1995

Implementing Signatures for C++

Gerald Baumgartner

Vincent R. Russo

Report Number:
95-025

IMPLEMENTING SIGNATURES FOR C++

Gerald Baumgartner
Vincent F. Russo

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR-95-025
August 1995

Implementing Signatures for C++*

Technical Report CSD-TR-95-025

Gerald Baumgartner Vincent F. Russo
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

gb@cs.purdue.edu russo@cs.purdue.edu

August 11, 1995

Abstract

We outline the design and detail the implementation of a language extension for abstracting types and for decoupling subtyping and inheritance in C++. This extension gives the user more of the flexibility of dynamic typing while retaining the efficiency and security of static typing. After a brief discussion of syntax and semantics of this language extension and examples of its use, we present and analyze the cost of three different implementation techniques: a preprocessor to a C++ compiler, an implementation in the front end of a C++ compiler, and a low-level implementation with back end support. Finally, we discuss the lessons we learned for future programming language design.

1 Introduction

In C++, as in several other object-oriented languages, the *class* construct is used to define a type, to implement that type, and as the basis for inheritance, type abstraction, and subtype polymorphism. We argue that overloading the class construct limits the expressiveness of type abstraction, subtype polymorphism and inheritance. We remedy these problems by introducing a new C++ type definition construct: the *signature*. Signatures provide C++ with a type system that allows for clean separation of interface from implementation and achieves more of the flexibility of dynamic typing without sacrificing the efficiency and security of static typing.

The remainder of the paper is structured as follows. First we present motivation for the addition of a type abstraction facility other than classes to C++. We then briefly present syntax and semantics of the core constructs of our language extension and follow with examples that illustrate how signatures solve the problems presented in the motivation section. The core sections of the paper discuss and compare three different implementation possibilities, and analyze the costs of each. We conclude with a discussion on the lessons we learned from this experiment and their implications for future programming language design.

Since the primary intent of this paper is to detail these implementation techniques, the motivation and language specification are of necessity brief. The reader interested in a more detailed motivation and complete syntax and semantics is referred to [2].

2 Motivation

Using inheritance as a subtyping mechanism suffers from three specific problems:

*Submitted to *ACM Transactions on Programming Languages and Systems.*

1. Using the same construct (class inheritance) for type abstraction and code sharing limits the power of both and unnecessarily couples implementation and interface specifications.
2. In some cases, it is difficult (if not impossible) to retroactively introduce abstract base classes to a class hierarchy for the purpose of type abstraction.
3. The hierarchy of abstract types and the class hierarchy of implementations may be difficult to reconcile with each other.

We will show how signatures allow us to overcome these problems without a major overhaul of the C++ type system.

2.1 Separation of Type and Class Hierarchies

A problem with a single class hierarchy defining both abstract types and their implementations is that as the type hierarchy becomes more complex, it might become necessary to duplicate code. We use an example from computer algebra [3, 2] to demonstrate this problem.

Consider the abstract type `GeneralMatrix` with subtypes `NegativeDefiniteMatrix` and `OrthogonalMatrix`. Both subtypes have functions that are not present in general matrices, e.g., the function `inverse()`. Assume we have several different implementations of these abstract types, namely `DenseMatrix`, which implements matrices as two-dimensional arrays, `SparseMatrix`, which uses lists of triples, and `PermutationMatrix`, which is implemented as a special case of sparse matrices that takes advantage of permutation matrices only having one element in each row and column.

If we try to model these types and implementations with a single class hierarchy, we end up either duplicating code or violating the type hierarchy. While `DenseMatrix` can be made a subclass of the abstract classes `GeneralMatrix`, `NegativeDefiniteMatrix`, and `OrthogonalMatrix` by using multiple inheritance, we cannot do the same for `SparseMatrix`. Doing so would make `PermutationMatrix`, which is a subclass of `SparseMatrix`, an indirect subclass of `NegativeDefiniteMatrix`. Since permutation matrices are positive definite, this would violate the type hierarchy. The alternative of having a separate class `SparseNegativeDefiniteMatrix` is not satisfying either since it causes code replications.

Similar arguments have been given in the literature to show that the `collection` class hierarchy of SMALLTALK-80 [13] is not appropriate as a basis for subtyping. While the problem does not arise with dynamic typing, it becomes an issue when trying to make SMALLTALK-80 statically typed while retaining most of its flexibility. The solution is to factor out the implementation aspect of classes into prototypical objects [16] or to factor out the type aspect into interfaces [5, 8].

2.2 Retroactive Type Abstraction

Another practical example [14] illustrates the need to introduce type abstractions of existing class hierarchies. Summarizing their presentation, suppose we have two libraries containing hierarchies of classes for X-Window display objects. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. Further suppose all the classes in each hierarchy implement virtual `display()` and `move()` member functions, and that both libraries are supplied in "binary-only" form. Can a display list of objects be constructed that can contain objects from *both* class libraries *simultaneously*? The answer is yes, but not without either explicit type discrimination or substantial software engineering costs due to the introduction of additional classes.

Obviously, the straightforward solution would be to create a common abstract superclass for both hierarchies. However, if only header files and binaries but no source code are available for the two libraries, retroactive code modification is not possible. If the member functions needed for the abstract type are non-virtual member functions, introducing an abstract superclass is not possible either, since it would modify the behavior. The only choices remaining are to use a discriminated union for the display list elements, to use multiple inheritance to implement a new set of leaf classes in each hierarchy, or to use a hierarchy of forwarding classes.[1] The former solution is rather inelegant, the latter two clutter up the name space with

[1] In C++, the task of creating these leaf and forwarding classes can be simplified using templates.

a superfluous set of new class names.

The problem is that C++ provides only one type abstraction mechanism, the class, and that implementations must explicitly state their adherence to an abstract type by inheriting from the abstract class. The nature of the restriction to binaries in this example prevents us from doing this. What we would like is a type abstraction mechanism that does not rely on classes and, therefore, leaves classes free to be used for implementation specification. Likewise, the adherence of a particular class to an abstract type would ideally be inferred from the class specification and not need to be explicitly coded in the class. This leaves us free to introduce new abstract types at a later time without altering any implementations.

A more realistic scenario for retroactive type abstraction would be that only one implementation is given in compiled form and that we would like to abstract the type of some of the given classes and provide an alternative implementation. If the original implementation was not designed with this form of reuse in mind, or if the alternative implementation uses different data structures, we end up with the same problems as above.

2.3 Implementation of Conflicting Type and Class Hierarchies

Often the abstract type hierarchy and the implementation class hierarchy cannot be made to agree. An example similar to one in [20] illustrates this point. Consider two abstract types `Queue` and `DEQueue` (doubly ended queue). The abstract type `DEQueue` provides the same operations as `Queue` as well as two additional operations for enqueuing at the head and for dequeuing from the tail of the queue. Therefore, `DEQueue` is a *subtype* of `Queue`.

However, the easiest way to implement `Queue` and `DEQueue` is to structure the inheritance hierarchy opposite to the type hierarchy. A doubly ended queue is implemented naturally as a doubly linked list. A trivial implementation of queue would be to copy the doubly ended queue implementation through inheritance and remove, or ignore, the additional operations.

In [9], it is argued that in order for a type system to be sound it should not be possible to use inheritance for subtyping purposes and also allow the removal of operations. Most object-oriented languages choose instead to restrict the use of inheritance for code sharing to situations where there is also a subtype relationship, and to disallow inheriting only a portion of the superclass.

3 Syntax and Semantics of the Signature Language Extension

We term the key language construct we add to C++ to support type abstraction a *signature*. A signature declaration defines an abstract type by specifying the member functions that any implementation of the abstract type needs to have. The signature language construct is related to types in RUSSELL [10], ML's signatures [17, 18], HASKELL's type classes [11], definition modules in MODULA-2 [24], interface modules in MODULA-3 [7], abstract types in EMERALD [4], type modules in TRELLIS/OWL [19], categories in AXIOM [15] and its predecessor SCRATCHPAD II [22, 23], and types in POOL-I [1].

To associate an implementation with a signature type, we introduce the notion of a *signature pointer* into the language. For an assignment of an object pointer to a signature pointer, the compiler verifies that the class implements all the member functions declared in the signature, i.e., it insures that the class structurally *conforms* to the signature. When calling a signature member function through a signature pointer, the appropriate class member function will be invoked.

The type system of C++ with signatures comes closest to those of AXIOM and POOL-I. RUSSELL, ML, HASKELL, and MODULA-2 do not have class types, MODULA-3 only has interfaces for modules but not for classes. EMERALD has first-class types instead of classes, and TRELLIS/OWL has a type hierarchy in which type information but no implementation is inherited. Domains in AXIOM differ from classes by having method dispatch on all argument types and on the return type. Compared to C++, POOL-I does not have private and protected member functions and overloading. While both categories and domains in AXIOM and types in POOL-I are first class, signatures and classes in our C++ extension are not, which makes the type

system slightly less expressive but allows for a more efficient implementation and for complete type checking at compile time.

In this section, we describe only those parts of our language extension that are relevant to contrasting the different implementation techniques discussed later in the paper. Specifically, this section details the syntax and semantics of signatures, signature pointers, and signature references. We also explain the semantics and utility of default implementations and constants in signatures.[2]

3.1 Signature Declarations

A signature type is declared in a way similar to a class except the keyword `signature` is used instead of `class`, or `struct`, to introduce a signature declaration.

A signature declaration, like a class declaration, defines a new C++ type. The key difference is that a signature declaration contains only *interface descriptions*. For example, the signature declaration

```
signature T {
    int * f ();
    int g (int *);
    T & h (int *);
};
```

defines an abstract type `T` with operations (member functions) `f`, `g`, and `h`.

