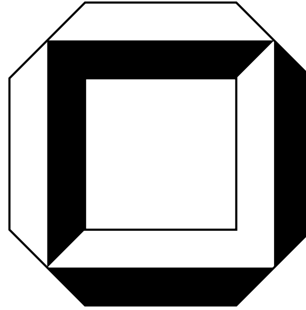

University of Karlsruhe
Department of Computer Science
System Architecture Group

Applying Source Code Transformation to Collapse Class Hierarchies in C++

Studienarbeit / Study Thesis

Jan Oberländer
<uslk@stud.uni-karlsruhe.de>
Supervisor: Uwe Dannowski

December 23, 2003

Abstract

For many software projects, it is desirable to use the features of an object-oriented programming language such as C++ to write maintainable code, to implement clean interfaces, and to facilitate code reuse. C++ virtual functions are a central building block for object-oriented software architectures. In certain use cases like operating system kernels and especially in microkernels extremely efficient code is required. However, such mechanisms impose too much run-time overhead, making the use of object-oriented code too impractical. This is one of the classic reasons against the use of object-oriented languages for building performance-critical systems.

This study thesis describes an optimization technique that allows the use of C++ class hierarchies with virtual functions while producing efficient code. This is achieved by transforming the source code to collapse any specified class hierarchy into one flat class. Semantic equivalence is maintained as far as necessary, in a limited environment.

The technique is implemented in a source-to-source transformation tool which this paper discusses in detail. Some evaluation shows that the tool produces optimized code which is more efficient, without producing significant compile-time overhead.

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that no sources other than the cited sources have been used.

Karlsruhe, den 23. Dezember 2003

Jan Oberländer

Contents

1 Motivation **1**
- 1.1 The Overhead Incurred by Polymorphic Functions . . . 1
- 1.2 Circumventing Polymorphic Calls . . . 2
- 1.3 A Case in Point: L4 . . . 3
- 1.4 The Idea . . . 5

2 Related Work **6**
- 2.1 Source Code Transformation Tools . . . 6
- 2.2 Representing C++: Data Exchange Formats . . . 7

3 Some Background on C++ **8**
- 3.1 The C++ Object Model . . . 8
- 3.2 Inheritance . . . 8
- 3.3 Virtual Base Classes . . . 8
- 3.4 Virtual Functions . . . 9
- 3.5 Member Lookup . . . 10
- 3.6 Constructors . . . 10
- 3.7 Inline Member Functions . . . 11

4 Design **13**
- 4.1 The C++ AST . . . 13
- 4.2 Visitors . . . 13
- 4.3 The Transformation . . . 15
  - 4.3.1 Finding All Class Declarations . . . 15
  - 4.3.2 Merging Class Members . . . 15
  - 4.3.3 Removing `virtual` Specifiers . . . 16
  - 4.3.4 Removing the Complete Base Class Declarations . . . 16
  - 4.3.5 Constructor Conversion . . . 16
  - 4.3.6 Removing Duplicate Members . . . 17
  - 4.3.7 Removing All Base Class Specifiers . . . 17
  - 4.3.8 Sorting . . . 17

5 The Implementation **18**
- 5.1 Visiting the AST . . . 18
- 5.2 Finding All Class Declarations . . . 18
- 5.3 Merging Class Members . . . 19
- 5.4 Devirtualizing All Methods . . . 20
- 5.5 Removing the Base Classes . . . 20
- 5.6 Constructor Conversion . . . 20
- 5.7 Removing Duplicate Members . . . 20
- 5.8 Removing the Base Class References . . . 21
- 5.9 Sorting . . . 21

6 Scope and Limitations of the Work **22**
6.1 Visibility Semantics 22
6.2 Base Class Semantics 22
6.3 Namespaces 22
6.4 Ordering and Alignment 23
6.5 Constructors and Member Initialization 23
6.6 Destructors 23
6.7 Semantic Checking 23

7 Performance **24**
7.1 Virtual vs. Non-virtual Function Calls 24
7.2 Build Process Overhead 24

A Functionality **25**
A.1 Using *collapse* 25
A.2 autoconf Support 26

B A fairly complex example for constructor conversion **27**

C Code excerpts from collapse **30**
C.1 A Visiting Algorithm 30
C.2 Inserting and Removing List Elements 30

Listings

1.1 Comparison of virtual and non-virtual functions . . . 2
1.2 Calls to virtual and non-virtual functions . . . 3
1.3 Example for class hierarchies requiring `virtual` specifiers . . . 4
3.1 Polymorphy in C++: non-polymorphic functions . . . 9
3.2 Polymorphy in C++: polymorphic functions . . . 10
3.3 Constructors with *mem-initializers* for base classes . . . 11
3.4 Inline and non-inline member functions . . . 12
5.1 Merging member functions . . . 19
B.1 A complex class hierarchy with constructors . . . 28
B.2 Collapsed version of the class `A` from Listing B.1 . . . 29
C.1 An algorithm for safely visiting lists while they are modified . . . 31
C.2 Inserting and removing list elements . . . 31

List of Figures

1.1 A small example for mix-in classes 3

3.1 Class hierarchy for `A` for non-virtual base classes 8
3.2 Class hierarchy for `A` for virtual base classes 9

4.1 A piece of C++ code and the corresponding AST 14

B.1 Inheritance graph for Listing B.1 27

List of Tables

7.1 Execution times for compiling a machine-generated example class hierarchy with and without collapsing. 24

Chapter 1

Motivation

A common and very important goal in software engineering is to enable programmers to write code that is well-structured and highly maintainable. A clean code structure with very clearly outlined components helps reduce the likelihood of mistakes in the code and improve verifiability. It is much easier to re-design parts of the code if the code is designed in a modular way. Object-oriented programming languages provide a great means to write such code with a modular design, a good structure and therefore, good maintainability.

Component-based software development focuses on building large software systems by integrating previously existing software components. By enhancing the flexibility and maintainability of systems, this approach can potentially be used to reduce software development costs, to assemble systems rapidly, and to reduce the spiraling maintenance burden associated with the support and upgrade of large systems. At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity so that common parts should be written once, rather than many times, and that systems should be assembled through reuse rather than be rewritten over and over again.

This approach can also be used to generate whole families of a software product from a configurable and portable code base. The component structure reflects the parts that need to be exchanged for generating another family member. The more fine-grained the component structure, the more degrees of freedom of configuration and thus higher portability.

Unfortunately, almost every additional component boundary adds run-time overhead. In most performance-critical contexts, the overhead introduced by some of the advanced concepts offered by object-oriented languages may be too high to justify their use. This might rule out the use of such concepts, or even object-oriented languages as the programming language of choice in general.

The goal of this work is to provide a tool that gives programmers the best of two worlds: The ability to use the well-accepted software engineering paradigm of object orientation with class hierarchies and polymorphic function calls, plus a way to still develop efficient software.

Polymorphic functions in class hierarchies let the software engineer write functions that exist in more general base classes as well as in specialized derived classes, bearing the same signature, but specialized implementation codes. The right instance of a function to be called is determined at run-time depending on the actual object. Such functions allow for the writing of well-maintainable code. But, the polymorphy mechanism incurs an overhead on each function call because it requires an indirect lookup of the address of the called function. This may in some cases be too inefficient to allow true object-oriented programming in strongly performance-dependent situations. One such situation being examined further in this paper is the use of object-oriented programming in microkernel code.

1.1 The Overhead Incurred by Polymorphic Functions

The difference in performance between polymorphic and non-polymorphic functions is due to the different calling mechanism used for polymorphic calls. The actual function to be called is determined at run time, depending on what class the object whose function is called is an instance of. This requires an indirect lookup of the function address.

The lookup differences can be seen in the code generated by Listing 1.1. Class `Foo` has a virtual function while class `Foo2` has a similar function without the `virtual` keyword.

In the first case, the actual function to be called must be determined at run time. If `o->foo()` is called for an instance `o`, then the correct `foo()` function to call is the one belonging to the class that `o` is an instance of (which may be either `Foo` or any derived class of `Foo`). In the second case, it is already clear at compile time which `foo()` function is called: it simply depends on the type of the object variable. If `o` is of type `Foo2`, then `Foo2::foo()` is called even if the object represented by `o` is actually an instance of a derived class of `Foo2`. The difference of a call to `Foo::foo()` and `Foo2::foo()` can be seen in the IA32 assembly code in Listing 1.2.

As the code example shows, a non-virtual function call simply places a call to the address resolved at linking time (`_ZN4Foo23fooEv`). A virtual call, on the other hand, implies that first the pointer to the function must be read from the object in question. This clearly requires more code and several additional memory accesses. The resulting execution time overhead of virtual function calls in C++ has been measured to be as high as 40% [1, 2].

Therefore, using virtual functions can cause a slowdown that might be too inefficient to consider such a programming style in a microkernel setting.

