefs}(\gamma, [\texttt{D}, \texttt{D}], \texttt{foo})) =\\ &\min(\{ [\texttt{D}, \texttt{S}], [\texttt{D}, \texttt{D·B}] \}) =\\ &[\texttt{D}, \texttt{D·B}]\end{aligned}$$

□

4 Class Hierarchy Slices

4.1 Objective

We may view a C++ program as a pair (γ, S), where γ is the declarative part, i.e., a class hierarchy and S is the non-declarative, or executable, part. Previous work on slicing can be adapted to compute slices of the executable part of a C++ program (see Section 6.3): that is, given a program (γ, S) and a slicing criterion, one can compute a *statement* slice (γ, S') which has the same execution behavior as the original program with respect to the criterion.

The goal of this work is to compute *class hierarchy slices*: given a program (γ, S') (which may be, but is not necessarily, the output of a statement slicing algorithm), we would like to compute a slice (γ', S') that has the same execution behavior as (γ, S'), where γ' is a subhierarchy (see Definition 4.3 below) of γ. In particular, we would like γ' to consist only of the parts of γ that are necessary to ensure that $\Sigma(\gamma')$ includes all subobjects that may be “instantiated” during execution of S', and that $\hat{M}(\gamma')$ includes all members that may be used during the execution of the S'. However, we also need to ensure that replacing γ by γ' does not change the execution behavior of S'. The primary component of the program execution behavior that depends on the class hierarchy is the member lookup operation. Hence, the class hierarchy slice should preserve the result of any member lookup that may be performed during the execution of the S'. We formalize these requirements below.

Definition 4.1 (subobject instantiation) *A subobject σ is said to be instantiated during program execution when a pointer or reference to an instance of σ is generated.*

Example 4.2 *Consider the code in Example 3.5. Line [1] instantiates the subobject $[D, D]$, while lines [2] and [3] instantiate the subobject $[D, D{\cdot}B]$. Further, the subobject $[D, D{\cdot}C]$ is not instantiated by this code fragment (even though the $[D, D]$ subobject instances created in line [1] contain instances of $[D, D{\cdot}C]$).*

Definition 4.3 (subhierarchy) *Let γ and γ' be class hierarchies. Then γ' is a subhierarchy of γ ($\gamma' \sqsubseteq \gamma$) iff: (i) $\hat{C}(\gamma') \subseteq \hat{C}(\gamma)$, (ii) $\hat{M}(\gamma') \subseteq \hat{M}(\gamma)$, (iii) $\hat{P}(\gamma') \subseteq \hat{P}(\gamma)$, and (iv) $\hat{S}(\gamma') \subseteq \hat{S}(\gamma)$.*

Example 4.4 *Let γ'_{ex} be a class hierarchy, where:*

$$\begin{aligned}\hat{C}(\gamma'_{ex}) &= \{ \texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{S} \}\\ \hat{M}(\gamma'_{ex}) &= \{ \langle \texttt{A}, \texttt{x}, \text{“non-virtual”} \rangle,\\ &\qquad \langle \texttt{S}, \texttt{foo}, \text{“virtual”} \rangle,\\ &\qquad \langle \texttt{B}, \texttt{foo}, \text{“virtual”} \rangle,\\ &\qquad \langle \texttt{C}, \texttt{bar}, \text{“non-virtual”} \rangle \}\\ \hat{S}(\gamma'_{ex}) &= \{ \langle \texttt{B}, \texttt{S} \rangle, \langle \texttt{C}, \texttt{S} \rangle \}\\ \hat{P}(\gamma'_{ex}) &= \{ \langle \texttt{D}, \texttt{C} \rangle, \langle \texttt{D}, \texttt{B} \rangle, \langle \texttt{B}, \texttt{A} \rangle \}\end{aligned}$$

Then from Definition 4.3 it follows that γ'_{ex} is a subhierarchy of the class hierarchy γ_{ex} that was defined in Example 3.2. □

Definition 4.5 (class hierarchy slice) *Let (γ, S') be a program. A subhierarchy γ' of γ is said to be a class hierarchy slice of (γ, S') if*

1. *$\Sigma(\gamma')$ includes every subobject σ that may be instantiated during execution of S'.*
2. *γ' preserves the lookup of any member m in any subobject σ that may be performed during execution of S': that is,*

$$\textit{lookup}(\gamma', \sigma, m) = \textit{lookup}(\gamma, \sigma, m)$$

4.2 Member Lookup Slices

We now introduce some terminology and notation that will be useful in presenting our algorithm. We define a *member lookup slicing criterion* to be a pair $\langle \sigma, m \rangle$ where m is the name of a member, and σ the subobject associated with the receiver expression e with respect to which m is referenced or called.