The specific difference from a class declaration is that only type declarations, constant declarations, member function declarations, operator declarations, and conversion operator declarations are allowed within a signature declaration. Specifically:

- A signature cannot have constructors, destructors, friends, or data member declarations.
- The visibility specifiers `private`, `protected`, and `public` are not allowed either in the signature body or in the base type list. They are unnecessary since signatures define interfaces and, therefore, all members are implicitly public.
- Signature base types have to be signatures themselves (a signature cannot inherit from a class). Similarly, a signature cannot be the base type of a class.
- The type specifiers `const` and `volatile` are not allowed for signature member functions, since they are storage location specifiers and are meaningless for members of an interface specification.
- The storage class specifiers (`auto`, `register`, `static`, `extern`), the function specifiers `inline` and `virtual`, and the pure specifier `=0` are not allowed. The latter two are needed in class declarations only to specify abstract classes and are, therefore, superfluous in signature declarations.

In the absence of a more complex type hierarchy, the type `T` in the above example could have been defined as an abstract class, i.e., a class containing only pure virtual member function declarations [12]. The behavior of both implementations would be similar except that classes implementing the abstract class's interface need to explicitly code that fact by inheriting from the abstract class. When using signatures to specify abstract types, this relationship is, instead, inferred by the compiler.

As a *type* hierarchy becomes more complex it becomes more and more difficult to model it precisely with a class hierarchy as shown in the computer algebra example. Signatures allow to build a type hierarchy structured independently from the class hierarchy. This enables more complex type hierarchies and facilitates the decoupling of subtyping and inheritance. Also, signatures can be used to define type abstractions of existing class hierarchies. With abstract classes, it would be necessary to retrofit abstract classes on top of the existing class hierarchy. This cannot be done without recompiling all existing source files. Signatures, therefore, improve C++'s capabilities for reusing existing code.

[2]The additional features of signature inheritance, the `sigof` construct (as in [14]), views, and opaque types are left out since they only affect the type checking phase of the compiler. For information on those constructs, as well as for more details on the semantics of signatures, see [2].

3.2 Signature Pointers and References

Since a signature declaration only describes an *abstract* type, it does not give enough information to create an implementation for that type. For this reason, it is nonsensical (and not valid) to declare objects of a signature type, as in

```
signature S { /* ... */ };
S obj;   // illegal! 'S' is an interface type
```

Instead, in order to associate a signature type with an implementation, we declare a *signature pointer* or a *signature reference* and assign to it the address of an existing class object. Signature pointers and signature references, therefore, can be seen as *interfaces* between abstract (signature) types and concrete (class) types.

Consider the following declarations,

```
signature S { /* ... */ };
class C { /* ... */ };
C obj;
S * p = &obj;  // legal if 'C' conforms to 'S'
```

For the initialization of the signature pointer p, or for an assignment to p, to be type correct, the class type C has to *conform* to the signature type S. I.e., the implementation of C has to satisfy the interface S, or the signature of C has to be a sub*type* of S.

A signature pointer or reference can also be assigned to another signature pointer or reference. In this case, the right hand side signature must conform to the left hand side signature, or in other words, the right hand side signature must be a subtype of the left hand side signature.

A signature pointer can also be assigned to, or implicitly converted to, a pointer of type `void*`. To assign a signature pointer to a class pointer, it is necessary to use an explicit type cast:

```
S *    p = new C;
void * q = p;          // ok
C *    r = p;          // error: explicit cast necessary
```

In general, we do not know the class of the object pointed to by a signature pointer. Assigning a signature pointer to a class pointer is therefore, like casting down the class hierarchy, an unsafe operation. The same is true for signature references.

3.3 The Conformance Check

The *conformance check* is the type check performed when initializing or assigning to a signature pointer or a signature reference. Except for the very rare case described below, the design and implementation of signatures implies no *run-time* cost for the conformance check. The conformance check is done at *compile* time.

To test whether a class C conforms to a signature S, the structures of C and S must be recursively compared. The specific conformance rules are:

1. For every member function, operator, and conversion operator declared in S, there must be a public declaration of the same member function or operator in C. Furthermore, this declaration must have the same name and conforming return and argument types. Also, every signature contains an implicit destructor declaration. This destructor is matched with the class's destructor if defined or with the default destructor otherwise. Specifically, a class member function `C::f` conforms to a signature member function `S::f` if and only if the following conditions hold:

   - The type of every argument of `S::f` conforms to the type of the corresponding argument of `C::f`.
   - The return type of `C::f` conforms to the return type of `S::f`.

- If `S::f` has an exception specifier, `C::f` must have an exception specifier as well, which only lists (a subset of) the exceptions specified by `S::f`.

Any default values of corresponding arguments of `S::f` and `C::f` are ignored for purposes of the conformance check. `C::f` can have more arguments than `S::f` only if all the additional arguments have default values.

2. For every constant declaration in `S`, there is a constant declaration of the same name and conforming type in `C`.

As the base case of this recursive definition, every type conforms to itself.

The conformance check for testing the conformance of one signature to another is exactly the same, substituting a signature `T` for class `C`.

In order to conform to C++'s rules for lexical scoping, type definitions in `S`, such as local classes, unions, or enumerations, are ignored in the conformance check. One use of local types is to aid in providing *default implementations* of signature member functions, which are discussed below. A local type `t` can also be referred to outside the signature using the syntax `S::t`. For example, if a local type is used as argument type or return type in signature member function declarations, classes need to refer to the type as `S::t` in their member function declarations in order to conform to the signature. A `typedef` only defines an *alias* for a type. It is, therefore, not necessary for the class to refer to it by name, the type it aliases can be used instead.

Field declarations as well as private or protected member functions and constructors in `C` are ignored during conformance checking. Also, `C` can have more public member functions or types than those specified in `S`.

For example, suppose we are testing the conformance of class `C` to signature `S`. Given signatures `T` and `U` and classes `D` and `E`, let signature `U` conform to signature `T`, let class `D` conform to signature `T`, and let class `E` be a subclass of class `D`. The signature member function

```
T * S::f (D *, E *);
```

can be matched with any of the following class member functions:

```
T * C::f (D *, E *);            // since the types are the same
T * C::f (D *, D *);            // since 'D' is a base type of 'E'
T * C::f (T *, E *);            // since 'D' conforms to 'T'
T * C::f (T *, T *);            // since both 'D' and 'E' conform to 'T'
D * C::f (D *, E *);            // since 'D' conforms to 'T'
E * C::f (D *, E *);            // since 'E' conforms to 'T'
U * C::f (D *, E *);            // since 'U' is a subtype of 'T'
T * C::f (D *, E * = NULL);     // since the default value is ignored
T * C::f (D *, E *, int = 0);   // since the 3rd argument has a default value
T * C::f (D *, E *) throw (X);  // since S::f allows any exception
```

Note that conformance is defined using contravariance [6] of the argument types of member functions and covariance of the result types. This makes subtyping based on signatures more flexible than the subtype relationship defined by class inheritance. Unlike elsewhere in C++, exception specifications are considered part of the type of member functions. This allows catching the violation of an exception specifications at compile time instead of aborting the running program.

If several member functions of `C` conform to one member function of `S`, we find the one that conforms best using a variant of C++'s algorithm for finding the function declaration that best matches the call of an overloaded function [12]. To apply C++'s overload resolution algorithm, the signature member function is treated as a class member function in a function call. In addition, the overload resolution algorithm has to be extended to consider the cost of converting an object pointer to a signature pointer to be higher than the cost of converting an object pointer to an object pointer of a base class.

If a member function of C conforms to several member functions of S, an error must be reported by the compiler. Otherwise, the subtype relationship induced by the conformance check would be semantically ill-defined.

These rules for handling overloading of signature member functions could be relaxed by considering different matches of C's member functions with S's member functions and by picking the best match according to some metric on signature types. I.e., instead of finding the best matching class member function for a single signature member function, the overload resolution algorithm could be extended to work with multiple signature member functions in parallel. However, we feel that any such algorithm would be sufficiently complex to confuse users.

3.4 Default Implementations

Since signature declarations declare interface types, they usually only contain member function and operator *declarations*. However, a signature declaration can also contain member function *definitions* (i.e., declarations together with implementations). Such definitions are called *default implementations*. Consider, for example, the signature

```
signature S {
    int f  (int);
    int f0 ()     { return f (0); };
};
```

For a class C to conform to S, it is not necessary for C to contain the member function '`int f0 ()`.' However, if `C::f0` is defined and of the right type, it will be used. If `C::f0` is not defined the default implementation `S::f0` is used instead.

Default implementations are useful for rapid prototyping during interface design since they allow quick implementations of functions and classes which can later be replaced by more efficient or sophisticated implementations. For example, a design could define an `integer` signature with addition and multiplication member functions, and implement it with a class which only supports addition. Multiplication could be implemented in the signature by a default member function which does repeated additions. In the later stages of the design, a class with a member function that does multiplication directly can be added without changing any other code.

One consequence of allowing default implementations is that they introduce a case that cannot be type checked fully at compile time. The problem arises when assigning a signature pointer of signature type T to a signature pointer of signature type S, where T contains a default implementation for a member function `f` but S only contains a *declaration* of `f`. Since it is not known at compile time whether the default implementation of `T::f` is actually used, a run-time test for it must be generated. Consider

```
signature S {
    int f ();
};

signature T {
    int f () { return 0; };
};

int foo (T * p)
{
    S * q = p;

    /* ... */
}
```

In the function `foo` above it cannot be known whether `p` will use `T`'s default implementation or not. If the default implementation is used, there will be a run-time type error in the assignment to `q`. Since using `T`'s default implementation when calling `q->f()` would violate the static scoping rules of the language, this is not an option.