1.2 Circumventing Polymorphic Calls

This paper proposes a way to circumvent the polymorphic call mechanism in certain cases. There are two major uses for polymorphic functions: specialization and composition from components. The first is best characterized by a simple example. Given a class hierarchy describing different geometric shapes, a polymorphic function calculating the area of a shape would be re-implemented for each class. A function operating on a collection of shapes will then call the area function for each shape, and the polymorphy mechanism will ensure that the correct area function is called depending on the exact type of the processed shape. For this pattern, polymorphy can not be avoided.

The second use for polymorphic functions is when constructing mix-in classes where only functions within the class itself are dependent on other member functions that need to be polymorphic. Given an abstract base class A declaring some functions $A.a, A.b$, one might derive a set of mix-in classes B_a, B_b, the former implementing $A.a$, and the latter implementing $A.b$. One could additionally write different implementations $B_{a,1}, \ldots, B_{a,m}, B_{b,1}, \ldots, B_{b,n}$ of these mix-ins. Then a class C could be derived from these mix-ins combining two different implementations of the functions a and b. If we now further assume that a calls b or vice versa, then the calls to a and/or b need to be polymorphic. In this case, the polymorphy mechanism is only needed internally, and no code using the constructed class depends on the polymorphy since all created objects would be of type C. This is the case we want to exploit, and in this case polymorphy can be circumvented by collapsing the class.

A simple example where such mix-ins might be appropriate is for a collection of three-dimensional bodies that can be constructed from two-dimensional shapes (such as pyramids or tori). To calculate the volume of such a body, a mix-in class structure as displayed in Figure 1.1 could be used. One mix-in component provides the volume calculation methods, using the base area calculation method provided by the other mix-in component. Mix-ins can be combined to create different bodies such as cones (Pyramid+Circle), cuboids

```
class Foo {
public:
    virtual void foo(void) {
        // Some code
    }
};

class Foo2 {
public:
    void foo(void) {
        // Some code
    }
};
```

Listing 1.1: Comparison of virtual and non-virtual functions

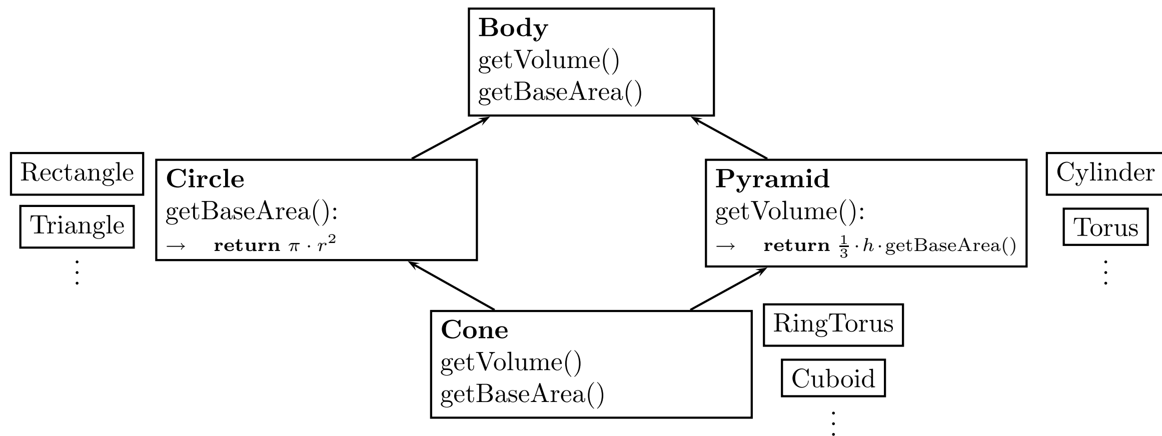

Figure 1.1: A small example for mix-in classes

(Cylinder+Rectangle), or ring tori (Torus+Circle).

As another example, consider the code given in Listing 1.3. To get the desired output as pointed out in the comments in `main()` (which corresponds to a mix-in pattern as explained above), `virtual` specifiers are needed.

The next section will show practical examples in an actual application case where the collapsing technique could pay off.

1.3 A Case in Point: L4

This project was created out of an idea of the L4Ka research group within the System Architecture group at the University of Karlsruhe [3]. The group is concerned with the development of the L4 microkernel originally created by Jochen Liedtke.

L4Ka::Pistachio is the latest and most portable L4 incarnation running on a number of different architectures, including Intel IA32 and IA64, PowerPC 32-Bit, Alpha, and MIPS 64-Bit architectures. The microkernel is written in C++, and as such wants to make use of the advantages an object-oriented language has to offer. Even now, its performance would be challenging for an assembly implementation (if one existed). The kernel may use any advantages of an object-oriented language, but it currently avoids the use of polymorphy because of said performance problems. There are many cases where the aforementioned mix-in pattern would be very useful,

```
// virtual function call to Foo::foo on IA32
    subl    $12, %esp
    movl    -12(%ebp), %eax // pointer to object
    movl    (%eax), %eax    // pointer to vtable
    pushl   -12(%ebp)
    movl    (%eax), %eax    // actual function address
    call    *%eax
    addl    $16, %esp

// non-virtual function call to Foo2::foo on IA32
    subl    $12, %esp
    pushl   -16(%ebp)
    call    _ZN4Foo23fooEv  // function address
    addl    $16, %esp
```

Listing 1.2: Calls to virtual and non-virtual functions

```
class B {
public:
    virtual void foo() {
        printf("B::foo()\n");
    }
    virtual void bar() {
        printf("B::bar()\n");
    }
};

class M1 : virtual public B {
public:
    virtual void foo() {
        printf("M1::foo()\n");
    }
    void goo() { bar(); }
};

class M2 : virtual public B {
public:
    virtual void bar() {
        printf("M2::bar()\n");
    }
    void baz() { foo(); }
};

class R : M1, M2 {
public:
    void gixgax() {  // Expected output:
      foo();         // M1::foo()
      bar();         // M2::bar()
      goo();         // M2::bar()
      baz();         // M1::foo()
    }
};
```

Listing 1.3: Example for class hierarchies requiring `virtual` specifiers

as some of the data structures provided in the kernel require different function implementations depending on the underlying architecture or the version of the L4 API (such as X.0, X.2, and v4).

A prominent example is the thread control block data type `tcb_t`. It provides functions for thread notification and for sending page fault IPCs which are obviously architecture-dependent. Other functions are different depending on the API version, and the TCB's member variables clearly depend on the underlying architecture as well. Therefore, a mix-in structure would be desirable, with a common abstract base class `tcb_t` providing all member function signatures, and mix-ins providing API-dependent functions or architecture-dependent functions, with different mix-in versions for the different APIs and architectures. For example, the child class `ia64_x2_tcb_t` might be derived from two parent classes `ia64_tcb_t` (containing special IA64-related members) and `x2_tcb_t` (containing functions as specified by the L4 version X.2 API), each of which is derived from the common `tcb_t` base class.

Another data structure that could be much improved is the address space representation `space_t`. It includes functions for handling TCBs which might have to be implemented differently for different API versions, and has some architecture-specific member variables and member functions, such as the page mapping functions. The mix-in pattern would work here as well, allowing for the same kind of optimization.

A proper object-oriented construction of these example classes using mix-ins implies, as explained in the previous section, the use of virtual classes and methods. L4Ka::Pistachio has classes that would be implemented in a much cleaner way if such class hierarchies could be built. However, the problem is that a microkernel requires high efficiency in order to avoid being a bottleneck for application performance, and therefore virtual methods must be avoided at all costs.

1.4 The Idea

C++ is widely considered to be inappropriate for microkernel programming; the L4Ka project wants to show that it is in fact possible to implement a microkernel in C++ without taking the described performance hit. Pistachio does perform quite well despite being written in C++, but currently avoids polymorphy. As a result, some of the classes in the code look rather unpleasant and are therefore not very maintainable.

It is possible to use more of what C++ has to offer without hurting the performance. In a limited setting, this goal can be achieved. The job is to transform the source code so that class hierarchies can be collapsed into flat classes, while trying to keep the class hierarchy semantics intact as much as necessary. This enables programmers to write normal C++ classes and have the performance hit compensated for by transforming the code so that crucial classes are collapsed.

Chapter 3 is going to provide a quick background on the features of C++. In Chapter 4, details about the design of *collapse* are discussed, while Chapter 5 talks about specifics of the implementation. Finally, some outlook is given in Chapter 6 describing limitations of the project and looking into possible future enhancements. Chapter 7 takes a look at some actual test cases for *collapse*.

Chapter 2

Related Work

The performance overhead introduced by the powerful mechanisms C++ and other object-oriented languages have to offer is a well-known issue, and the run-time penalty incurred by dynamic dispatch for virtual functions has been discussed in a number of research papers [1, 2].