Definition 4.6 (member lookup slicing criterion) *For a given class hierarchy γ, a member lookup slicing criterion is defined as a pair $\langle \sigma, m \rangle$, where $\sigma \in \Sigma(\gamma)$ and $m \in \hat{M}(\gamma)$.*

We define a union operator on class hierarchies as below. (Note that this operator is only defined on hierarchies that have a common superhierarchy.)

Definition 4.7 ($\cup$) *Let $\gamma' = \langle \hat{C}(\gamma'), \hat{M}(\gamma'), \hat{S}(\gamma'), \hat{P}(\gamma') \rangle$ and $\gamma'' = \langle \hat{C}(\gamma''), \hat{M}(\gamma''), \hat{S}(\gamma''), \hat{P}(\gamma'') \rangle$ be subhierarchies of a class hierarchy γ. Then:*

$$\gamma' \cup \gamma'' \triangleq \langle\ \hat{C}(\gamma') \cup \hat{C}(\gamma''),\ \hat{M}(\gamma') \cup \hat{M}(\gamma''),\ \hat{S}(\gamma') \cup \hat{S}(\gamma''),\ \hat{P}(\gamma') \cup \hat{P}(\gamma'')\ \rangle$$

Our basic approach is to identify the subhierarchies of γ that are necessary to satisfy various criteria and to take their union. In particular, we satisfy condition (2) of Definition 4.5 by identifying the set of all member lookups $\langle \sigma, m \rangle$ that may be performed, and by computing a slice of γ for each lookup, and by taking the union of all these slices.

What should a slice with respect to member lookup $\langle \sigma, m \rangle$ be?

Let γ'' and γ be class hierarchies such that $\gamma'' \sqsubseteq \gamma$ and $\sigma \in \Sigma(\gamma'')$. Let us call γ'' a *weak slice* of γ with respect to the member lookup criterion $\langle \sigma, m \rangle$ if

$$\mathit{lookup}(\gamma'', \sigma, m) = \mathit{lookup}(\gamma, \sigma, m)$$

Although a weak slice S with respect to a criterion C captures the essential property of preserving lookup behavior for a criterion C, it is not very useful because it lacks a desirable compositional property: If S_1 and S_2 are weak slices with respect to criteria C_1 and C_2, respectively, then $S_1 \cup S_2$ is not necessarily a weak slice with respect to criterion C_1 (or C_2). This can be seen from the example of Figure 4. This leads to the following definition.

Definition 4.8 (slice) *Let γ'' and γ be class hierarchies such that $\gamma'' \sqsubseteq \gamma$. γ'' is said to be a member lookup slice of γ with respect to criterion $\langle \sigma, m \rangle$ if for all γ' such that $\gamma'' \sqsubseteq \gamma' \sqsubseteq \gamma$ and $\sigma \in \Sigma(\gamma')$ we have that*

$$\mathit{lookup}(\gamma', \sigma, m) = \mathit{lookup}(\gamma, \sigma, m)$$

We may define the concepts of a *static lookup slice* and *dynamic lookup slice* analogously by replacing "*lookup*" in the above definition by "*static-lookup*" or "*dynamic-lookup*" respectively.

Example 4.9 *We will study the member references* `d.bar()` *and* `this->foo()` *in the program and slice of Figure 2. The hierarchies γ_{ex} and γ'_{ex} for the program and the slice were shown in Examples 3.2 and 4.4, respectively.*

For call `d.bar()`*, we assume that subobject $\sigma =$ [D, D] $\in \Sigma(\gamma_{ex}, \texttt{D})$ is associated with expression* `d`*. Since we have that*

$$\mathit{lookup}(\gamma_{ex}, [\texttt{D}, \texttt{D}], \texttt{bar}) =$$
$$\mathit{lookup}(\gamma'_{ex}, [\texttt{D}, \texttt{D}], \texttt{bar}) = [\texttt{D}, \texttt{D·C}]$$

γ'_{ex} is a weak slice of γ_{ex} with respect to criterion $\langle$[D, D], bar$\rangle$.

For call `this->foo()`*, suppose that subobject [D, D·C] $\in \Sigma(\gamma_{ex}, \texttt{D})$ is associated with expression* `this`*. Since we have that*

$$\mathit{lookup}(\gamma_{ex}, [\texttt{D}, \texttt{D·C}], \texttt{foo}) =$$
$$\mathit{lookup}(\gamma'_{ex}, [\texttt{D}, \texttt{D·C}], \texttt{foo}) = \quad [\texttt{D}, \texttt{D·B}]$$

γ'_{ex} is a weak slice of γ_{ex} with respect to criterion $\langle$[D, D·C], foo$\rangle$.

The reader may verify that γ'_{ex} is also a slice w.r.t. each of the criteria mentioned above. □

The following lemma states that the slices of Definition 4.8 may be composed with the union operation of Definition 4.7. This property is very important because it implies that a slicing algorithm can compute member lookup slices element-wise, and safely compute their union without affecting the result of any of the lookups. As the example of Figure 4 shows, there is no analogue of this theorem for weak slices.