Note that this is the only case where a run-time type check is necessary, in all other cases conformance can be fully checked at compile time. The compiler should warn of the possibility of a run-time type error by printing a warning message when generating the run-time test. In addition, it might be desirable for the compiler to provide a command-line flag for turning the run-time test into a compile-time error.

3.5 Constants

As mentioned in the definition of the conformance check, a signature can contain constant declarations. Unlike constant declarations elsewhere, constants in signatures need not be initialized. Instead, they are treated like nullary functions. For example, a class conforming to

```
signature S {
    const int n;
};
```

has to have a public declaration of constant `n`. The value of the class's constant can then be accessed through a signature pointer as in the following example.

```
class C {
public:
    const int n = 17;
};

S * p = new C;
int i = p->n;
```

The variable `i` above gets the value 17. The behavior is the same as if the constant `n` had been replaced by a nullary function returning the constant value, except that it can be implemented more efficiently.

It is possible to implement *initialized* constants in signatures, and treat them like constant nullary functions with a default implementation, i.e., the value of the class's constant overrides the value of the signature's constant. However, since we also want to use constants for defining data structures, we require that the value of a constant in both the class and the signature is the same. Otherwise, it would be impossible to write code such as

```
signature S {
    const int n = 17;
    typedef int[n] array;
    int f (array);
};
```

since the value of `n` would not be known at compile time.

4 Example Uses of Signatures

4.1 Signatures to Separate Type and Class Hierarchies

The solution to model the type and implementation hierarchies in the computer algebra example is to use signatures instead of abstract virtual classes for the type hierarchy:

```
signature GeneralMatrix          { /* ... */ };
signature NegativeDefiniteMatrix { /* ... */ };
signature OrthogonalMatrix       { /* ... */ };
```

Since `NegativeDefiniteMatrix` and `OrthogonalMatrix` conform to `GeneralMatrix` they are also subtypes of `GeneralMatrix`. By using inheritance of signatures, as defined in [2], we can simplify the definition of the latter two signatures.

For modeling the implementation hierarchy we use classes and class inheritance:

```
class DenseMatrix  { /* ... */ };
class SparseMatrix { /* ... */ };
class PermutationMatrix : private SparseMatrix { /* ... */ };
```

Signature conformance ensures that we can use these classes as implementations of the above signature types. Note that we use private inheritance for defining `PermutationMatrix`. This allows us to hide any member functions defined in `NegativeDefiniteMatrix` but not in the other two signatures.

4.2 Signatures for Retroactive Type Abstraction

The solution to the X-Window object example using signatures is actually quite simple. All that is needed is to introduce a signature to define the abstract type `XWindowObject`,

```
signature XWindowObject {
    void display ();
    void move    ();
};
```

and to implement the display list as a list of pointers to `XWindowObjects`,

```
XWindowObject * displayList[NELEMENTS];
```

Given a pair of implementation hierarchies such as:

```
class OpenLookObject {
public:
    virtual void display ();
    virtual void move    ();
    // ...
};
```

and

```
class MotifObject {
public:
    virtual void display ();
    virtual void move    ();
    // ...
};
```

It is simple to use the display list. For example,

```
int main ()
{
    displayList[0] = new OpenLookCircle;
    displayList[1] = new MotifSquare;
    // ...
```

```
    displayList[0]->display ();          // invokes OpenLookCircle::display
    displayList[1]->display ();          // invokes MotifSquare::display

    return 0;
}
```

where `OpenLookCircle` is a subclass of `OpenLookObject` and `MotifSquare` is a subclass of `MotifObject`.

If we have only one implementation provided in compiled form and we would like to abstract the type of some of its classes and add an alternative implementation, the solution is similar as above. The types of classes are abstracted by defining signatures, an alternative implementation then consists of classes conforming to those signatures.

4.3 Signatures to Implement Conflicting Type and Class Hierarchies

The solution to the `Queue/DEQueue` problem presented earlier is also quite easy using signatures. Simply define an implementation class, and two signatures to define the abstract types `Queue` and `DEQueue`.

```
template <class T> class DoublyLinkedList {
public:
    void enqueueHead (T);
    T    dequeueHead ();
    void enqueueTail (T);
    T    dequeueTail ();
    // ...
};

template <class T> signature DEQueue {
    void enqueueHead (T);
    T    dequeueHead ();
    void enqueueTail (T);
    T    dequeueTail ();
};

template <class T> signature Queue {
    void enqueueTail (T);
    T    dequeueHead ();
};

Queue<int> *      q1 = new DoublyLinkedList<int>;
DEQueue<char *> * q2 = new DoublyLinkedList<char *>;
```

It should be noted that this same effect can be achieved in C++ without signatures by using multiple inheritance. E.g., by implementing `Queue` and `DEQueue` as abstract classes and having `DoublyLinkedList` inherit from both. To see where this type of solution breaks down, consider adding another type, `Stack`, with member functions `push` and `pop`. With signatures it is simple to define a `Stack` signature and whenever assigning a `DoublyLinkedList` use a *view* [2] to rename `enqueueHead` to `push` and `dequeueHead` to `pop`. With the multiple inheritance based solution, it would be necessary either to introduce a new multiply inherited abstract class that *implements* `push` and `pop` by delegating to `enqueueHead` and `dequeueHead`, or to alter `DoublyLinkedList` to implement `push` and `pop` directly. The former unnecessarily constrains the implementation of other classes that might implement an abstract stack type, while the latter needlessly clutters the implementation of `DoublyLinkedList`.

5 Implementation Techniques

We detail three options for implementing signatures. The first method could be used in a compiler preprocessor (e.g., a `cfrontfront`) that translates C++ with signatures into C++ without signatures. The second is a compiler based implementation that produces a C-level intermediate code version of signatures and needs direct access to the type checking phases of a C++ compiler, but is independent of the compiler back-end and machine architecture. This method has been implemented in the GNU C++ compiler [21] as a modification of GCC's C++ front end, `cc1plus`. The same techniques are equally applicable to AT&T's `cfront`, or other C++ compilers. Finally, we outline an implementation technique that requires support from the compiler back-end and code generation phases to generate assembly-level code to further optimize signature member function calls.

5.1 Preprocessor-Based Implementation

The central idea of this implementation technique is to generate interface objects that encapsulate the class objects. These interface objects forward the signature member functions to the appropriate class member functions. Signature pointers are then implemented as regular C++ pointers that point to these interface objects.

Consider the declarations

```
signature S {
    int f ();
    int g (int, int);
};

C obj;
S * p = &obj;
```

and assume `C` conforms to `S`. The signature declaration itself is simply a type declaration; no code needs to be generated. The code for the interface object is generated when compiling the assignment to the signature pointer `p`.

In the particular case above, the interface object must redirect the signature member functions `S::f` and `S::g` to the corresponding class member functions `C::f` and `C::g`.

To create such interface objects for any class `C` that conforms to a signature `S`, we first generate an abstract virtual class `S_Interface`. For each class `C`, we then need a subclass of `S_Interface` that redirects the signature member functions to the class member functions of the given class.

For the signature `S` given above, we generate the following abstract virtual class:

```
class S_Interface {
public:
    virtual ~S_Interface ()    = 0;
    virtual operator void * () = 0;
    virtual int f ()           = 0;
    virtual int g (int, int)   = 0;
};
```

The virtual destructor is used to allow deletion of a class object through a signature pointer. The conversion operator is used for implicitly converting a signature pointer to a pointer of type `void*`. For creating the classes of interface objects, we generate a template class `S_C_Interface` as public subclass of `S_Interface`.

```
template <class C> class S_C_Interface : public S_Interface {
    C * optr;
public:
```