Much work has been done trying to reduce the number of virtual function calls (VFCs) in object-oriented programs; some of it was inspired while trying to solve other problems. Class hierarchy analysis [4] inspects call sites to potentially reduce dynamic dispatch to static dispatch by inferring from context (pointer p will always point to objects of class C). Profile-based type feedback [5] allows well-predicted run-time type checking followed by static dispatch or inlining. However, the virtual function call is only turned into a likely to be taken yet conditional call. Sometimes virtual inheritance is unnecessary for a given application and can be turned into normal inheritance after whole-program inspection [2, 6]. Class hierarchy specialization [6, 7, 8] creates a new class hierarchy with reduced object size and potentially devirtualized functions. Common to most of these approaches is that they analyze a given source base only. In particular, none of them use developer expertise to guide their process. They may fail to achieve the maximum possible level of optimization because of not knowing the developer's intentions, but just inspecting how he/she managed to express themselves. Still, this is not a bad thing because it enables optimizations even for code for which no expert knowledge exists anymore.

Two papers concern themselves with optimization techniques for virtual function calls. [5] presents two methods to eliminate virtual function calls. The Unique Name technique finds VFC sites for which there only exists one single possible target function to call, in which case the call is replaced by a direct call. The Single Type Prediction technique uses profiling techniques to find for each VFC site a target function that is most often called, and replaces the lookup in the virtual function table with a multi-way branch placing the most likely called target function first. The implementation is an existing C++ compiler with a modified front-end. In [9], a similar approach is used. Only here the optimizer uses source code transformation to achieve its goals. Both projects also emphasize that with their approach, it is also possible to inline such optimized virtual function calls, something that is generally not possible with virtual calls.

Aspect-oriented programming (AOP) may be used to generate results similar to those of class collapsing. However, code needs to be specifically written for AOP. It is not clear yet whether AOP is equally suitable for all the cases we can address with our technique. Currently available tools [10, 11] have not matured yet to be used for low-level programming as in, for example, microkernels.

While these approaches can in some cases successfully optimize code in situations where *collapse* can not do anything (for instance, in the specialization case explained in Section 1.2), the approach described here has two advantages. Firstly, acting as a pure source code transformation tool decouples the implementation from the used compiler and is thus more general. Secondly, the limited setting assumed for *collapse* reduces the complexity of the required source code analysis.

2.1 Source Code Transformation Tools

The implementation described in this paper builds on the libcast [12] library to parse C++ code, to obtain a data structure representing the source code, and to write the possibly transformed data structure back to a source code file.

Some other projects provide similar functionality upon which a *collapse*-like tool could be built. The most interesting project seems to be OpenC++ [13], a source-to-source translator facilitating the development of C++

language extensions, domain specific compiler optimizations and run-time metaobject protocols. OpenC++ is also a code base for projects requiring a C++ parser and static analyzer.

Another project, the Open Source C++ Fact Extractor cppx [14], produces a graph from the source code in different graph description languages such as the Graph Exchange Language GXL [15, 16]. Preprocessing, parsing, and semantic analysis are left to the GNU C++ compiler [17]. While the GXL format is a very general format intended as a standard exchange format for graphs and is thus rather complex, it should be possible to do the same transformation work on a GXL tree as with the other tools, albeit in a probably less comfortable way.

2.2 Representing C++: Data Exchange Formats

The Workgroup on Standard Exchange Format (WoSEF) [18] mentions several efforts to represent C++ at the AST level, among them the DATRIX abstract semantic graph [19] as the probably most popular model which can be used to represent C, C++, and Java programs, and which seems to be the quasi-standard exchange format used by many tools. The above mentioned GXL format is also used by the DATRIX group.

Chapter 3

Some Background on C++

3.1 The C++ Object Model

C++ has enhanced C by adding a complex object model. In addition to `struct`s, C++ knows `class`es, multiple inheritance, polymorphy, operator overloading, abstract functions, a complex constructor/deconstructor model, and templates.

In C++, `class`es and `struct`s can both contain member variables and functions and inherit from base classes. `union`s still exhibit the same behavior as in standard C, but can now also contain member functions.

3.2 Inheritance

The inheritance semantics of all three aggregate types are very different. Unions can not have any base classes, nor can they be used as a base class [20, Section 9.5, §1]. Classes and structs are essentially identical, but their members have different default visibilities. Members of a `class` are by default private, as are any base class sub-objects. `struct` members are by default public, to mimic standard C behavior.

C++ allows multiple inheritance (as opposed to, for example, Java), meaning multiple base classes and/or structs can be specified. This leads to problems regarding unambiguous member names; such naming conflicts are pointed out by the compiler. Generally, it is possible to have base classes with members of the same name, so long as all references to those members can be resolved unambiguously, for instance by explicitly specifying the base class name of the requested member. For example, if a class `A` is derived from two classes `B` and `C`, each of which is derived from a class `D`, and `D` contains a member `foo`, then, if one wants to access `foo` from `A`, the naming is ambiguous and one needs to qualify `foo` as either `B::D::foo` or `C::D::foo`. This changes when base classes are declared `virtual`.

3.3 Virtual Base Classes

The example mentioned above corresponds to the class hierarchy shown in Figure 3.1. By changing the definitions of the classes `B` and `C` so that their base class `D` is declared `virtual`, `D` will be unique within the hierarchy of `A` and other classes derived from `B` and `C`. By writing

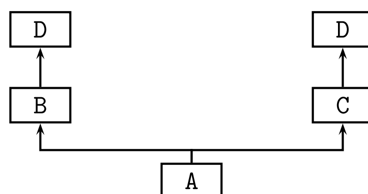

Figure 3.1: Class hierarchy for `A` for non-virtual base classes

```
class B : virtual public D {
    /* ... */
};
```

(and similarly for `C`), the hierarchy then looks as shown in Figure 3.2, and the member `foo` from the above example will be unambiguous.

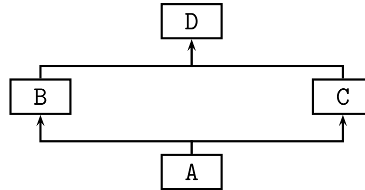

Figure 3.2: Class hierarchy for `A` for virtual base classes

3.4 Virtual Functions

To enforce polymorphic behavior of function calls, member functions need to be declared `virtual`. If not, the called function will be determined by the type of the object variable at compile time. Only when a function is declared `virtual` it is determined at run time what class a particular object is an instance of, and the appropriate version of the function is called. As an example, consider the code segment given by Listing 3.1.

```
class Foo {
public:
    void foo() {
        printf("I am Foo::foo\n");
    }
};

class Bar : public Foo {
public:
    void foo() {
        printf("I am Bar::foo\n");
    }
};

int main(void) {
    Foo *f = new Foo;
    Foo *g = new Bar;

    f->foo();
    g->foo();
    return 0;
}
```

Listing 3.1: Polymorphy in C++: non-polymorphic functions

Both calls to function `foo()` within `main` will yield an output of "`I am Foo::foo`", since, by default, polymorphy is not supported. To achieve polymorphic calls to `foo`, it needs to be declared `virtual`, as suggested by Listing 3.2. Then, the call to `g->foo()` in `main()` will instead yield an output of "`I am Bar::foo`".

```
class Foo {
public:
    virtual void foo() {
        printf("I am Foo::foo\n");
    }
};

class Bar : public Foo {
public:
    virtual void foo() {
        printf("I am Bar::foo\n");
    }
};
```

Listing 3.2: Polymorphy in C++: polymorphic functions

3.5 Member Lookup

The ISO C++ specification [20, Section 10.2] defines the lookup of member names as follows:

> *The following steps define the result of name lookup in a class scope,* `C`*. First, every declaration for the name in the class and in each of its base class sub-objects is considered. A member name* `f` *in one sub-object* `B` hides *a member name* `f` *in a sub-object* `A` *if* `A` *is a base class sub-object of* `B`*. Any declarations that are so hidden are eliminated from consideration. Each of these declarations that was introduced by a* using-declaration *is considered to be from each sub-object of C that is of the type containing the declaration designated by the* using-declaration*. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity and the program is ill-formed. Otherwise that set is the result of the lookup.*

If a member name is ambiguous, one must fully qualify it by preceding it with the name of the class the desired member belongs to.