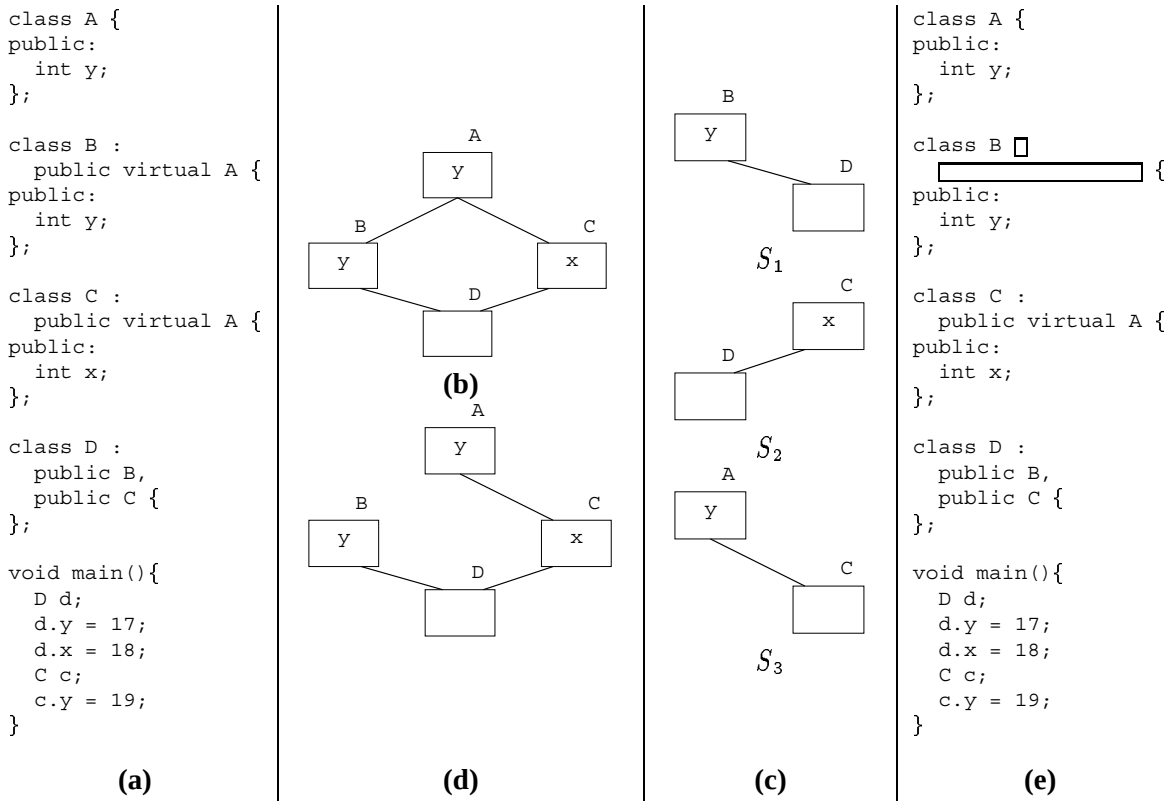


Figure 4: **(a)** Example program. **(b)** Pictorial view of the class hierarchy of **(a)**. **(c)** Pictorial view of three weak slices S_1, S_2, and S_3 w.r.t. criteria $\langle [D,\ D],\ y\rangle$, $\langle [D,\ D],\ x\rangle$, and $\langle [C,\ C],\ y\rangle$. **(d)** Pictorial view of the union $S_1 \cup S_2 \cup S_3$ of the weak slices of **(c)**. **(e)** Program corresponding to the **(d)**. Observe that the member-selection expression `d.y` is ambiguous. This implies that lookup-behavior is not preserved, and therefore that $S_1 \cup S_2 \cup S_3$ is *not* a weak slice w.r.t. $\langle [D,\ D],\ y\rangle$.

Lemma 4.10 *Let γ be a class hierarchy, let γ_1 be a slice of γ w.r.t. criterion S_1, and let γ_2 be a slice of γ w.r.t. criterion S_2. Then $(\gamma_1 \cup \gamma_2)$ is a slice w.r.t. S_1, and $(\gamma_1 \cup \gamma_2)$ is slice w.r.t. criterion S_2.*

Proof. Follows directly from Definition 4.8. □

5 Computing Class Hierarchy Slices

5.1 The Algorithm

In Section 4 we presented a formal definition of what constitutes a slice. In this section we present an algorithm for slicing a class hierarchy (see Figure 5). The algorithm is defined as a function *Slice* that takes a program P as an argument and returns a class hierarchy slice of P.

The first step in computing the class hierarchy slice is to identify all subobjects that may be instantiated during the execution of the statement slice.