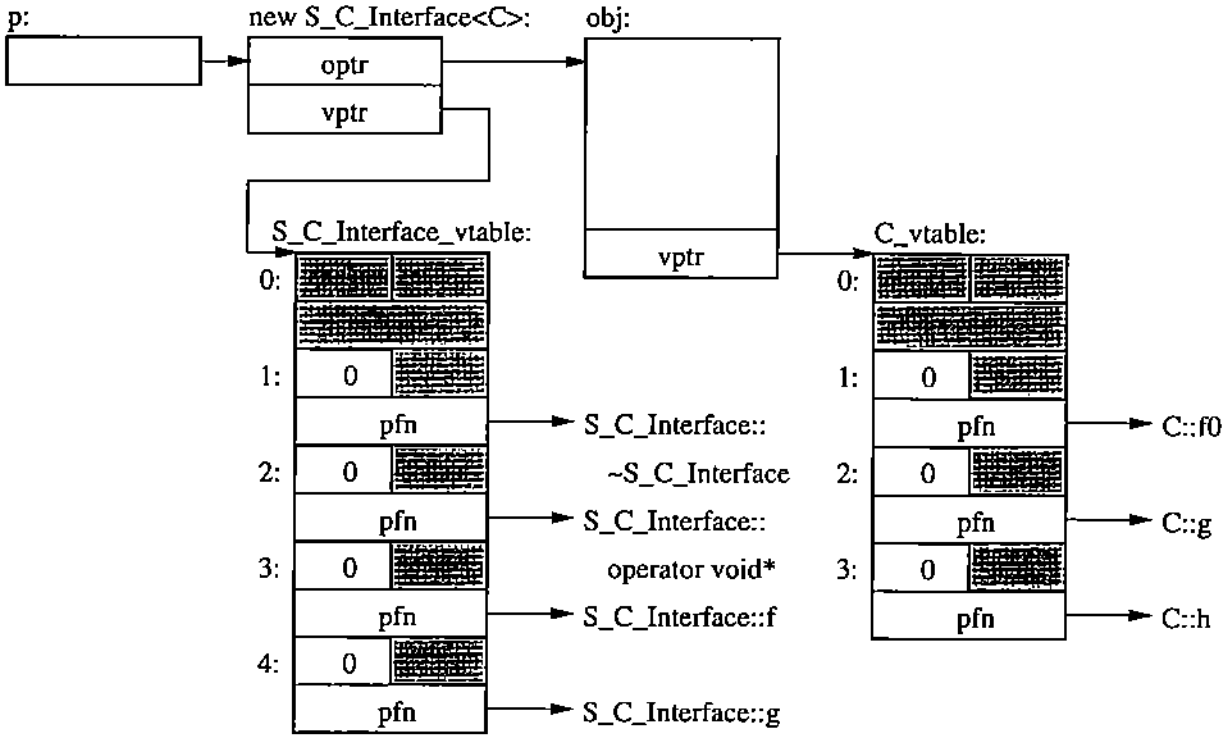

Figure 1: Preprocessor-based implementation.

```
    S_C_Interface (C * q) { optr = q; };
    ~S_C_Interface ()     { delete optr; };
    operator void * ()    { return (void *) optr; };
    int f ()              { return optr->f (); };
    int g (int x, int y)  { return optr->g (x, y); };
};
```

This template class is then instantiated with some class `C` to build the class of objects interfacing `S` and `C`.

Signature pointers can now be implemented as pointers to objects of type `S_C_Interface<C>` for a given class `C`. That is, the declaration

```
S * p = &obj;
```

is translated to

```
S_Interface * p = new S_C_Interface<C> (&obj);
```

The resulting data structure is displayed in Figure 1 (unused data members in the virtual function tables are shaded).

If there is another signature pointer `q` of type `S*` on the RHS of the assignment, the preprocessor simply generates an assignment of the resulting pointers of type `S_Interface*`. If `q` is a signature pointer of type `T*`, we pass `q` as argument to the constructor of `S_C_Interface<T_Interface>`. This has the effect that the data member `optr` of the LHS interface object will point to the RHS interface object.

Since a signature pointer is a standard C++ pointer in this scheme, we do not need to do anything special to compile a signature member function call. The call `p->f()` simply invokes `S_C_Interface<C>::f`, which in turn calls `C::f`. Similarly, the statement '`delete p`' results in a call of the destructor, which, in turn,

deletes the class object. To convert a signature pointer into a pointer of type `void*`, the (implicit) cast expression `(void *)p` needs to be translated into `(void *)*p`, which results in the conversion operator call '`(*p).operator void*()`.'

To compile signature constants without initialization, the constant must be translated into a variable in the interface class. Assume signature `S` contains the constant declaration '`const int c;`.' We translate this declaration into the public member declaration '`int c;`' in class `S_Interface`, and initialize `c` in the constructor of the template class `S_C_Interface`:

```
S_C_Interface (C * q, int i) { optr = q;  c = i; }
```

For initializing a signature pointer, or for assigning to one, the value of the class constant has to be provided as the second argument of the constructor:

```
S_Interface * p = new S_C_Interface<C> (&obj, C::c);
```

To implement default implementations, we have to add a flag to the interface object that indicates whether a given member function is provided by the class or not. Assume that the signature member function `f` comes with a default implementation. We add the flag '`unsigned int f_flag:1;`' as a public data member to class `S_Interface` and generate the following code for the member function `f` in class `S_C_Interface`:

```
int f ()
{
    if (f_flag)
        return optr->f ();

    // code for default implementation
}
```

Similarly as with signature constants, the flag has to be initialized in `S_C_Interface`'s constructor.

When assigning a signature pointer to another signature pointer of a different type, we have to generate run-time tests to make sure that no default implementation of the RHS signature could be used through the LHS signature pointer in violation of the static scoping rules. For example, assume that signature `T` is identical to signature `S`, except that `T::f` has a default implementation while `S::f` does not. The declarations

```
T * q = new C;
S * r = q;
```

are translated into

```
T_Interface * q = new T_C_Interface<C> (new C);
S_Interface * r;
if (q->f_flag) {
    cerr << "'r' cannot be initialized with signature pointer" << endl
         << "using default implementation 'T::f'" << endl;
    abort ();
} else
    r = new S_C_Interface<T_Interface> (q);
```

In this example, the optimizer can remove the test and the error message by constant folding. In case `C::f` exists, the default implementation `T::f` would not be used, and the flag `q->f_flag` would, therefore, be zero. In general, the compiler does not know the class of the object the RHS signature pointer points to and, hence, has to generate this run-time test.

Translating a signature pointer into a pointer to an interface object has the advantage that it is straightforward to implement in a preprocessor for a C++ compiler. It requires, however, to allocate interface objects

on the heap. Another disadvantage is that assignments with a signature pointer on the RHS can result in the LHS signature pointer accessing the class object through a chain of interface objects.

To avoid heap allocation, we can use the interface object itself as a signature pointer. In this case, the declaration of p is translated to

```
S_C_Interface<C> p = &obj;
```

This solution requires some more intelligence in the preprocessor to make p behave as if it were a pointer of type `S_Interface*`. For example, the signature member function call `p->f()` now needs to be translated into `p.f()`. Signature references are implemented exactly the same way as signature pointers.

For signatures that do not have default implementations or constants, the storage needed for an interface object is two words, the pointer to the class object, `optr`, and the pointer to `S_C_Interface<C>`'s virtual function table. Each default implementation requires one additional bit, and constants can be arbitrarily large. Therefore, performing the above optimization for reducing heap allocation should be conditional on the size of the interface object. With signature pointers being the interface objects themselves, assigning one signature pointer to another requires copying the entire `S_C_Interface<C>` structure. If the signature pointer takes only two words of storage, copying is not a problem. With a constant array of several kilobytes in a signature, copying is certainly a bad choice.

5.2 Compiler Front-End Implementation

As the preprocessor-based implementation, the compiler front-end implementation is based on the basic idea of encapsulating class objects with interface objects. However, by translating signatures to the level of abstraction of C code instead of C++ code, we are able to produce more efficient code. Although the description of the compiler front-end implementation relies on details of how the GNU G++ compiler [21] compiles C++ classes, the same ideas can be used in other compilers as well.

In the preprocessor-based implementation, there are two main sources of inefficiency. One is that when calling a signature member function, two member functions calls have to be performed in the generated code, the call to the interface object's member function and the call to the actual class member function. The other problem is that using signature constants can cause interface objects to become very large.

Instead, to optimize signature member function calls, signature pointers and signature references are directly used as interface objects. However, rather than relying on the virtual function call mechanism and specializing the interface object with a template to the class of the object, we introduce a special table, called the *signature table*, that allows us to perform the signature member function call independent of the class of the object. In essence, we inline the call of the member functions of class `S_C_Interface<C>` by storing all the class specific information contained in those member functions in the signature table. A signature table is similar in structure to a virtual function table but contains additional information. A signature table only depends on a signature and conforming class pair and, therefore, can be shared between multiple signature pointers.

The key to optimizing the space requirements of interface objects is to observe that signature constants, as well as the default implementation flags, do not depend on the actual object but only on the class of the object. The values of both signature constants and default implementation flags can be determined in the conformance check. Since the values are class specific, the obvious place to store them is in the signature table.

This optimization of signature member function calls is only possible in this implementation if the class of the object is *strictly conforming* to the signature. Strict conformance means that a signature member function and the corresponding class member function need to have the same number of arguments, exactly the same argument types, and exactly the same return type. In the general case, we might need to convert argument types or the return type in a signature member function call, but we do not have place in a signature table to store the conversion code. If conversion of arguments or the return value is necessary, we need to generate a function to do it. This means that, as in the preprocessor-based implementation, we need two member function calls to perform one signature member function call.

Outline of the Implementation

In order to outline the structure of the compiler front end implementation, we initially ignore default implementations, signature constants, classes with virtual member functions, and multiple and virtual inheritance of classes. Also, we restrict conformance to strict conformance.

For the signature declaration

```
signature S {
    int f ();
    int g (int, int);
};
```

the compiler generates an internal representation of the following structure of function pointers:

```
struct S_Table {
    const void * _.dtor;
    const int    (* _f) (void *);
    const int    (* _g) (void *, int, int);
};
```

where the data member `_.dtor` represents the destructor that is implicitly declared in every signature. The first argument of type `void*` of the function pointers is used to pass the object pointer `this` to a member function. The type `S_Table` will be the type of signature tables for signature `S`.

In the preprocessor implementation, an interface object contains a pointer to the class object and a pointer to a virtual function table. In this scheme, we have a pointer to the signature table instead of the virtual function table pointer. Since we store the interface object directly in the signature pointer, this leads to the following type declaration for signature pointers:

```
struct S_Pointer {
    void *          optr;
    const S_Table * sptr;
};
```

Signature references use the same representation. Conceptually, the type of `optr` should be *pointer to any object* instead of pointer to nothing. Since neither C nor C++ allow us to express this, the compiler must generate appropriate casts when using `optr`.