3.6 Constructors

The C++ constructor model is a fairly complex one. In Java, for example, constructors basically look like methods with no type bearing the name of their class. Constructors can be overloaded with different signatures. This is basically the same in C++, but much more is possible. Most importantly, the initialization rules are much more complex. First, it is possible to specify *mem-initializers* as shown in Listing 3.3. It is possible to initialize members and explicitly specify how constructors for base classes are to be called. The process of initializing a class and its members thus becomes more complex, and works as described in the following paragraphs, as quoted from [20, Section 12.6.2, §§5f.]:

> 5. *Initialization shall proceed in the following order:*
>   - *First, and only for the constructor of the most derived class as described below, virtual base classes shall be initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where left-to-right is the order of appearance of the base class names in the derived class* base-specifier-list.
>   - *Then, direct base classes shall be initialized in declaration order as they appear in the* base-specifier-list *(regardless of the order of the* mem-initializers*).*
>   - *Then, nonstatic data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the* mem-initializers*).*
>   - *Finally, the body of the constructor is executed.*

```
class Foo : public Bar {
    int a;
public:
    Foo(int x, int y) : a(x) {
        /* Member a is initialized with the value of x */
    }

    Foo(int x, int y) : Bar(x, y-1), a(y) {
        /* The base class Bar is initialized with a call to its constructor,
         * Bar(x, y-1). The member a is initialized with the value of y.
         */
    }
};
```

Listing 3.3: Constructors with *mem-initializers* for base classes

6. *All sub-objects representing virtual base classes are initialized by the constructor of the most derived class (1.8). If the constructor of the most derived class does not specify a* mem-initializer *for a virtual base class* `V`*, then* `V`*'s default constructor is called to initialize the virtual base class subobject. If* `V` *does not have an accessible default constructor, the initialization is ill-formed. A* mem-initializer *naming a virtual base class shall be ignored during execution of the constructor of any class that is not the most derived class.*

3.7 Inline Member Functions

C++ distinguishes between inline and non-inline member functions. In the ANSI Standard [20], this classification is actually used in two senses. First, if a function is declared `inline`, the compiler may decide to insert the function code at any call site instead of actually producing a call. This is recommended for smaller functions because it can be much more efficient to simply insert the function code in-line.

With respect to classes, there is a second meaning, related but slightly different. If a member function is defined inline, the complete function definition is inside the class body. In the non-inline case, the function is only declared within the class body, but the complete function definition is given outside the class declaration, as illustrated in Listing 3.4. This is not possible, for example, in Java, where all member functions are declared and defined inside their respective classes. Member Functions declared inline may be optimized by the compiler as explained in the previous paragraph. But it is technically possible to write a member function non-inline (i.e., outside its class declaration) and to specify the `inline` keyword to allow the compiler to inline the function code as well.

For this paper, the only relevant distinction is whether the function body is defined inside or outside the class declaration. We will therefore use this distinction when referring to inline and non-inline functions.

```
class A {
    void foo() {
        /*
         * inline function definition
         */
    }

    void bar(); // non-inline function declaration

    void baz(); // non-inline function declaration

};

void A::bar() {
    /*
     * non-inline function definition
     */
}

inline void A::baz() {
    /*
     * non-inline function definition, with an indication
     * that the compiler may inline the function code for
     * optimization
     */
}
```

Listing 3.4: Inline and non-inline member functions

Chapter 4

Design

The function of *collapse* is to parse C++ code and transform it so that base class sub-objects are merged in with the most derived classes, resulting in a flat class containing all the member variables and functions as seen from the most derived class. The resulting class therefore does not need any base classes, yet should be able to be used like the original class.

Collapse is designed to act as a preprocessor for C++ code, generating output that can be passed on directly to the compiler. It parses the C++ code generating an *abstract syntax tree* (AST) representing the code, then applies a number of transformations to create the collapsed classes, and finally writes the transformed AST back as C++ code.

4.1 The C++ AST

Collapse parses the input file and generates an internal data structure, an AST that represents the program source found in the input file. The parsing and the AST generation is performed by the libcast library written by Andreas Haeberlen, as part of the IDL^4 project [12]. Libcast's parser is based on that of the GNU C++ compiler [17], which is also the compiler this project focuses on using.

For most constructs available in C++, the AST provides different node types. Each node is actually a classical tree node with certain subelements, but at the same time constitutes a circular doubly-linked list, referencing other nodes of the same type. For instance, a `CASTConstructorDeclaration` links to different subnodes: a `CASTCompoundStatement` to describe the body of the constructor; a list of `CASTDeclarators` describing the constructor's name and its parameters; and a list of `CASTDeclarationSpecifiers` describing the additional specifiers of the constructor, such as `const` or `volatile` specifiers. The `CASTCompoundStatement` referenced by the constructor declaration in turn contains a whole list of statements. A piece of sample code and its AST equivalent are shown in Figure 4.1. The sample shows the AST of a simple class declaration. The class contains four declaration statements; two member variables, a constructor, and a member function. These declaration statements are also arranged in a circular doubly-linked list. The `CASTDeclarationStatement` nodes also contain the visibility information in a `CASTVisibility` subnode; this is not included in the AST.

Collapse walks over this AST to merge all members of the parent classes of any desired class down into this class. It relies heavily on the concept of *visitors* provided by libcast, explained next.

4.2 Visitors

Libcast provides a simple depth-first visitor which traverses the complete AST and takes actions upon encountering nodes depending on the node type. There exist `visit` functions for each node type as hooks. A class derived from libcast's `CASTDFSVisitor` can simply overwrite these hook functions for any node types that are of interest. The existing basic `visit` functions simply call `this::visit` for all subnodes of the current node to provide the depth-first search framework. Since each AST node is more precisely a linked list of nodes, helper functions are used to ensure all available nodes within a node list are visited. A visitor object is somewhat similar to a finite automaton in that it can keep state during tree traversal and reacts on the different nodes upon their encounter as if they were tokens of a data stream. For *collapse*, it was decided to not produce a completely new output AST, but modify the traversed AST in-place. Additionally, of course, traversal may be

```
class A {
private:
    int r;
    double pi;

public:
    A() {
        r = 3;
        pi = 3.14159;
    }

    double a() {
        return (double) (r*r) * pi;
    }
};
```

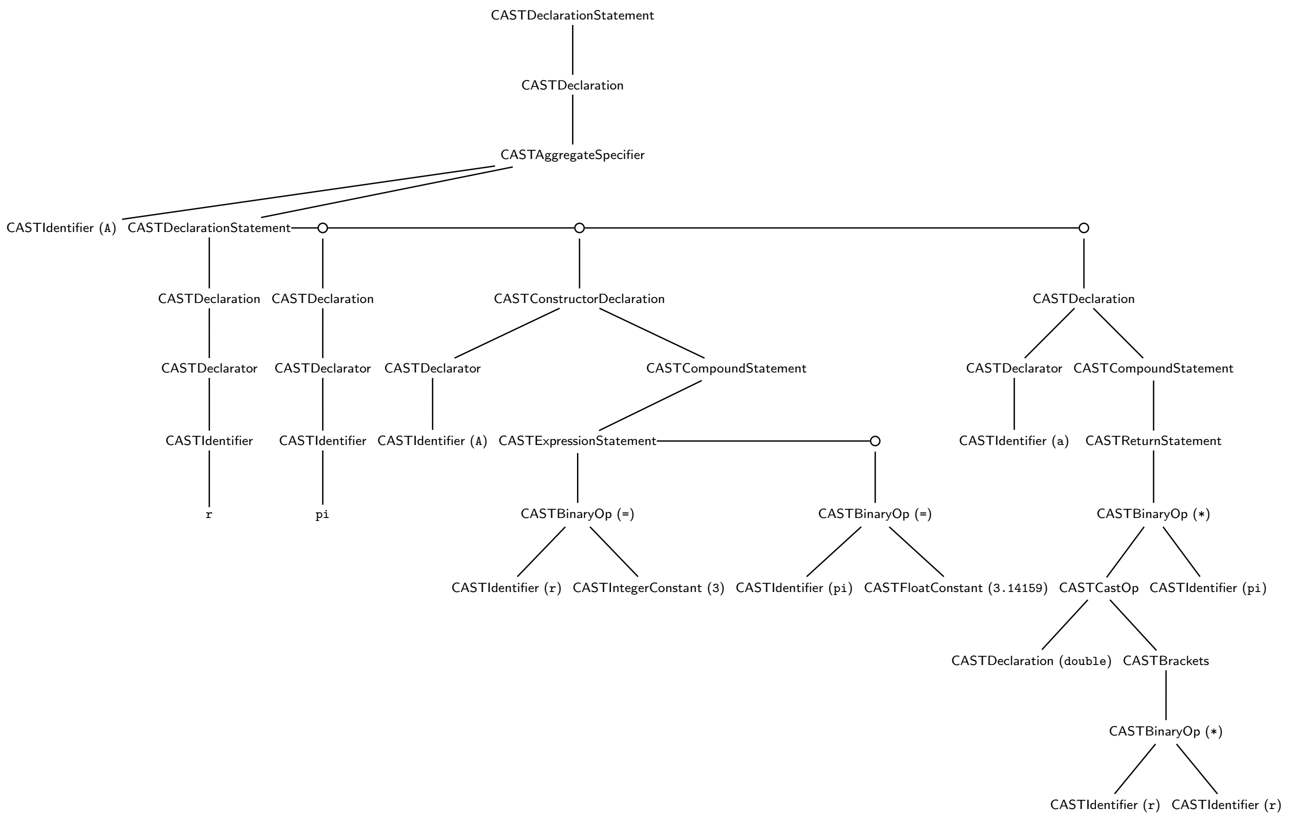

Figure 4.1: A piece of C++ code and the corresponding AST

more complex than just plain depth-first, since a hook function can also decide to visit other branches of the tree instead of just its subnodes.