Objects may be created in a C++ program through the use of various program constructs such as the new operator (e.g., "$new\ X$") and variable definitions (e.g., "$X\ x$;"). By scanning P for all object creation constructs[7], we identify the set of all "full" objects (subobjects of the form $[X,\ X]$) that may be instantiated during the execution of P, and add the corresponding class X to the slice. (See lines [4] through [7].)

However, this is not enough. We also need to identify references (or pointers) to "partial" subobjects (subobjects of the form $[X,\ \alpha{\cdot}Y]$, where $Y \neq X$) that may be created. Such references are usually created through a sequence of (implicit or explicit) typecasts starting from a full object. Hence, we scan P for all typecast expressions and add an appropriate part of the class hierarchy to the slice. (See lines [8] to [10].) In particular, for a typecast from a class F to a class T we add *SubHierClass*(F, T) to the slice, where *SubHierClass*(F, T), defined below, denotes the part of the class hierarchy that lies between the two classes F and T.

[7] C++ programs usually have a number of *implicit* constructs: constructs that are generated by the compiler if the programmer omits them. For instance, a constructor for a class may contain *implicit* instantiations of all its data members. The algorithm needs to take care of these implicit constructs as well.

Definition 5.1 (*SubHierClass*)

$$\begin{aligned} \textit{SubHierClass}_\gamma(F,T) \triangleq \langle\ C,\ \emptyset,\ \hat{S}(\gamma) \cap (C \times C),\\ \hat{P}(\gamma) \cap (C \times C)\ \rangle \textit{ where}\\ C = \{\ X \mid \langle F,\ X\rangle \in \hat{I}(\gamma) \textit{ and } \langle X,\ T\rangle \in \hat{I}(\gamma)\ \} \end{aligned}$$

We now turn our attention to member lookups. The next step in the algorithm requires a type inference algorithm to identify for every member lookup operation $e.m$ or $e \rightarrow m$ in P the set of possible run-time types for e, which we denote by *PotentialRunTimeTypes*(e) (line [13]). (The algorithm requires the run-time type information only for lookups of virtual members. For the sake of simplicity, we ignore this possible refinement.) Once this information is available, the set of relevant member-lookup slicing criteria is easily determined. The set of possible run-time types of e and the static type of e are used to determine the set *SubObjSet* of possible subobjects that e may denote at this program point. (This is the information computed by Function *CorrespondingSubobjects* in lines [20]–[26].) Each $\sigma \in$ *SubObjSet* identifies a member-lookup slicing criterion $\langle \sigma,\ m\rangle$.

Once the set of member-lookup slicing criteria that are relevant have been identified, the algorithm simply computes member lookup slices with respect to each of these individual criteria and adds them all to the slice being computed.

The computation of member lookup slices uses a function *SubHier*, defined below, that given any two subobjects σ_1 and σ_2 identifies the part of the original class hierarchy that lies "between" these two subobjects. (Note that $\textit{SubHier}_\gamma(\sigma_1, \sigma_2) \sqsubseteq \textit{SubHierClass}_\gamma(\textit{ldc}(\sigma_1), \textit{ldc}(\sigma_2))$. We could use $\textit{SubHierClass}_\gamma(\textit{ldc}(\sigma_1), \textit{ldc}(\sigma_2))$ in place of $\textit{SubHier}_\gamma(\sigma_1, \sigma_2)$ when computing member lookup slices, but that would lead to larger slices in some cases.)

Definition 5.2 (*SubHier*)