Code generated for the declaration '`S * p = new C;`' looks as follows:

```
static const S_Table S_C_Table = { &C::~C, &C::f, &C::g };
S_Pointer p = { new C, &S_C_Table };
```

To initialize the signature table `S_C_Table`, the compiler needs to cast the destructor and member functions of class C to the appropriate function pointer types. If `C` does not have a destructor, the default destructor is used. Since C++ does not allow taking the address of a destructor, this must done in the compiler front end.

While we can use a default constructor for initializing a signature pointer as shown above, we need to translate an assignment to a signature pointer into a compound expression. For the assignment expression '`p = new C`,' or for passing an object to a signature pointer parameter in a function call, the compiler generates the compound expression

```
( p.optr = new C,
  p.sptr = &S_C_Table,
  p
)
```

as well as the declaration and initialization of the signature table:

```
static const S_Table S_C_Table = { &C::~C, &C::f, &C::g };
```

If the assignment is in an inner scope, the signature table declaration needs to be moved out of this scope into file scope.

Since signature tables are static and constant, only one signature table declaration per signature-class pair needs to be generated in each file.

To compile a function call such as

```
int i = p->g (7, 11);
```

we need to dereference p's `sptr` and call the function whose address is stored in the data member `_g`, which is `C::g` in our example. We need to pass the value of p's `optr` as first argument, so that `C::g` gets a pointer to the right object passed for its implicit first parameter called `this`.

```
int i = p.sptr->_g (p.optr, 7, 11);
```

If the compiler knows the current value of `p->sptr`, this can be optimized to a direct call to `C::g`.

Signature Tables

If classes with virtual member functions or classes that are defined using multiple and/or virtual inheritance are used as implementations of signature types, we need additional information in the signature table to perform a signature member function call correctly. Also, default implementations and signature constants need to be represented in the signature table.

When a signature member function is implemented by a virtual class member function, since we do not know the actual type of the object pointed to by the signature pointer, we do not know the address of the function to call until run time. Instead, we must look up the address of the function in the appropriate virtual function table. To facilitate casting objects up and down the class hierarchy, implementations of C++ typically do not use a single virtual function table per class but one virtual function table for each base class that contains virtual functions. To allow finding the appropriate virtual function table in a member function call, an object contains possibly multiple pointers to virtual function tables. For a given virtual function, we therefore need to store in the signature table the index into the virtual function table and the offset in the object at which to find the pointer to the proper virtual function table.

In GCC, member functions are implemented as regular functions that take a pointer to the object, called `this`, as first argument. If a member function was inherited from a base class and multiple inheritance was used, the `this` pointer might need to be adjusted to point to the beginning of the sub-object of the correct type. In order to adjust the `this` pointer correctly for a given class member function, we need to store the offset that has to be added to `this` in the signature table.

To make matters worse, in the case of virtual inheritance we might not even know the layout of an object at compile time. Virtual inheritance is used to prevent duplication of members that are accessible through multiple paths in the inheritance hierarchy. If a member function was inherited through virtual inheritance, we need to follow an additional indirection for adjusting the `this` pointer and to find the appropriate virtual function table pointer. To allow this indirection, we must store in the signature table the offset into the object at which we find the pointer to a virtual base object.

Last but not least, we need three flags in a signature table entry to determine whether a non-virtual member function, a virtual member function, or default implementation has to be called and whether or not virtual inheritance was used.

To summarize, a signature table entry has the following structure:

```
struct sigtable_entry_type {
    short tag;          // non-virtual, virtual, or default implementation?
    short vb_off;       // offset to virtual base pointer
    short delta;        // 'this' adjustment
    short index;        // vtable index
```

```
    union {
        void * pfn;     // pointer to function
        short  vt_off;  // offset to vtable pointer
    };
};
```

The data member `tag` contains two flags to distinguish between non-virtual and virtual member functions and default implementations. If a member function was inherited from a virtual base class, the data member `vb_off` contains the offset at which the virtual base pointer is found. If no virtual inheritance was used, `vb_off` is negative, i.e., the third flag mentioned above is the sign bit of `vb_off`.

The data member `delta` contains the value to be added to `this`, `pfn` contains a function pointer in case of a non-virtual member function or a default implementation, and, in case of a virtual member function, `vt_off` and `index` contain the offset of the virtual function table pointer in the object and the index for the virtual function table, respectively. The data member `vt_off` occupies the same memory location as `pfn`. For type checking purposes, the compiler needs to cast `pfn` to the appropriate function pointer type.

Conceptually, a signature table entry is a member function pointer. The only difference is that while a regular member function pointer can only point to a class member function, a signature table entry can point to a default implementation as well. We expect, therefore, some similarity in the data structures. Indeed, the data members `delta`, `index`, `pfn`, and `vt_off` are the same as used in the data structure of member function pointers and virtual function table entries. An alternate declaration for signature table entries would, therefore, be:

```
struct sigtable_entry_type : public vtable_entry_type {
    short tag;
    short vb_off;
};
```

In `vtable_entry_type`, the name `delta2` is used instead of `vt_off`. In a member function pointer, the sign bit of `index` is used to distinguish between a virtual and a non-virtual member function. To store the bit to distinguish between a class member function or a default implementation, we need the data member `tag`. The lack of the data member `vb_off` in member function pointers can cause member functions from virtual base classes to be called incorrectly.

If a member function was inherited through two or more occurrences of virtual inheritance, even the one data member `vb_off` in a signature table entry is insufficient. In the general case, we might have to follow multiple virtual base pointers to find the right base object. This would require multiple `vb_off` data members. Since the number of `vb_off` data members would depend on the class hierarchy, we could not statically determine the size of a signature table entry. A better solution would be to change the object format by introducing additional virtual base pointers so that any virtual base could be found with only one indirection.

When calling a member function through a member function pointer, the G++ compiler determines the layout of the object based on the class name used in the member function pointer declaration. In most cases, this strategy works correctly. However, if an object of a subclass is used, G++ has no way of knowing the actual layout of the object. In this case, the member function call might produce unpredictable results. Since the class of the object pointed to by a signature pointer/reference is not known at compile time, we cannot use this approach of assuming an object layout that would work in most cases. We always need the data member `vb_off` in a signature table entry. To correctly call a member function through a member function pointer in all cases, it also would be necessary to add a `vb_off` data member to `vtable_entry_type` and to include additional virtual base pointers in the object.

The signature table is now a structure that contains a data member of type `sigtable_entry_type` for every member function declared in the signature and one for the implicitly declared destructor. For signature `S` declared earlier, the signature table looks as follows:

```
struct S_Table {
```

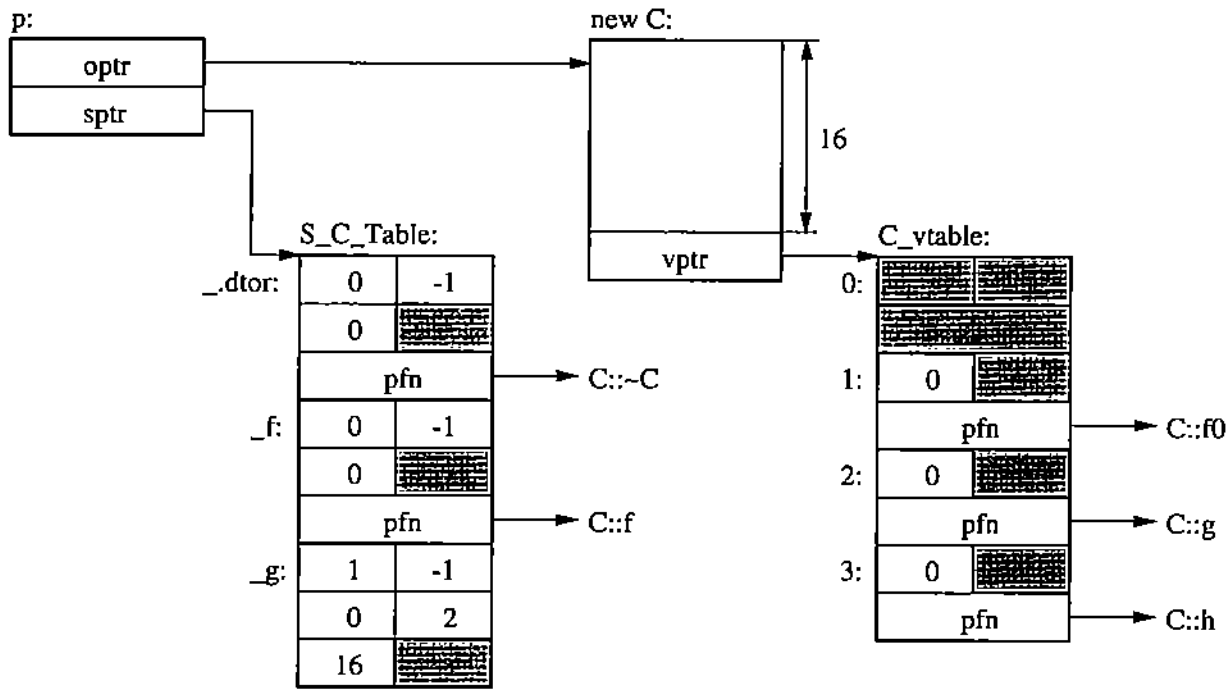

Figure 2: Compiler front-end implementation.

```
    sigtable_entry_type _.dtor;
    sigtable_entry_type _f;
    sigtable_entry_type _g;
};
```

We will see later why the data members of `S_Table` cannot be constant. In addition, for each uninitialized constant in the signature, we insert a data member declaration into the signature table type. All the information for initializing the data members of a signature table entry and for initializing constants can be obtained at compile time from the class of the object on the RHS of a signature pointer assignment or initialization.