Collapse uses a number of visitors to extract, modify, and delete information from the AST, and adds new nodes as class members are merged. Many of these visitors not only travel the AST depth-first, but also jump into other subtrees, e.g. when traversing the base class hierarchy of a certain class[1]

4.3 The Transformation

The transformation happens in a number of steps:

1. All class declarations are visited, and references to the corresponding AST nodes are stored in a hash so that they can later be accessed by the class name.
2. For each class specified by the user, all base class members are merged into the class.
3. For each class, `virtual` specifiers are removed as they become superfluous.
4. Optionally, the corresponding base class definitions can be removed to only leave the destination classes.
5. All constructors are converted if desired, keeping semantics intact as much as possible.
6. All duplicate member definitions in the destination classes are removed. During the merge phase, all members have been merged down in such an order that the first definition of a member is the one to keep, and others of the same name must be removed.
7. All base class references are removed from the collapsed class declarations. This completes the collapsing process.
8. If desired by the user, the class member definitions may be sorted.

4.3.1 Finding All Class Declarations

A C++ class declaration is represented in the abstract syntax tree by a `CASTAggregateSpecifier` node which describes its type (class, struct, or union) and holds a `CASTIdentifier` subnode for the class name.

With the help of a visitor, the complete AST is traversed and, upon encountering an aggregate specifier, aggregates of type struct or class are inserted into a hash, indexed by the aggregate's identifier.

4.3.2 Merging Class Members

Collapse keeps the scope and name lookup semantics identical to that of the class before collapsing. In particular, this means that any unqualified member identifiers are first looked up in the class and then its base classes, with members in more derived classes hiding members of the same name in their base classes, as explained in Section 3.5. *Collapse* only keeps unqualified member name lookups intact; member names qualified with a base class name are not transformed and will therefore not work. This also means that member names must be unique except for function overloading; base class members sharing their name with members of a more derived class are not simply hidden, but removed completely.

For each class that is to be collapsed, all members of the base classes are merged down at first. No renaming is done. This means that at first, many duplicate identifiers may accumulate. These will later be removed until all remaining member names are unique again.

The merging visitor differs a bit from the other visitors because it does not completely traverse the AST depth-first. Instead, it visits the base class hierarchy of the class to be collapsed in breadth-first order. This ensures that a hidden member from a base class is added to the class *after* the hiding member has been added. This allows the visitor removing the duplicates to simply keep the first instance of a member name it sees and throw later instances away. The removal of duplicates is described in Section 4.3.6.

When merging members from one base class, all members defined outside the class declaration must also be merged. This concerns member function definitions and the static member declarations. All these definitions

[1] In this case, the visitor could visit the AST depth-first, but upon encounter of a base class specification, jump into the node declaring that base class.

must be copied and qualified with the correct class name. The copies are then added to the AST at the very end so that their order follows the same principles as for inline declarations.

Visibility semantics are transferred in a simplified way. The visibilities for base class members are determined as the maximum of the member visibility and the *access-specifier* for the given base class as denoted in the derived class's *base-specifier-list*.

4.3.3 Removing `virtual` Specifiers

As shown earlier, the main performance hit that *collapse* wants to eliminate is the indirect address lookup for virtual member functions, as explained in Section 1.3. Any `virtual` specifiers within the class declaration must therefore be removed; they are useless in a collapsed class anyway since only the "correct" members remain. A simple visitor takes care of this, visiting the `CASTAggregateSpecifier` subtrees and removing any `virtual` specifiers encountered. All `CASTDeclaration` nodes may include a list of `CASTDeclarationSpecifier` subnodes; a `virtual` specifier appears in that list as a `CASTStorageClassSpecifier` node. Upon encountering such a node with a `virtual` keyword, the node is deleted from the list.

This removal process only concerns those `virtual` specifiers for class members. `virtual` specifiers within base class lists are not removed as they play an important role during constructor conversion.

4.3.4 Removing the Complete Base Class Declarations

By default, the base class declarations are not removed in case instances of those classes are still used in the code. But if only the most derived, collapsed class is of interest, *collapse* can optionally remove the complete base class declarations. To do this, a visitor is used to find and remove a class declaration of a given name. A separate function traverses the class hierarchy depth-first and creates a visitor for each base class to remove it. If a base class of a collapsed class is also collapsed, it is of course not removed. As mentioned before in Section 4.3.1, class declarations are given by a `CASTAggregateSpecifier` node. In addition to deleting this node, its enclosing `CASTDeclaration` parent is removed.

Further, any member function definitions outside the class declaration must also be removed. Member functions are `CASTDeclaration`s whose identifier (found in the `CASTDeclarator` subnode's `identifier` field) is a `CASTNestedIdentifier` bearing the class name as its scope name.

4.3.5 Constructor Conversion

Collapse tries to keep constructor semantics intact as much as possible. This means that the initialization of the class and its base class sub-objects must be done as shown in Section 3.6. To achieve this, all constructors of the base classes are converted into `void` member functions in the collapsed class. Since the collapsed class does not have any base classes, the constructors of the collapsed class must be equipped with appropriate calls to these functions in the constructors. Any of the *mem-initializers* referring to base class constructors must be converted into function calls and inserted into the constructor in the correct order. *Mem-initializers* that initialize data members are not removed since they should still work as intended. In detail, constructor conversion is designed as follows:

- If the class has no constructors, an empty default constructor must be inserted. The following steps then apply to each encountered constructor.
- If the processed constructor belongs to the most derived class, all virtual base classes must be initialized in the correct order. Therefore, the complete inheritance lattice is traversed depth-first, left-to-right according to the order of specification of base classes, searching for virtual base classes. For each such base class, the list of base *mem-initializers* of the constructor is searched for one referring to that base class, and if a *mem-initializer* is found, a call to the corresponding constructor (respectively the `void` function it will be converted to) with the given parameters is inserted into the body of the constructor. Each virtual base class must only be initialized once.

  The virtual base classes can be assumed to provide a default constructor; if it were missing, the program would not be well-formed [20, Section 12.6.2, §6].
- All non-virtual direct base classes are then initialized in declaration order. This applies to constructors of the most derived class as well as any others from the base classes. Again, the *mem-initializer* specification

in the constructor declaration must be taken into consideration, just as in the previous step. A non-virtual base class need not provide any constructor; if none is available, the class is simply left uninitialized.

- Explicit initializers for nonstatic data members are ignored. For the most derived class's constructors they are simply left in place; for the base class constructors, they should also be converted into appropriate expressions which are then inserted into the constructor's body. This is a fairly complicated issue to correctly convert and therefore not currently taken care of.

The list of base classes of a class is given in the `parents` subtree of the `CASTAggregateSpecifier`, which is a node of type `CASTBaseClass`. A `CASTConstructorDeclaration` describes a constructor, its body described in a `CASTCompoundStatement` subtree, and its *mem-initializers* in a `CASTExpression` subtree named `baseInit`. Again, care must be taken to distinguish between constructors declared inside and outside the class declaration.

4.3.6 Removing Duplicate Members

As described in Section 4.3.2, the members in the collapsed class are already ordered so that a visitor for removing duplicate members can process the class declaration in one pass, keeping the first instance of any member and removing any instance that has been seen before. For definitions outside the class declaration, the complete AST is traversed once in the same way. Any member identifiers that are seen must simply be recorded, and upon a second encounter of an identifier the corresponding declaration must be removed.

4.3.7 Removing All Base Class Specifiers

In this rather simple step, the *base-specifier-list* of the collapsed class is emptied so that the class is completely flat. This simply means that the `baseInit` subtree of the `CASTAggregateSpecifier` node must be removed.

4.3.8 Sorting

Finally, the possibility to sort members is given. While this is mostly interesting because the resulting code looks nicer, it also delivers a better binary class structure.

Members can be grouped by visibility and by type (variable/function).

Chapter 5

The Implementation

In this section, details of the implementation are described. General concepts are explained, and the transformation steps as shown in Section 4.3 are discussed.

5.1 Visiting the AST

The visitors in *collapse* do not read the AST and output a completely new AST, but instead modify the AST in place. This places additional burdens on the functions used to traverse the circular lists of nodes, since a visitor may decide to delete its current node or insert additional nodes while the same node list is being traversed. *Collapse* therefore provides special functions to visit all nodes in a list and safely insert and remove nodes from the list, even during traversal.