$$\begin{aligned} \textit{SubHier}_\gamma(\sigma_1,\sigma_2) \triangleq \langle C,\ \emptyset,\ S,\ P\rangle \textit{ where}\\ R = \{\ \sigma \mid \sigma_1 \leq^* \sigma \leq^* \sigma_2\ \}\\ C = \{\ \textit{ldc}(\sigma) \mid \sigma \in \mathrm{R}\ \}\\ S = \{\ \langle \textit{ldc}(\sigma),\ \textit{ldc}(\sigma')\rangle \mid \sigma, \sigma' \in R,\ \sigma <_s \sigma'\ \}\\ P = \{\ \langle \textit{ldc}(\sigma),\ \textit{ldc}(\sigma')\rangle \mid \sigma, \sigma' \in R,\ \sigma <_p \sigma'\ \} \end{aligned}$$

```
[1]  function Slice(Program P) : hierarchy;
[2]      let γ be the class hierarchy of P;
[3]      instantiatedClasses = ∅;
[4]      for each (implicit or explicit) object creation construct e in P do
[5]          instantiatedClasses = instantiatedClasses ∪ { class instantiated by e } ;
[6]      end for
[7]      γ_slice = ⟨instantiatedClasses, ∅, ∅, ∅⟩
[8]      for each (implicit or explicit) typecast expression (T *) e or (T &) e or (T) e in P do
[9]          γ_slice = γ_slice ∪ SubHierClass_γ(StaticType(e), T);
[10]     end for
[11]     call ComputePotentialRunTimeTypes(P);
[12]     for each (implicit or explicit) expression e.m or e → m in P do
[13]         SubObjSet = CorrespondingSubobjects(γ, PotentialRunTimeTypes(e), StaticType(e));
[14]         for each σ in SubObjSet do
[15]             γ_slice = γ_slice ∪ MemberLookupSlice(γ, σ, m);
[16]         end for
[17]     end for
[18]     return γ_slice;
[19] end;

[20] function CorrespondingSubobjects(hierarchy γ; TypeSet T; Class X): SubObjectSet;
[21]     SubObjSet = ∅;
[22]     for each class C ∈ T do
[23]         SubObjSet = SubObjSet ∪ { σ | σ ∈ Σ(γ, C) and ldc(σ) = X };
[24]     end for
[25]     return SubObjSet;
[26] end;

[27] function MemberLookupSlice (hierarchy γ; Subobject σ; member m): hierarchy;
[28]     σ' = static-lookup(γ, σ, m);
[29]     staticLookupSlice = ⟨∅, M̄(γ, ldc(σ'), m), ∅, ∅⟩
[30]                         ∪ SubHier_γ(σ, σ')
[31]                         ∪ (⋃ { SubHier_γ(σ', σ'') | σ'' ∈ VisibleDefs(γ, σ, m) });
[32]     if ⟨ldc(σ'), m, "virtual"⟩ ∈ M̂(γ) then /* virtual member lookup */
[33]         σ' = dynamic-lookup(γ, σ, m);
[34]         dynamicLookupSlice = ⟨∅, M̄(γ, ldc(σ'), m), ∅, ∅⟩
[35]                              ∪ SubHier_γ([mdc(σ), mdc(σ)], σ')
[36]                              ∪ (⋃ { SubHier_γ(σ', σ'') | σ'' ∈ VisibleDefs(γ, [mdc(σ), mdc(σ)], m) });
[37]         lookupSlice = staticLookupSlice ∪ dynamicLookupSlice;
[38]     else
[39]         lookupSlice = staticLookupSlice;
[40]     end if
[41]     return lookupSlice;
[42] end;
```

Figure 5: Algorithm for slicing class hierarchies.

Let us now consider how the slice is computed for a lookup of a non-virtual member m in a subobject σ. (See lines [28]–[31].) Assume that the lookup returns a subobject σ' of least derived class C. The slice consists of several components. The first component (line [29]) ensures that the class C has the member m; for convenience we use $\overline{M}(\gamma,\ C,\ m)$ to denote the singleton set $\{\ \langle C,\ m,\ v\rangle\ \} \subseteq \hat{M}(\gamma)$, for some $v \in \mathcal{V}$. The second component (line [30]) ensures that σ' will be a subobject of σ in the new hierarchy and, hence, that the member m of σ' will be visible in σ. The third component (line [31]) adds enough of the hierarchy to ensure that the member m in σ' dominates any other member m that may be visible in σ. (This component is not necessary for a weak slice, but is essential for a slice.)

While the above components are all that are required in the case of a lookup for a non-virtual member, the lookup for a virtual member requires more parts of the class hierarchy to be added to the slice. The components added in lines [34]–[36] are analogous to the components added in lines [29]–[31] (except that they apply to the full object containing σ).

Remark: The functions *SubHierClass* and *SubHier*, as defined above, return all of the class hierarchy that lies between the specified classes or subobjects. For example, if D derives from a virtual base B in multiple ways, then *SubHierClass*(D, B) will include all the "inheritance paths" between D and B. In actuality, it would suffice if one of the "inheritance paths" between D and B was included in the slice. We choose to include all paths, rather than choose some path "non-deterministically". (Ideally, it might be preferable for the slice to just "record" that D is required to inherit from B in some fashion. It would, however, be difficult to "project" such slices onto the source program for display purposes.)

5.2 Correctness of the Algorithm

We now establish the correctness of our algorithm. We need to show that the hierarchy γ_{slice} returned by Function *Slice* satisfies the two conditions in Definition 4.5.

Consider condition (1). Let us denote the result of typecasting a subobject σ to a class X by *typecast*(γ, σ, X).