Given a signature `S` with member functions `f` and `g` and a conforming class `C`, the assignment of an object of class `C` to a siganture pointer `p` results in the data structure displayed in Figure 2.

Signature Member Function Call

To call a signature member function, we need to generate a conditional expression that tests the data member `tag` of the signature table entry and, depending on its value, call a non-virtual member function, a virtual member function, or a default implementation. We also have to make sure that the right offset gets added to the `this` pointer. The signature member function call

```
int i = p->g (7, 11);
```

from our example above is now translated into

```
int i = (s    = p.sptr->_g,
         base = (s.vb_off < 0) ? p.optr : *(p.optr + s.vb_off),
         this = base + s.delta,
         (s.tag == 0)
```

```
        ? // non-virtual member function call
          s.pfn (this, 7, 11)
        : // virtual member function call
          ((**(base + s.vt_off))[s.index]).pfn (this, 7, 11)
);
```

where `s`, `this`, and `base` are compiler generated temporary variables. The structure `s` contains the signature table entry. If virtual inheritance is used, `base` points to the part of the object corresponding to the virtual base class. Otherwise, `base` points to the beginning of the object. The pointer `this` is offset from `base` to point to the part of the object corresponding to the base class from which the member function `g` was inherited. These temporary variables can be kept in registers.

The above code assumes that a virtual function table entry contains a data member `pfn` that contains the pointer to the function. The `delta` stored in the virtual function table entry is not needed since it is the same as the `delta` stored in the signature table entry. If the virtual function table contains pointers to pieces of code that adjust the `this` pointer and then branch to the function, which is an alternative implementation of virtual function tables, the virtual member function call in our example becomes

```
((**(base + s.vt_off))[s.index]) (p.optr, 7, 11)
```

In this case, we do not add `s.delta` to the implicit first argument, as it will be added in the code piece pointed to from the virtual function table.

In case the signature member function `g` has a default implementation, we need to add a third branch to the conditional expression. The signature member function call now becomes

```
int i = (s    = p.sptr->_g,
         base = (s.vb_off < 0) ? p.optr : *(p.optr + s.vb_off),
         this = base + s.delta,
         (s.tag == 0)
             ? // non-virtual member function call
               s.pfn (this, 7, 11)
             : (s.tag >= 0)
                   ? // virtual member function call
                     ((**(base + s.vt_off))[s.index]).pfn (this, 7, 11)
                   : // default implementation call
                     s.pfn (p, 7, 11)
);
```

Since in practice a non-virtual function call is expected to be the most common case, it should be reached with only one test.

If instead of the signature pointer variable `p` in our example, we have an expression that evaluates to a signature pointer, the result needs to be stored in a temporary signature pointer variable first to prevent the expression from being evaluated multiple times.

If a signature member function is called while constructing or destructing the object the signature pointer/reference points to, the behavior is undefined. In particular, calling a virtual member function through a signature pointer before the virtual function table pointer in the object is initialized is likely to result in a crash. However, this is nothing new. If a class pointer is used instead of a signature pointer, the behavior is the same. The only way for the compiler to detect such aliasing is through global data flow analysis.

Full Conformance

So far we have only considered strict conformance. If we lift this restriction, it might be necessary to convert arguments and/or the return value when calling a signature member function. In the following we discuss how conversion functions look like and how they are installed in a signature table.

Assume T is a signature and D a class conforming to T. Consider the declarations

```
signature S {
    int f (D *);
    T * g (int);
};

class C {
public:
    int f (T *);
    D * g (int, int = 0);
};
```

The member function C::f conforms to S::f since the type of C::f's argument, T*, is a base type of S::f's argument type D*. Similarly, C::g conforms to S::g since its return type D* is a subtype of (i.e., conforms to) S::g's return type T* and since its second argument has a default value. Therefore, C conforms to S. Since this is not strict conformance, conversion functions are needed for both member functions.

The conversion functions are generated together with the signature table S_C_Table, i.e., when testing conformance of S and C for compiling an assignment statement or declaration of the form

```
S * p = new C;
```

Like the signature table, the conversion functions have static linkage. They have the same type as the signature member functions S::f and S::g, for which they are generated. Since the conversion functions are not in signature scope but in file scope, we need to explicitly provide them with the first argument this of type S*. For our example, the compiler would need to generate the conversion functions

```
static int S_C_f (S * this, D * arg1)
{
    return ((C *) this)->f ((T *) arg1);
}

static T * S_C_g (S * this, int arg1)
{
    return (T *) ((C *) this)->g (arg1);
}
```

Since they have the same types as the original signature member functions, we can treat them like default implementations, flag them as such, and install pointers to these conversion functions in the signature table entries S_C_Table._f and S_C_Table._g, respectively.

The signature member function call p->f() now results in the conversion function S_C_f being executed. Since the entry _f in the signature table is flagged as a default implementation, the conversion function gets the signature pointer p passed for it's first argument this. In the body of the conversion function, this can safely be converted to a pointer of type C* since this conversion function can only ever be called by dispatching through the S_C_Table, and this table is only used when p.optr points to an object of type C.

Since, like signature tables, conversion functions are declared static, they may be duplicated in other translation units.

Signature-Signature Tables

When compiling a signature pointer assignment/initialization with another signature pointer on the RHS, we do not always have enough information to compute the contents of the LHS signature table. Since it is not known at compile time which signature table the RHS signature pointer points to, we might have to initialize the LHS signature table at run time. An alternative would be to store the information to call a

RHS signature member function in a LHS signature table entry. However, this would result in an additional table lookup when calling a LHS signature member function. The number of table lookups needed to call a signature member function would depend on the number of assignment statements executed and could, therefore, be arbitrarily high.

If the heap is garbage-collected, the most efficient solution is to allocate dynamically initialized signature tables on the heap. Signature tables that result from an object pointer on the RHS are still initialized statically. When assigning a signature pointer to another signature pointer of the same type, we simply copy the two data members `optr` and `sptr`. If the types are not the same but the signature table entries needed in the LHS signature table are found in the correct order as a contiguous block of data in the RHS signature table, we can share the RHS signature table and let the LHS `sptr` point into the RHS table. If the RHS table cannot be shared, the LHS signature table is allocated on the heap and initialized from the appropriate RHS signature table entries.

If no garbage collector is available, we have to resort to allocating signature tables on the stack. To do so the compiler reserves a signature table variable for every signature pointer (or signature reference). A signature table can now only be shared if both LHS and RHS signature pointers are in the same scope or if the RHS signature pointer is in an outer scope. If the LHS signature pointer is in an outer scope, the RHS signature table has to be copied into the table associated with the LHS signature pointer. Similarly, if a local signature pointer is returned as a function value, the signature table has to be copied into the signature table variable associated with the function return value. These copy rules assure that a signature pointer always points to a table in static memory, in the same activation record, or in an activation record higher up on the stack. If a signature table variable associated with a signature pointer was never assigned to, it can be removed during optimization.

Using data flow analysis, it is often possible to determine that the RHS signature table has been statically initialized or that it is in an outer scope. In either case, copying the signature table is unnecessary for assigning to a LHS signature pointer in an outer scope. Another solution to avoid copying would be to test at run time whether a signature table is in static memory or in an outer stack frame. An efficient but architecture-specific implementation of this test would be a comparison of the address of the table with the current stack pointer. A portable solution for testing if a signature table is in static memory would be to include an additional bit in the data member `tag` of a signature table's destructor entry.

If the RHS signature is derived from the LHS signature using single inheritance, the RHS signature table type is a subtype of the LHS signature table type. In this case, the RHS `sptr` can simply be copied into the LHS signature pointer. To allow sharing of the RHS signature table in case of multiple signature inheritance, it is necessary to duplicate the destructor entries in the signature table. For each base signature, the signature table contains one entry that points to the class's destructor. Now for any RHS signature that is a descendent of the LHS signature in the signature inheritance hierarchy, we can avoid copying of table entries.

We argue that in most cases, copying of signature tables entries, or allocating signature tables on the heap, can be avoided by carefully designing the signature hierarchy. Even if the RHS signature is not a descendent of the LHS signature, if the LHS signature member functions are in the same order at the beginning of the RHS signature, copying is avoided.

To alert the programmer of an inefficient signature pointer assignment, the compiler should print a warning message whenever signature table entries have to be copied. Independent of whether copying table entries is necessary, if the RHS signature contains a default implementation where the LHS signature only has a member function *declaration*, the compiler must generate a run-time test and should print a corresponding warning message.

5.3 Implementation with Back-End Support

In the compiler front-end solution, there is room for optimization in the calling sequence for a signature member function call. In this section, we demonstrate how to address these inefficiencies using support from the compiler back-end. While this solution is not directly portable since it depends on knowledge of the

native calling sequence, it can nevertheless be implemented on any architecture. It is especially efficient on modern RISC processors.

For calling a signature member function in the previous solution, the generated code tests the information stored in a signature table entry to decide on how to call the signature member function. The key idea in this solution is to customize the calling sequence for calling a particular class member function and to store (a pointer to) the code of this calling sequence in the signature table entry. The signature table now contains only pointers to these pieces of code, called *thunks*, instead of flags and offsets. A signature member function call is now translated into branching to the thunk, which then adjusts the `this` pointer and branches to the class member function or performs a virtual function call. Such an implementation was proposed in [14]. The same idea is used in some compilers for implementing a virtual member function call.

Each thunk only contains the code necessary to call *one* specific class member function. It is not necessary to test any flags, we can just branch to the thunk directly. The thunk does the right thing for whichever member function is being called. Signature table entries are now reduced to single function pointers.