The visitor function `visitAllSafe()` takes two arguments; a pointer `**head` pointing to a variable which in turn points to the head of the node list, and a pointer `*visitor` to the visitor object which is to be applied to the tree. Since the deletion of the head node means that the head node will change, the indirect pointer is needed so `visitAllSafe()` can react to the change. The function keeps a pointer to the current list head and iterates over the complete list until it arrives back at the list head. Because elements may be removed by the visitor, the value of `*head` must be constantly checked to see if the list is empty, in which case the traversal is finished. Further, elements might be inserted at the list beginning, or the list head may be removed. In these cases, the head changes and the reference pointer must be updated. The code of `visitAllSafe()` is shown in appendix C.1.

When inserting or removing elements from a circular list, the operation might change the list head, when the head is removed or a new element is inserted before the head. Special insert and remove functions were needed that return a potentially updated list head. Appendix C.2 shows the functions `insertElem()` and `removeElem()`.

5.2 Finding All Class Declarations

To be able to find a node for a given class by its name, a reference to each class node is stored in a simple hash. For this implementation, it seemed more efficient to implement a custom hash data structure (`IDHash`) instead of using the C++ Standard Template Libary's `map` structure. The hash structure generates a hash value from the given name by mangling the ASCII values of all characters in the name and their position within the name, modulo the size of the hash. Collisions are resolved by chaining with linked lists. This scheme achieves a fairly uniform distribution that works well enough for the given purposes.

For storing references to all `CASTAggregateSpecifier` nodes, a 512-bucket hash is created, and a visitor of type `ClassHierarchyVisitor` visits the complete tree, filling the created hash. The implementation for this visitor can be found in `hierarchy.cc` within the source distribution.

The `visit()` method for `CASTAggregateSpecifier` nodes ignores any aggregates of type `union`, since, as mentioned before, unions can not be part of a class hierarchy and are therefore not of interest [20, Section 9.5, §1].

Special care must also be taken when encountering incomplete types. Incomplete types [20, Section 3.9, §7] are definitions of types such as

```
class A {
public:
    void a() { /* ... */ }
};

class B : A {
public:
    void a() { /* ... */ }
};

class C {
    /* ... */
}

class D : public B, C {
    /* B::a() is merged first
     * A::a() is merged second
     * A::a() is therefore deleted during
     *        removal of duplicates
     * B::a() remains
     */
};
```

Listing 5.1: Merging member functions

```
class X;
```

where the declarations are missing. The complete definitions might appear at a later point; therefore, references to incomplete types must be stored in the hash as well, and must be replaced upon encountering a corresponding complete definition. If no complete definition is found, the incomplete one might still be needed, for example when removing base class definitions.

5.3 Merging Class Members

The merging process is, along with the constructor conversion, the most complicated process. A visitor of type `MergeAllVisitor` visits the complete AST, merging all class members of any base classes of the given class down into the given class declaration. The base classes must be merged in a particular order so as to keep the correct versions of any member functions: Given a class hierarchy as in Listing 5.1, all versions of `void a()` will be merged at first. The correct one to keep, though, is `B::a()`. To achieve this, base classes are visited in breadth-first, left-to-right order. Later, during removal of duplicates, `A::a()` will appear after `B::a()`, and will therefore be removed.

Suppose for a moment that there were also a function `C::a()`. *Collapse* will then not be able to deliver a semantically correct transformation. The classes are merged in the order `B`–`C`–`A`, and during duplicate removal, only `B::a()` will remain. C++, on the other hand, will complain about calls to `a()` due to the ambiguity, and therefore all calls to `a()` must be disambiguated by explicitly calling `C::a()`, `B::a()`, and so on [20, Section 10.2]. *Collapse* currently does not allow for multiple versions of functions originating from different base classes, nor does it transform any statements within function bodies. In other words, a call to `C::a()` within a member function of `D` will not be transformed, and thus the transformed code will not compile. This means that for code to be transformed correctly, all members must be unambiguous, and calls to base class member functions must not be made using qualified names [20, Section 10.2, §4].

All member declarations within the class declaration are thus merged; additionally all non-inline member definitions must be merged as well. This concerns non-inline member functions as noted in Section 3.7 on page 11, and static member definitions. Therefore not only the aggregate node is visited, but also all general declarations in the AST belonging to the merged class must be copied and given the correct name. If, as in

Listing 3.4 on page 12, `A::bar()` is defined outside the class declaration of `A`, and `A` is a base class of a class `C` which is to be collapsed, then its name must also be changed from `A::bar()` to `C::bar()`. Such names are described in a `CASTNestedIdentifier` node, which in turn points to two `CASTIdentifier` nodes for the class name and the member name. The merging visitor takes care of changing the class name as well.

5.4 Devirtualizing All Methods

Since the classes are being collapsed, it makes no difference regarding semantics whether their member functions are virtual or not. Our goal of improving the performance ultimately demands that the `virtual` specifiers be removed. Therefore, the file `killvirt.cc` defines the `DevirtualizeMethodsVisitor` which simply visits the collapsed classes and removes any `virtual` specifiers. Each class that has been collapsed is then visited by this visitor.

5.5 Removing the Base Classes

The `KillClassVisitor` is used to completely remove a given class from the AST. This involves removing the aggregate specifier and any non-inline member function declarations, as well as additional incomplete definitions of the class as mentioned in 5.2. A helper function `killBaseClasses()` is used to recursively visit all base classes of a collapsed class depth-first and run a `KillClassVisitor` on each of these base classes.

5.6 Constructor Conversion

Constructor conversion follows the steps as explained in Section 4.3.5. All constructors merged in from base classes are first converted to `void` functions. *Collapse* converts a constructor `A()` to a function `void __CTOR_A()`. A new `CASTIdentifier` is built from the existing one bearing the new name, and a new `CASTDeclaration` with the new identifier, the original declarators and the original compound statement is created. All thus created `void` functions are created with private visibility.

Copying the base class initialization semantics means inserting calls to these `void` functions at the beginning of any constructor of the collapsed class. If the most derived class has no constructors, an empty default constructor must be created which is then filled with the appropriate calls to initialize the base classes. To insert an empty default constructor, *collapse* creates a `CASTConstructorDeclaration` that contains a compound with only an empty statement, a `CASTDeclarator` holding the identifier and a `CASTEmptyDeclaration` for its `parameters` list. This default constructor is inserted with public visibility.

To initialize all virtual base classes, a helper function `insertVirtualBaseInits` visits the base classes recursively, depth-first and left-to-right, distinguishing between virtual and and non-virtual base classes. For each virtual base class found, the list of *mem-initializers* (denoted in `CASTConstructorDeclaration::baseInit`) is searched for a corresponding initializer. If none is found, a call to the default constructor is added to the currently processed constructor body. The function call is a `CASTExpressionStatement` containing a `CASTFunctionOp`, which in turn simply holds a `CASTIdentifier` with the name of the constructor. Note that for a base class `A`, not `A::A()` is called, but rather the void function `__CTOR_A()` created from the constructor. If a `baseInit` entry is found, it is converted to a call to the appropriate `__CTOR_` function.

For all non-virtual direct base classes, the `baseInit` list containing all *mem-initializers* for the constructor is searched for an initialization call. If none is found, and a default constructor is available for the base class, the base class sub-object is initialized with a call to that default constructor. If a *mem-initializer* exists, it is converted to a function call with the given parameters.

A separate visitor, `CtorVisitor` (in `ctors.cc`), takes care of constructor conversion, and is run once for every collapsed class.

A complex example, which was used as a test case to ensure the proper functioning of *collapse*, can be found in appendix B.

5.7 Removing Duplicate Members

As mentioned above in Section 5.3, all members are at first merged down, and duplicate members are removed later on. This happens in this phase. A `KillDuplicateMemberVisitor` visits the complete AST once for

each collapsed class. It uses an IDHash to keep track of member names. Whenever it encounters a member name that has already been present, it removes that member declaration. Again, if the declaration is a non-inline function declaration, the complete function definition also has to be removed. This means that duplicate member declarations are removed once within the aggregate, and again within the global scope, looking for any member definitions.

Since the IDHash stores strings, the member variables and functions must be identified by a unique string. For variables, this is simply the variable name; for functions, it is the function name plus the signature. For instance, the two functions

```
void foo(int a, char *b);
int bar(char &c, int a[]);
```

are identified as "foo(int char*)" and "bar(char& int[])".

5.8 Removing the Base Class References

During this phase, the base class references of the collapsed classes are simply removed. That means, a collapsed class

```
class Foo : virtual public A, B, private C {
    /* declarations */
};
```

becomes, after being visited by the FlattenClassVisitor,

```
class Foo {
    /* declarations */
};
```

All that needs to be done is to set the parents element of the corresponding CASTAggregateSpecifier to NULL.