Lemma 5.3 *For every subobject σ instantiated during the execution of P, there exists a sequence of subobjects $[\mathtt{D},\ \mathtt{D}] = \sigma_0,\ \sigma_1,\ \cdots,\ \sigma_k = \sigma$, where D is a class instantiated in P, and for every $i > 0$, σ_i is obtained (during execution of P) from σ_{i-1} through a typecast (that is, σ_i = typecast$(\gamma, \sigma_{i-1}, X_i)$) or lookup (that is, σ_i = lookup$(\gamma,\ \sigma_{i-1},\ m_i)$).*

Proof. Straightforward. □

Lemma 5.4 *Let $\sigma_1 \leq^*_\gamma \sigma_2$. Then, for all $\gamma' \sqsupseteq$ SubHier$_\gamma(\sigma_1, \sigma_2)$ if $\sigma_1 \in \Sigma(\gamma')$ then $\sigma_2 \in \Sigma(\gamma')$ and $\sigma_1 \leq^*_{\gamma'} \sigma_2$.*

Proof. Straightforward. □

Lemma 5.5

1. *Let σ_2 = static-lookup$(\gamma,\ \sigma_1,\ m)$. For all $\gamma' \sqsupseteq$ SubHier$_\gamma(\sigma_1, \sigma_2)$, if $\sigma_1 \in \Sigma(\gamma')$ then $\sigma_2 \in \Sigma(\gamma')$.*

2. *Let σ_2 = dynamic-lookup$(\gamma,\ \sigma_1,\ m)$. For all $\gamma' \sqsupseteq$ SubHier$_\gamma([$mdc$(\sigma_1),\ mdc(\sigma_1)], \sigma_2)$, if $\sigma_1 \in \Sigma(\gamma')$ then $\sigma_2 \in \Sigma(\gamma')$.*

3. *Let σ_2 = typecast(γ, σ_1, T). For all $\gamma' \sqsupseteq$ SubHierClass$_\gamma($ldc$(\sigma_1),$ ldc$(\sigma_2))$, if $\sigma_1 \in \Sigma(\gamma')$ then $\sigma_2 \in \Sigma(\gamma')$.*

Proof. Follows from Lemma 5.4. (For the third part of the lemma, note that SubHierClass$_\gamma($ldc$(\sigma_1),$ ldc$(\sigma_2)) \sqsupseteq$ SubHier$_\gamma(\sigma_1, \sigma_2)$.) □

It follows from Lemma 5.3 and Lemma 5.5 that γ_{slice} satisfies condition (1) of Definition 4.5.

Consider condition (2) of Definition 4.5. Let us denote the value of the variable *lookupSlice* in a specific invocation of the function *MemberLookupSlice*(γ, σ, m) by *lookupSlice*(γ,σ,m). *staticLookupSlice*(γ,σ,m) and *dynamicLookupSlice*(γ,σ,m) are defined similarly.

Lemma 5.6 *staticLookupSlice(γ, σ, m) is a static-lookup slice of γ with respect to $\langle\sigma,\ m\rangle$.*

Proof. Consider any γ' such that

$\gamma' \sqsubseteq \gamma, \gamma' \sqsupseteq$ staticLookupSlice(γ, σ, m), and $\sigma \in \Sigma(\gamma')$.

We need to show that

$$\textit{static-lookup}(\gamma', \sigma, m) = \textit{static-lookup}(\gamma, \sigma, m).$$

Let σ' denote *static-lookup*(γ, σ, m).

First, we show that $\sigma' \in$ *VisibleDefs*(γ', σ, m). Observe that $\sigma' \in \Sigma(\gamma')$ and $\sigma \leq^*_{\gamma'} \sigma'$. (This follows from Lemma 5.4 since $\sigma \in \Sigma(\gamma')$ and $\gamma' \sqsupseteq \textit{SubHier}_\gamma(\sigma, \sigma')$.) This implies that there is a "path" $\sigma <_{\gamma'} \sigma_1 \cdots \sigma_k <_{\gamma'} \sigma'$ in the subobject graph of γ'. Clearly, $m \notin \hat{M}(\gamma)(\textit{ldc}(\sigma_i))$ for any i, since otherwise we would not have σ' = *static-lookup*(γ, σ, m). Since $\gamma' \sqsubseteq \gamma$, $m \notin \hat{M}(\gamma')(\textit{ldc}(\sigma_i))$ either. It follows that σ' must be in *VisibleDefs*(γ', σ, m)

We now need to show that σ' is the least element of *VisibleDefs*(γ', σ, m). Let σ'' be some element in *VisibleDefs*(γ', σ, m). This implies that there is some "path" $\sigma <_{\gamma'} \sigma_1 \cdots \sigma_{k-1} <_{\gamma'} \sigma_k = \sigma''$ in the subobject graph of γ'. Let i be the minimum value such that $m \in \hat{M}(\gamma)(\textit{ldc}(\sigma_i))$. Then, $\sigma_i \in$ *VisibleDefs*(γ, σ, m). By construction, *staticLookupSlice*$(\gamma, \sigma, m) \sqsupseteq \textit{SubHier}_\gamma(\sigma', \sigma_i)$. Hence, $\gamma' \sqsupseteq \textit{SubHier}_\gamma(\sigma', \sigma_i)$. Lemma 5.4 implies that $\sigma' \leq^*_{\gamma'} \sigma_i$. Hence, $\sigma' \leq^*_{\gamma'} \sigma''$, from the transitivity of $\leq^*_{\gamma'}$. □

Lemma 5.7 *dynamicLookupSlice*(γ, σ, m) *is a dynamic-lookup slice of* γ *with respect to* $\langle \sigma, m \rangle$.