For example, given the signature `S` with member functions `f` and `g` as above, the signature table is of type

```
struct S_Table {
    void * _.dtor;
    void * _f;
    void * _g;
};
```

Given a class `C` conforming to `S`, assume that `C::f` is a non-virtual member function and that `C::g` is a virtual member function. Neither function requires any offset to be added to `this`. The thunk needed for calling `C::f` is the following short piece of code:

```
S_C_f_Thunk:
{
    this = this.optr;
    goto C::f;
}
```

Before branching to the thunk, the compiler will have set up the activation record correctly for calling `C::f`. In particular, all the arguments were either pushed onto the stack or are in registers. The value passed for the first argument, `this`, is the signature pointer. Before branching to `C::f`, we need to extract the data member `optr` so that `this` points to the object.

For calling the virtual member function `C::g` we need the thunk

```
S_C_g_Thunk:
{
    this = this.optr;
    goto (**(this + VT_OFF))[INDEX].pfn;
}
```

The values `VT_OFF` and `INDEX` are constants that can be determined at compile time and are hard-coded into the thunk. If virtual function tables are implemented using thunks as well, we do not need to select the data member `pfn`. The resulting data structure using a thunk-based implementation of the virtual function table is displayed in Figure 3.

If `C::g` were inherited from a virtual base class and would require a non-zero offset to be added to `this`, the thunk would be

```
S_C_g_Thunk:
{
    base = *(this.optr + VB_OFF);
```

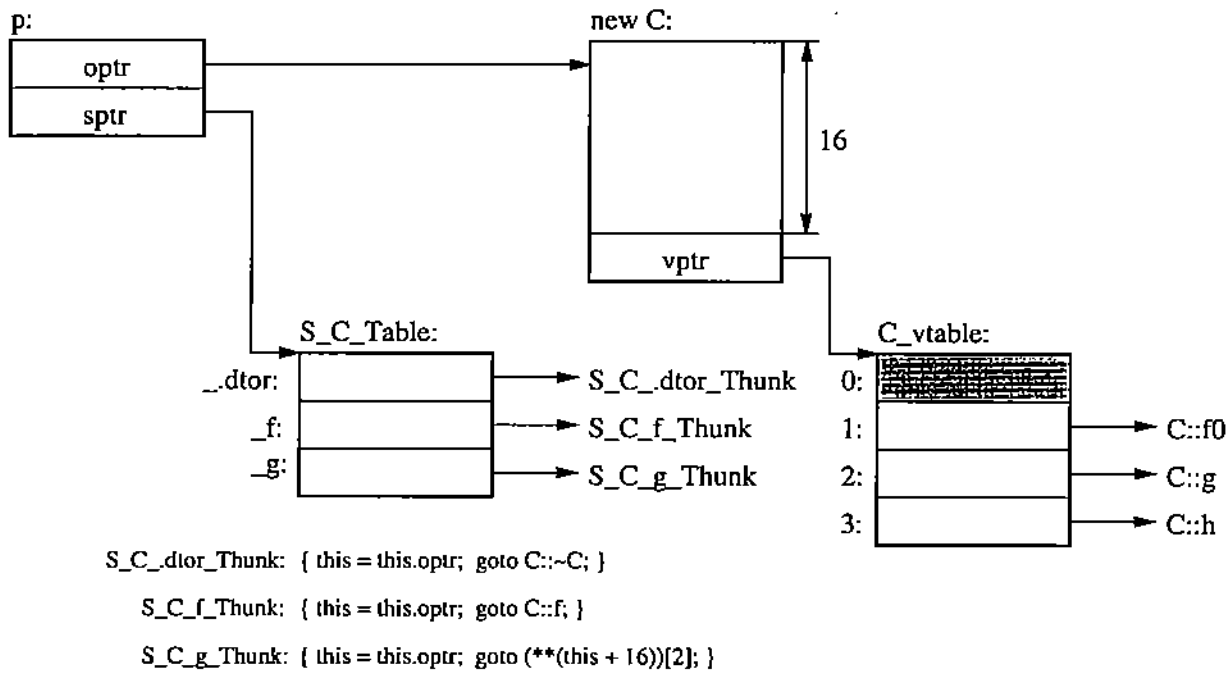

Figure 3: Thunk-based implementations of signature table and virtual function table.

```
    this = base + DELTA;
    goto (**(base + VT_OFF))[INDEX].pfn;
}
```

Also `DELTA` and `VB_OFF` would be constants hard-coded into the thunk.

For calling a default implementation, no work has to be done, since a default implementation expects `this` to be a signature pointer. Therefore, no thunk is needed and the entry in the signature table can point to the code of the default implementation directly.

When compiling an assignment of an object of class `C` to a signature pointer, the compiler generates the above thunks and a declaration of the signature table,

```
static
const S_Table S_C_Table = { &S_C_.dtor_Thunk, &S_C_f_Thunk, &S_C_g_Thunk };
```

and initializes it to point to the thunks. If a default implementation is used, the corresponding signature table entry contains a pointer to the code of the default implementation.

Instead of resulting in a large conditional expression, the signature member function call

```
int i = p->g (7, 11);
```

now reduces to

```
int i = p.sptr->_g (p, 7, 11);
```

Another advantage of using thunks is that code for converting argument types could be included in the thunk. It is not necessary to use a separate conversion function as we did in the front-end solution. The code for converting arguments would simply go before the `goto`. Since a signature table is unique for each signature-class pair, the compiler can generate the conversion code for each thunk when generating the signature table. For converting the return type we could call, instead of branching to, the class member function from the thunk using a light-weight function call sequence. A thunk for the non-virtual member function call would look then as follows:

```
S_C_f_Thunk:
{
    this = this.optr + OFFSET;
    // convert argument types
    temp = ret_addr;
    ret_addr = L;
    goto C::f;
 L: // convert return type
    ret_addr = temp;
    return;
}
```

There is no run-time penalty compared to the front-end implementation if a signature member function does not require conversions. On the contrary, by not having to test the data member `tag` of a signature table entry, by not having to add a zero `delta`, and by not having to test `vb_off`, a few instructions will be saved. The only disadvantage of using thunks is that it requires generation of low-level, machine dependent code, which complicates or even prohibits its use in a compiler that generates C code, such as AT&T's `cfront` compiler.

As in the front-end implementation, assigning a signature pointer to another signature pointer might require copying entries of the RHS signature table to the LHS signature table. In most cases, it is possible to copy the pointer to the thunk. If a member function of the LHS signature does not have the exact same argument and return types as the member function of the RHS signature, however, the compiler needs to generate a new thunk that performs the conversions needed and then branches to the thunk from the RHS signature table, which might do further conversions.

In the thunk implementation described in [14], copying of signature table entries is avoided by having the `optr` of the LHS signature pointer point to the RHS signature pointer instead of pointing to the object. This makes assignment more efficient but requires multiple indirections in a signature member function call. Furthermore, to allow assigning a local signature pointer to a non-local signature pointer, the solution in [14] has to be corrected and signature pointers have to be heap allocated.

There is one final detail in assigning a signature pointer to another signature pointer. If the RHS signature table contains a default implementation that is not allowed to be copied to the LHS signature table, an error has to be reported at run time. To allow this run-time test, we have to reintroduce a flag that indicates whether a default implementation is used or not. This flag can be stored in the low-order bit of the function/thunk pointer in the signature table. We just have to make sure that class member functions and thunks are aligned on half-word or word boundaries, which is required on most RISC-based architectures anyway. When calling a signature member function that might use a default implementation, this bit must be masked out. If the architecture allows, the mask instruction could be omitted by starting default implementations at odd addresses. The only time this flag needs to be tested is in the code for an assignment when performing the run-time error check. If it can be guaranteed that the code of a default implementation is not duplicated across compilation units (either through linker support or by using pragmas), we do not need this extra flag but can compare the function pointer in the signature table with the address of the default implementation instead.

As a possible optimization of signature member function calls, the signature table can contain the code for thunks directly instead of a pointer to a thunk. If a thunk contains conversion code and does not fit into the allocated space, the signature table would contain a branch instruction to jump to the thunk. This makes signature member function calls more efficient for the most common cases. What would become inefficient, however, is copying signature table entries when assigning one signature pointer to another. To avoid that, this optimization could be controlled by the user, or restricted to the case where the compiler determines that no copying of signature table entries is necessary in the entire source file.