5.9 Sorting

Sorting the members of a collapsed class is an optional functionality. *Collapse* can sort members by visibility so that all public, protected, and private members are grouped, and, additionally, member functions and member variables can be sorted separately so that a class declaration contains all member variables followed by the member functions (or vice versa). This functionality is implemented in the file sort.cc. The sortMembers() function simply keeps six different doubly linked lists of declaration statements, to account for functions and variables at all three visibility levels. The six lists are then interconnected in the desired order, and the aggregate specifier's declarations entry is updated with the new list head.

It would be helpful to extend these sorting capabilities so that the user can give an explicit ordering of members, including alignment specifications. This is especially interesting in the L4 case, where member order and alignment can have an important influence on microkernel performance.

Chapter 6

Scope and Limitations of the Work

6.1 Visibility Semantics

Collapse either merges private members of base classes or ignores them. This leads to a change in visibility; a private member variable of a base class can normally be accessed by a member function of the base class. A member function of a derived class, though, can not access the member variable, but may call a public base class function to read or change the variable. This distinction is currently not possible in *collapse*. If a private member variable is merged, any function in the merged class may access it; if it is not merged, not even functions merged from the base class can access it.

This problem may be resolved with a fairly great deal of variable renaming, but for this project, such a complete implementation was considered out of scope.

6.2 Base Class Semantics

Collapse will not produce correct results if members of a base class are used in a derived class with an explicit qualified name such as `A::foo`. Obviously, if `A` is merged into a derived class, that name is no longer valid. *Collapse* does not perform any rewriting of such names.

If such renaming were in place and all merged code were processed with a renaming visitor, it could even be possible to have one class used as a base class multiple times on different paths with ambiguous names as shown in Figure 3.1 on page 8.

This sort of refinement could be possible and could greatly enhance the semantic equivalence of collapsed output. But again this was not considered necessary within the scope of this study thesis.

6.3 Namespaces

The libcast parser currently can not deal with namespaces correctly, and the AST provides no nodes for namespace declarations. As a consequence, *collapse* does not support namespaces either.

Adding support for namespaces would make the implementation more complicated, as all classes would have to be qualified with their full namespace to determine their exact identity. Currently, a class `A` and a class declared in a namespace, `nmspc::A` would both appear as `A` to collapse since, even if namespaces were supported, *collapse* would not consider them as part of the name.

Name lookups would be more complicated as well, since a name would have to be looked up in the normal scope as well as all active namespaces depending on `using namespace` keywords.

For the special environment in which we plan to use *collapse*, namespaces are not needed. Of course namespace support would be a great advantage, and possibly easier to implement than complete visibility semantics. It would, however, require a much more complicated semantic analysis.

6.4 Ordering and Alignment

Currently, *collapse* gives the user the possibility to at least sort members by their visibility and by variables and functions. To further automatically optimize collapsed data structures, it would be desirable to be able to specify an explicit ordering of members, and maybe even specify alignments for member variables. This extension to *collapse* could be implemented in a separate project.

6.5 Constructors and Member Initialization

Libcast can not currently parse *mem-initializers* for member variables. It only deals well with those for base class sub-objects. For a class `A` containing a member `int A::foo`, it is therefore not possible to write a constructor `A(int f) : foo(f)`. For this reason, *collapse* can not take care of such *mem-initializers* during constructor conversion either.

When a variable is of an object type and not a pointer to an object (for instance, by writing `Foo f;` instead of `Foo *f;` in some function body), libcast can not handle parametrized initializations calling a specific constructor. For an object pointer, writing `Foo *f = new Foo(3);` can be dealt with correctly. But declarations such as `Foo f(3);` can not be parsed by libcast.

6.6 Destructors

Unfortunately libcast from the current IDL[4] release can not parse destructors correctly, and has no way to represent them in the AST. Therefore, *collapse* does not handle destructors either. If it did, it would have to convert all base class destructors as well and ensure that they are being called in the correct order from the derived class's destructor. This should not be too much of a problem once libcast supports destructors.

6.7 Semantic Checking

Collapse currently does not spend much time on performing checks for semantic correctness; instead, it tries to process any given code and lets the C++ compiler report any errors. For example, *collapse* prefixes converted constructors with `__CTOR_` without checking whether a user-defined function of such a name already exists. The user could also specify calls to a function of such a name, knowing that it will exist after collapsing. The first case will yield an ill-formed output program while the second case is only well-formed after collapsing. All this error checking is left to the compiler. Preferably, *collapse* should generate much more warnings.

Chapter 7

Performance

7.1 Virtual vs. Non-virtual Function Calls

As could be seen in the generated code shown in Listing 1.2 on page 3, calls to virtual functions require more code and can thus generally be expected to be slower. A simple benchmark was constructed, making one billion calls to two different virtual functions. In comparison, the collapsed version takes about 72 percent of the time the uncollapsed version takes to run[1]. Even though this benchmark is hardly suitable for real-world cases, it shows that some speedup is possible in time-critical cases such as context switching performed by a microkernel. Naturally, such a result had to be expected since we have already shown in Section 1.1 that calling a non-virtual function is much simpler.

7.2 Build Process Overhead

The implementation has not yet been applied to large-scale real-world software systems. An artificial benchmark was created to test *collapse*. Class hierarchies of varying complexity were generated by using different depth and branching factor parameters for the inheritance tree. Comparing the time to compile the generated file with and without collapsing produced the results seen in table 7.1.

Number of Classes	g++	collapse and g++	speedup
1102	408s	30s	14x
2613	2098s	101s	20x
8315	died after 676s	641s	n/a

Table 7.1: Execution times for compiling a machine-generated example class hierarchy with and without collapsing.

[1]Performed on an AMD Athon XP at 1533 MHz. Averaged over three runs, the timings were 20.55 s (collapsed version) vs. 28.47 s (uncollapsed version), i.e. the collapsed version took 72.18% of the time.

Appendix A

Functionality

A.1 Using collapse

Collapse preprocesses a C++ source code file and completely collapses the class hierarchy for certain user-specified classes, making them completely flat, without any base class references or virtual member functions.

The usage is as follows:

`collapse [options] [`*`file`*` [`*`class`*`...]]`

The input file *`file`* is processed, and all specified classes *`class`* are collapsed. If no input file name is given, standard input is processed. The following options are available:

`-o, --output=`*`file`* Allows the user to specify the output file name. As a special case, a file name of '`-`' denotes standard output. By default, the output file name is the input file name with a suffix of `.ii`.

`-c, --collapse=`*`class`* This specifies the name of a class whose hierarchy should be collapsed into the class. Specifying classes here is identical to specifying them after the input file name, but if one wants to process the standard input, this option is the only way to specify classes.

`-r, --remove-base` If this option is given, *collapse* will completely remove all base class declarations of the collapsed classes from the source.

`-C, --convert-ctors` This activates constructor conversion as explained in Sections 4.3.5 and 5.6.

`--ignore-private` This changes the semantic behavior of *collapse*. Normally, private members of base classes are merged into the destination class even though that makes them visible to the destination class members as well, because this allows public base class functions that access private members of their class to still work. If private members are not to be merged, use this option.

`-s, --sort=`*`mode`* By default, the resulting class with all relevant members merged contains these members in the order it discovered them. The `--sort` switch allows them to be sorted in one of several ways:

- `NONE` This is identical to the default behavior of not sorting.
- `VISIBILITY` Members are grouped by visibility, such that all private members are listed together, followed by all protected members and all public members.
- `FUNCS_FIRST` Member functions appear before any member variables at the beginning of the class declaration. Functions and variables are additionally sorted by visibility.
- `VARS_FIRST` This works like the previous mode, except that now member variables appear before functions.

`-D `*`definition`* Adds a preprocessor definition. *Collapse* itself adds one preprocessor definition, `__COLLAPSE__`, its value representing the version number.

`-I `*`directory`* Specify an additional include directory used during the preprocessing stage.

`-W `*`warning`* Enable specific warnings. *Collapse* currently uses no warnings, but one can pass through options to the preprocessor this way, using '`-W p,`*`preprocessor-option`*'.

`-P, --with-cpp=`*path* Use this option to specify another preprocessor to use. By default, *collapse* uses `/usr/bin/cpp`.

`-d, --debug=`*mode* Enable more verbose debug output. There are five different levels of debug verbosity: `QUIET`, `BRIEF`, `VERBOSE`, `DETAILED`, and `INSANE`. Each displays more debugging output. Alternatively, each time an option '`--debug`' without a specified level is encountered, the current debug level is raised.

`-V, --version` This displays some version information and exits.

`-h, --help` Displays the help screen.

A.2 autoconf Support

Collapse supports GNU autoconf [21]. To facilitate the use of *collapse* in an autoconf-controlled project, a simple m4 macro is included. With this macro, one only needs to add a line containing the text "`AC_PROG_COLLAPSE`" to the file `configure.in`. This checks and, if necessary, sets an environment variable `COLLAPSE` for the path to *collapse*. Files that need to be preprocessed by *collapse* should be given a unique file name suffix, e.g. `.ci`.