Proof. This follows just like Lemma 5.6. □

Lemma 5.8 *lookupSlice*(γ, σ, m) *is a member lookup slice of* γ *with respect to* $\langle \sigma, m \rangle$.

Proof. Follows from Lemma 5.6 and Lemma 5.7. □

It follows from Lemma 5.8 and Lemma 4.10 that γ_{slice} satisfies condition (2) of Definition 4.5 also. Hence, we have the following theorem.

Theorem 5.9 *Function Slice computes a class hierarchy slice of* P.

6 Related Work

6.1 Elimination of Class Components

The work that is most closely related to ours is that by Agesen and Ungar [4, 3], who describe an approach for *application extraction* for the dynamically typed object oriented language Self. Agesen and Ungar's objective is similar to ours: the elimination of unused parts of objects while preserving program behavior. It is interesting to observe that, whereas for statically typed object-oriented languages eliminating such redundancies is a useful optimization, for Self it is almost essential: Due to the absence of declarations, it is unclear *a priori* what code is used by an application and, hence, even a small application requires the incorporation of the entire run-time environment unless countermeasures are taken. Comparing the application extraction algorithm of [4] with our class hierarchy slicing algorithm is difficult because of the different languages under consideration, and the differences in presentation of the algorithms, but a few similarities are evident:

- Both algorithms rely on a type inference algorithm to determine the potential targets of method calls (message sends in Self). For C++, due to the static nature of class hierarchies, there is the possibility of using efficient algorithms that only use class hierarchy information (see Section 6.2). For Self, this option is not available.
- Both algorithms essentially determine a separate slice for each lookup/send, and the slice with respect to a program consists of the union of these single-point slices.

In his PhD thesis [3], Agesen discusses related work on application extraction for dynamically typed object-oriented languages.

To our knowledge, the problem of eliminating dead *declarative* code in C++ programs has not been studied. For statically typed object-oriented languages such as C++, the focus has thus far been on eliminating dead *executable code*, and we are unaware of any approaches that go much beyond call graph analysis [28] and virtual call elimination [7]. As Srivastava's study [28] indicates, object-oriented programs are likely to contain more dead code than programs written in procedural languages. Agesen [3] remarks that "as the current trend towards the use of frameworks in the development of applications continues, the amount of dead code will likely increase further." We believe that this applies to executable as well as declarative code.

6.2 Type Inference

The class hierarchy slicing algorithm of Section 5 requires a set of potential run-time types for each expression that is involved in a member access. A number of different approaches for type inference have been discussed in the literature.

In the context of C++, much work has been done recently on type inference for the sake of virtual method call elimination. The objective of virtual call elimination is to identify calls to virtual functions from receiver expressions that can only have a single run-time type; such calls can be replaced by direct function calls. Besides the fact that a direct call can be implemented more efficiently, they can also be inlined, which enables various intraprocedural optimizations that cannot be performed across function boundaries. Although C++ type inference algorithms are often designed primarily to determine whether the run-time type of a receiver expression is unique or not, extension of these algorithms to compute a set of potential run-time types instead seems straightforward in most cases. C++ type inference algorithms fall into two broad categories: algorithms that only use signature information or class hierarchy information [10, 16, 28, 8, 7, 13] and more sophisticated algorithms that are based on alias analysis [23, 11]. Although the latter category theoretically offers the most precise results, it is unclear how much better these results are than those of algorithms in the former category in practice [7].

In addition to the C++ type inference algorithms described above, several *constraint-based* type inference algorithms for object-oriented languages have been presented in the literature. Constraint-based type inference methods compute type information by determining the solution of a constraint network where the nodes correspond to type variables, and where the edges represent constraints between type variables [22, 2, 24].

6.3 Statement Slicing for C++

Although class slicing in abstract form simply specializes a class hierarchy with respect to a target program, several applications also require a statement slicing algorithm. Krishnaswamy [19] and Larsen and Harrold [20] have addressed statement slicing in C++.

Krishnaswamy's algorithm [19], which is based on an algorithm for the C language by Livadas and Croll [21], has insufficient detail for us to confidently evaluate its accuracy or correctness, although it appears to outline a slicing algorithm based on interprocedural dataflow analysis in a procedure call graph derived from a set of methods.

Later work on slicing in C++ by Larsen and Harrold [20], based on work on interprocedural slicing by Horwitz, et al. [18] and Reps et al. [25], contains considerably more detail. The most significant contributions of Larsen and Harrold's work, in our view, are the idea of treating all instances of method data as global variables (i.e., "`static`" members in C++), and the introduction of a special node in their dependence graph representation to account for the dynamic binding implicit in virtual function calls.