With the right layout of the activation record in registers or on the stack, no work needs to be done for adjusting `this`. For example, on a RISC processor, one register could be reserved for the data member `sptr` of `this` when calling a signature member function. This register would not be used for passing arguments

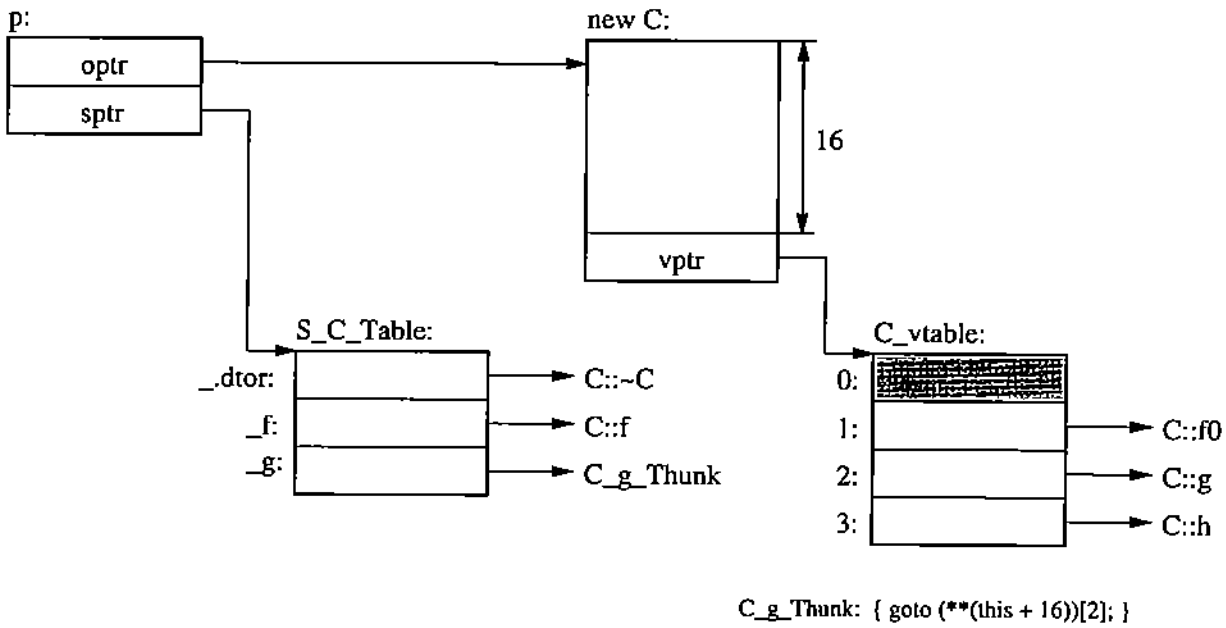

Figure 4: Fully optimized thunk-based implementation.

to class member functions. In this case, we can get rid of the thunk altogether and store a direct pointer to `C::f` in the signature table entry. The data member `optr` of the signature pointer would already be in the correct register to be passed as `this` to the class member function. The register that was reserved for `sptr` can be used for computation in the class member function.

Observe that the information in the thunks is strictly class specific. To avoid the duplication of thunks across multiple compilation units, it would, therefore, be possible to generate the thunks together with the class instead of with signature tables. When compiling a class, the compiler would generate signature thunks for all public member function that are virtual and/or inherited. If no `this` adjustment is necessary and a register is reserved for the `sptr`, it is not necessary to generate a thunk, the address of the member function can be put into the signature table directly (Figure 4).

6 Cost Comparison

In this section, we compare the costs of the three proposed implementation techniques. Since detailed cost analysis in terms of instruction counts and timings would be highly architecture and compiler-specific, we choose instead to analyze space requirements in terms of words of memory and time requirements in terms of logical operation counts.

Ignoring default implementations and constants, the memory required for interface objects in the preprocessor implementation is two words, one for the data member `optr` and one for the virtual function table pointer. This is the same size as the size of signature pointers in the other two implementations, where we have the data member `sptr` instead of the virtual function table pointer. In interface objects, we need additional space for constants and default implementation flags. In the compiler-based implementations, the extra space is placed into the signature tables, which are usually in static memory.

The space needed for the signature table in the compiler front-end implementation is fifty percent more than the space needed for the virtual function table in the preprocessor implementation, three words for each signature member function and an additional three words for the implicitly declared destructor. This is not surprising, since a signature table conceptually is structure containing pointers to member functions and,

as we discussed in the implementation section, a correct implementation of a pointer to a member function would require three words. In the thunk implementation, the signature table takes only one third the space since we only need one pointer per table entry. But in addition we need static storage for the thunks. For the front-end and thunk implementations, signature constants require additional space in the signature table.

When assigning a class object to a signature pointer in the preprocessor implementation, we need to call the constructor of the template class `S_C_Interface`, allocate the interface object on the heap, and then assign two pointers. In addition, if the signature contains uninitialized constants or default implementations, the corresponding data members and flags in the interface object have to be assigned as well. In the compiler-based solutions, it requires only two pointer assignments.

Assigning a signature pointer of a different type than the LHS signature pointer can become expensive in the compiler based implementations if signature table entries have to be copied. Since in the preprocessor implementation the LHS interface object just points to the RHS interface object, the cost is the same as assigning a class object.

In the preprocessor implementation, a signature member function call takes as much time as two class member function calls, a virtual member function call for calling the interface object member function, followed by the class member function call, which may or may not be virtual depending on the class.

In the front-end implementation, dispatching through the signature table to call a non-virtual member function takes roughly the same time as a regular virtual member function call. Calling a virtual member function through a signature pointer requires two table lookups, one to get the signature table entry and another to get the virtual function table entry. In both cases there is the additional constant overhead of dereferencing the `optr` and of testing the data members `vb_off` and `tag` of a signature table entry. If the data member `optr` of the signature pointer is in the wrong register, we also need a register-to-register move. If the signature contains a default implementation, there is an additional test to distinguish between a virtual member function and a default implementation. The cost of calling a default implementation is three tests added to the cost of a virtual member function call.

In the thunks implementation, we do not need to perform any tests when calling a signature member function. Assuming the register layout is such that the data member `optr` of the signature pointer is in the right register to be passed on to the class member function, we can make a signature member function call *exactly* as efficient as a standard virtual member function call in the case of calling a non-virtual member function or a default implementation. When calling a virtual member function through a signature pointer, we have to perform an additional virtual function table lookup.

7 Conclusion

We have discussed the limitations of inheritance for achieving subtype polymorphism and for code reuse. We have proposed language constructs for specifying and working with abstract types that allow us to decouple subtyping from inheritance, have given the syntax and semantics of such an extension, and have proposed three possible implementation strategies for this language extension.

While we have presented the ideas of such a language extension as an extension to C++, they would equally well apply to any statically typed object-oriented programming language.

A signature is a language construct that allows the separation of the concepts of abstract and concrete types. Using structural conformance, we also have separated subtyping from inheritance. Not only are these concepts semantically separated, their implementations are decoupled as well. With the thunk implementation, the mechanism of dynamically dispatching through signature tables is decoupled from any mechanism for implementing concrete types and code reuse (i.e., inheritance in C++). For example, the subtype relationship defined by multiple inheritance is subsumed by structural conformance. In cases where multiple inheritance was used only for subtyping purposes, we no longer need to pay the cost of adjusting the *this* pointer and of following pointers to virtual bases.

With subtype polymorphism defined by signature conformance and implemented through signature pointers and references, it is no longer necessary for inheritance to define a subtype relationship at all. Therefore,

virtual function tables are no longer needed as a dispatch mechanism to achieve polymorphism. Without the need for virtual functions, class inheritance becomes a pure code reuse mechanism. Having decoupled subtyping from inheritance, it is also possible to change the semantics of inheritance and make it conceptually simpler and more versatile for code reuse by allowing to inherit only parts of a base class or by allowing renaming of inherited data members and member functions. While for pragmatic reasons, such changes to C++ are undesirable as they would affect the behavior of existing programs, future programming languages should take advantage of this separation.

8 Availability

Parts of the language extension have been implemented in GCC as a compiler extension. The implementation is included as part of the GCC distribution starting with GCC version 2.6.0.

References

[1] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the OOPSLA '90 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 161–168, Ottawa, Canada, 21–25 October 1990. Association for Computing Machinery. *ACM SIGPLAN Notices*, 25(10), October 1990.

[2] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software Practice & Experience*, 25(8):863–889, Aug. 1995.

[3] Gerald Baumgartner and Ryan D. Stansifer. A proposal to study type systems for computer algebra. RISC-Linz Report 90-87.0, Research Institute for Symbolic Computation, University of Linz, Linz, Austria, March 1990.

[4] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[5] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of the OOPSLA '89 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 457–467, New Orleans, Louisiana, 1–6 October 1989. Association for Computing Machinery. *ACM SIGPLAN Notices*, 24(10), October 1989.

[6] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, New York, New York, 1984. Proceedings of the International Symposium on the Semantics of Data Types, Sophia-Antipolis, France, 27–29 June 1984.

[7] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–43, August 1992.

[8] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Proceedings of the OOPSLA '92 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, Vancouver, Canada, 18–22 October 1992. Association for Computing Machinery. *ACM SIGPLAN Notices*, 27(10), October 1992.

[9] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, California, 17–19 January 1990. Association for Computing Machinery.

[10] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[11] Paul Hudak (ed.), Simon Peyton Jones (ed.), Philip Wadler (ed.), Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, May 1992.

[12] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[14] Elana D. Granston and Vincent F. Russo. Signature-based polymorphism for C++. In *Proceedings of the 1991 USENIX C++ Conference*, pages 65–79, Washington, D.C., 22–25 April 1991. USENIX Association.

[15] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System.* Springer-Verlag, New York, New York, 1992.

[16] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An exemplar based Smalltalk. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 322–330, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[17] David B. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2), October 1985.

[18] David B. MacQueen. An implementation of Standard ML modules. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 212–223, Snowbird, Utah, 25–27 July 1988. Association for Computing Machinery.

[19] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 9–16, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[20] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–45, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[21] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Cambridge, Massachusetts, 14 June 1995. Available as part of the GCC-2.7.0 distribution.

[22] Robert S. Sutor and Richard D. Jenks. The type inference and coercion facilities in the Scratchpad II interpreter. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 56–63, St. Paul, Minnesota, 24–26 June 1987. Association for Computing Machinery. *ACM SIGPLAN Notices*, 22(7), July 1987.

[23] Stephen M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. The Scratchpad II type system: Domains and subdomains. In Alfonso M. Miola, editor, *Computing Tools for Scientific Problem Solving*, pages 63–82. Academic Press, London, Great Britain, 1990.

[24] Niklaus Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin-Heidelberg, Germany, 1985.