In the Makefiles (usually called `Makefile.am`), the source files that need to be created by applying *collapse* to some input files are specified through the automake variable `BUILT_SOURCES`. Finally, any sources built by *collapse* should be mentioned in an explicit Makefile target, such as

```
test.cc: test.ci
        $(COLLAPSE) -c SomeClass test.ci -o test.cc
```

Appendix B

A fairly complex example for constructor conversion

Listing B.1 shows a more complex class hierarchy which is also displayed in Figure B.1. Each of the constructors simply contains code to initialize the class's data members, either with a default value or with the value given in the parameter. `C` and `D` are (obviously) virtual base classes.

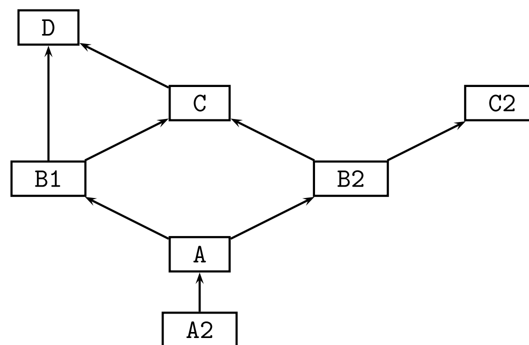

Figure B.1: Inheritance graph for Listing B.1

Upon creation of different objects of the given classes, base class sub-objects are initialized in this order:

`new A()`: First, virtual base classes are initialized: `D()`, then `C()`. Next, non-virtual direct base classes are initialized: `B1()`, `C2()`, `B2()`, and finally, `A()`.

`new A(3)`: Virtual base classes first: `D()`, then `C()`. After that, `B1(4)`, `C2()`, `B2()`, and lastly `A(3)`.

`new A(2, 4)`: `D(4)`, `C()`, then `B1(3)`, `C2(5)`, `B2(4)`, and `A(2, 4)`.

`new A2()`: the order is identical to that of `new A()`; `A2` is initialized last, but nothing is done since there is no constructor for `A2`.

Listing B.2 shows the collapsed version of `A`, exhibiting the correct behavior.

```
class D {
public:
    int d;
    D() { /* .. */ }
    D(int d) { /* .. */ }
};

class C : virtual public D {
public:
    int c;
    C() : D(1) { /* .. */ }
    C(int c) { /* .. */ }
    C(int c, int d);
};

C::C(int c, int d) : D(d) { /* .. */ }

class C2 {
public:
    int c2;
    C2() { /* .. */ }
    C2(int c2) { /* .. */ }
};

class B1 : virtual public C, virtual public D {
public:
    int b1;
    B1() : D(3) { /* .. */ }
    B1(int b1) { /* .. */ }
};

class B2 : virtual public C, public C2 {
public:
    int b2;
    B2() { /* .. */ }
    B2(int b2) : C2(b2+1) { /* .. */ }
};

class A : public B1, public B2 {
public:
    int a;
    A() { /* .. */ }
    A(int a) : B1(a+1) { /* .. */ }
    A(int a, int d) : B1(a+1), B2(a+2), D(d) { /* .. */ }
};

class A2 : public A {
};
```

Listing B.1: A complex class hierarchy with constructors

```
class A {
public:
    int a;
    A(void) {
        __CTOR_D(); __CTOR_C(); __CTOR_B1(); __CTOR_B2();
        /* ... */
    }
    A(int a) {
        __CTOR_D(); __CTOR_C(); __CTOR_B1(a+1); __CTOR_B2();
        /* ... */
    }
    A(int a, int d) {
        __CTOR_D(d); __CTOR_C(); __CTOR_B1(a+1); __CTOR_B2(a+2);
        /* ... */
    }
    int b1; int b2; int c; int d; int c2;
private:
    void __CTOR_B1() { /* ... */ }
    void __CTOR_B1(int b1) { /* ... */ }
    void __CTOR_B2() { __CTOR_C2(); /* ... */ }
    void __CTOR_B2(int b2) { __CTOR_C2(b2+1); /* ... */ }
    void __CTOR_C() { /* ... */ }
    void __CTOR_C(int c) { /* ... */ }
    void __CTOR_C(int c, int d);
    void __CTOR_D() { /* ... */ }
    void __CTOR_D(int d) { /* ... */ }
    void __CTOR_C2() { /* ... */ }
    void __CTOR_C2(int c2) { /* ... */ }
};

void A::__CTOR_C(int c, int d) { /* ... */ }
```

Listing B.2: Collapsed version of the class A from Listing B.1

Appendix C

Code excerpts from **collapse**

C.1 A Visiting Algorithm

`visitAllSafe()`, as explained in Section 5.1, traverses a circular doubly-linked list of AST nodes, even if that list is modified during traversal. Listing C.1 shows the algorithm.

C.2 Inserting and Removing List Elements

When removing an element from a circular node list, it is obviously possible that the head element gets removed. The `removeElem()` function is therefore given the current list head and returns the updated list head after element removal.

If an element is inserted before the list head, it becomes the new list head. `insertElem()` therefore works similar to `removeElem()` and returns a potentially updated list head. The code for these functions is shown in Listing C.2. To append an element at the list tail, it must be inserted at the list head without updating the list head pointer.

```
void visitAllSafe(CASTBase **head, CASTDFSVisitor *visitor) {
  CASTBase *iterator = *head;
  CASTBase *refHead = *head;
  if (iterator) {
    while (true) {
      iterator->accept(visitor);
      if (!(*head)) break;                       /* no more head              */
      if (*head != refHead) {                    /* head changed              */
        if (iterator == refHead) {               /* we're on the head right now */
          if (refHead->next->prev != refHead) { /* We were removed           */
            iterator = refHead = *head;          /* Start at new head         */
            continue;                            /* Make sure to process it   */
          }
        }
        refHead = *head;                         /* Simply acknowledge new head */
      }
      iterator = iterator->next;
      if (iterator == *head) break;
    }
  }
}
```

Listing C.1: An algorithm for safely visiting lists while they are modified

```
CASTBase *removeElem(CASTBase *head, CASTBase *elem) {
    if (!head) return NULL;              // Nothing to do
    if (elem->next == elem) return NULL; // Last element removed
    elem->next->prev = elem->prev;       // Unchain ourselves
    elem->prev->next = elem->next;
    if (elem == head) return elem->next; // List head was removed
    return head;                         // List head did not change
}

CASTBase *insertElem(CASTBase *head, CASTBase *where, CASTBase *elem) {
    if (!head) return elem;              // This is the first element
    if (!where) where = head;            // Insert at the beginning
    where->add(elem);                    // Add
    if (where == head) return elem;      // New element is new head
    return head;                         // List head did not change
}
```

Listing C.2: Inserting and removing list elements

Bibliography

[1] Y.-F. Lee and M. J. Serrano. Dynamic measurements of C++ program characteristics. Technical Report ADTI-1995-001, IBM Santa Teresa Laboratory, January 1995.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96 Conference Proceedings*, pages 324–341, San Jose, CA, October 1996.

[3] The L4Ka web site. `http://www.l4ka.org/`.

[4] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[5] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre Cointe, editor, *ECOOP '96—Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer, 1996.

[6] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.

[7] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000.

[8] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–332, 1998.

[9] Sara Porat, David Bernstein, Yaroslav Fedorov, Joseph Rodrigue, and Eran Yahav. Compiler optimization of C++ virtual function calls. In USENIX, editor, *Proceedings of the Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 17–21, 1996, Toronto, Canada*, pages 3–14, Berkeley, CA, USA, 1996. USENIX.

[10] University of Magdeburg OS Research Group. PUMA - the PURE manipulator. `http://wwwivs.cs.uni-magdeburg.de/~puma/puma-eng.html`.

[11] University of Magdeburg OS Research Group. AspectC++. `http://www.aspectc.org/`.

[12] Andreas Haeberlen. *Using Platform-Specific Optimizations in Stub-Code Generation.* Study thesis, University of Karlsruhe, July 2002.

[13] The OpenC++ project web site. `http://opencxx.sourceforge.net/`.

[14] The CPPX open source C++ fact extractor. `http://swag.uwaterloo.ca/~cppx/`.

[15] The Graph Exchange Language. `http://www.gupro.de/GXL/`.

[16] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE '00*, November 2000.

[17] The GNU Compiler Collection. `http://gcc.gnu.org/`.

[18] Workgroup on standard exchange format. `http://www.cs.toronto.edu/~simsuz/wosef/`.

[19] Bell-Canada. DATRIX Abstract Semantic Graph Reference Manual.

[20] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998. Available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`.

[21] GNU Autoconf. `http://www.gnu.org/software/autoconf/`.