However, this work does not explicitly address a number of common constructs in C++; e.g., object reference via the "`this`" pointer, direct access to member data of objects (only access via member functions is discussed), and calls to methods on objects that are method arguments or member data. The latter two omissions prevent the Larsen/Harrold algorithm from being immediately applicable to the example of Figure 3, although it is not difficult to apply some simple "patches" to their algorithm sufficient to yield the slice in the example.

Despite the limitations of existing C++ slicing algorithms, considerable progress has been made in developing slicing techniques that address problematic constructs in other languages, e.g., unstructured control flow [5, 12, 9], composite datatypes [6], and pointers [6, 15, 17]. We believe that many of these techniques are applicable to C++ as well, and algorithms for statement slicing in C++ are likely to improve over time.

7 Conclusions and Future Work

We have defined a semantically well-founded notion of a slice of a class hierarchy. This notion of a slice is defined using an adaptation of Rossie and Friedman's algebraic semantics for subobject selection [26]. In addition, we have presented an algorithm for computing class hierarchy slices. This algorithm relies on the availability of run-time type information for receiver

expressions of virtual calls, but any suitable type inference method can be used to compute this information, allowing for a variety of cost/accuracy tradeoffs.

Our class hierarchy slicing algorithm has two principal applications when used in conjunction with a statement slicing algorithm. First, it enhances the utility of statement slicing in traditional slicing application such as debugging and program understanding. Second, the combination of the two algorithms may be used to optimize an object-oriented program by reducing its space requirements.

For future work, we would like to extend the framework presented in this paper to other declarative aspects of C++ not treated here, such as static class members, pointers to members, and `typedef` constructs. In addition, slicing in the presence of preprocessing mechanisms such as templates or macros remains a difficult open problem.

Finally, we wish to investigate how to compute slices that are not necessarily projections of the source code, e.g., by allowing transitive inheritance relations to be replaced by direct inheritance relations.

Acknowledgements

We thank Jon Rossie for several helpful discussions about the subobject model of [26].

References

[1] ACCREDITED STANDARDS COMMITTEE X3, I. P. S. Working paper for draft proposed international standard for information systems—programming language C++. Draft of 26 september 1995.

[2] AGESEN, O. Constraint-based type inference and parametric polymorphism. *Proceedings of the First International Static Analysis Symposium (SAS'94)* (September 1994), 78–100. Springer-Verlag LNCS vol. 864.

[3] 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.

[4] 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).

[5] AGRAWAL, H. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (Orlando, Florida, 1994), pp. 302–312. SIGPLAN Notices 29(6).

[6] AGRAWAL, H., DEMILLO, R., AND SPAFFORD, E. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)* (1991), pp. 60–73. Also Purdue University technical report SERC-TR-93-P.

[7] 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).

[8] BACON, D. F., WEGMAN, M., AND ZADECK, F. K. Rapid type inference for C++. Tech. Rep. RC 1234, IBM Thomas J. Watson Research Center, 1995.

[9] BALL, T., AND HORWITZ, S. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 206–222.

[10] 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.

[11] 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.

[12] CHOI, J.-D., AND FERRANTE, J. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems 16*, 4 (July 1994), 1097–1113.

[13] DEAN, J., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. Tech. Rep. 94-12-01, Department of Computer Science, University of Washington at Seattle, December 1994.

[14] ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[15] ERNST, M. Practical fine-grained static slicing of optimized code. Tech. Rep. MSR-TR-94-14, Microsoft Research, Redmond, WA, 1994.

[16] FERNANDEZ, M. F. Simple and effective link-time optimization of modula-3 programs. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation* (June 1995), 103–115.

[17] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[18] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[19] KRISHNASWAMY, A. Program slicing: An application of object-oriented program dependency graphs. Technical report TR94-108, Dept. of Computer Science, Clemson University, 1994.

[20] LARSEN, L., AND HARROLD, M. J. Slicing object-oriented software. In *Proceedings of the 1996 International Conference on Software Engineering (ICSE-18)* (Berlin, March 1996).

[21] LIVADAS, P. E., AND CROLL, S. Program slicing. Report serc-tr-61-f, Computer Sciences Department, University of Florida, 1992.

[22] PALSBERG, J., AND SCHWARTZBACH, M. I. Object-oriented type inference. *Proceedings of the ACM 1991 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)* (October 1991), 146–161. ACM SIGPLAN Notices 26(11).

[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] PLEVYAK, J., AND CHIEN, A. A. Precise concrete type inference for object-oriented languages. *Proceedings of the ACM 1994 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)* (October 1994), 324–340. ACM SIGPLAN Notices 29(10).

[25] REPS, T., HORWITZ, S., SAGIV, M., AND ROSAY, G. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering* (New Orleans, LA, December 1994), pp. 11–20.

[26] 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).

[27] SAKKINEN, M. A critique of the inheritance principles of C++. *Computing Systems 5*, 1 (1992), 69–110.

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

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

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