Implementing Signatures for C++

GERALD BAUMGARTNER and VINCENT F. RUSSO
Purdue University

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 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. We follow with an analysis of the performance of the three implementation techniques and show that our extension actually allows subtype polymorphism to be implemented more efficiently than with virtual functions. We conclude with a discussion of the lessons we learned for future programming language design.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Design, Languages, Measurement, Performance

Additional Key Words and Phrases: C++, dispatch tables, inheritance, object interfaces, polymorphism, subtyping

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. In an earlier paper [Baumgartner and Russo 1995], we proposed to remedy these problems by introducing a new C++ type definition construct: the *signature*. Signatures provide C++ with a conservative extension to its 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. This article details and analyzes three different implementations of this extension.

This work was supported in part by Purdue Research Foundation grant 690-1398-2278.
Authors' address: Department of Computer Sciences, Purdue University, West Lafayette, IN 47907; email: {gb; russo}@cs.purdue.edu.

The remainder of the article is structured as follows. First we review the 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. The main sections of the article then discuss and compare three different implementation possibilities and presents an analysis and measurements of the performance of each. We conclude with a discussion on the lessons we learned from this experiment and their implications for future programming language design.

2. MOTIVATION

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

(1) 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.
(2) The hierarchies of abstract types and the class hierarchies 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 Retroactive Type Abstraction

A practical example [Granston and Russo 1991] 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 since the root classes of the hierarchies would need to be declared to inherit from the new abstract base class, which would require recompilation. If the member functions needed for the abstract type are nonvirtual 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, and the latter two clutter up the name space with 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 defining the type. The nature of the restriction to binaries in this example prevents us from doing this. What

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

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.

Another, perhaps 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.

2.2 Implementation of Conflicting Type and Class Hierarchies

The abstract type hierarchy and the implementation class hierarchy cannot always be made to agree. An example similar to one in Snyder [1986] 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 Cook et al. [1990], 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 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. 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.

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 article. Specifically, this section details the syntax and semantics of signatures, signature pointers, and signature references, as well as the conformance-checking

algorithm.[2]

3.1 Signature Declarations

A signature declaration is similar to a class declaration except that the keyword `signature` is used instead of `class`, or `struct`, to introduce the 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, member function declarations, operator declarations, conversion operator declarations, and a destructor declaration are allowed within a signature declaration. Specifically

—A signature cannot have constructors, 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 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.

As with a class, an implicit destructor declaration is added if the destructor is not explicitly declared.

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 [Ellis and Stroustrup 1990]. 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.

Signatures allow a type hierarchy to be structured independently from the class hierarchy. This facilitates building complex type hierarchies and the decoupling of

[2]The additional features of signature inheritance, the `sigof` construct (as in Granston and Russo [1991]), and views are left out, since they are straightforward to implement in the typechecking phase of the compiler. The experimental constructs for default implementations, constants in signatures, and opaque types have been removed from our design and implementation, since they do not allow full static typechecking. For information on these constructs, as well as for more details on the semantics of signatures, see Baumgartner and Russo [1995].

subtyping and code reuse. 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 as described earlier. Signatures can, therefore, improve C++'s capabilities for reusing existing code.

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* and assign to it the address of an existing class object. Signature pointers, 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`. That is, 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 can also be assigned to another signature pointer. In this case, the right-hand-side (RHS) signature must conform to the left-hand-side (LHS) 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, which is an unsafe operation.

Like a signature pointer, a *signature reference* is an interface between signature and class types. Because of the semantics of references in C++, signature references can only be initialized to refer to a class object. Any subsequent assignments do not assign to the signature reference but to the class object. Since the compiler does not have any information about the layout of the referenced object, assignments to a signature reference are only valid if the signature contains an assignment operator declaration.

3.3 The Conformance Check

The *conformance check* is the typecheck performed when initializing or assigning to a signature pointer or when initializing a signature reference. The design and implementation of signatures implies no *run-time* cost for the conformance check. The conformance check is performed during the typechecking phase at *compile time*.

To test whether a class or signature type r on the right-hand side of an assignment symbol conforms to the signature type l on the left-hand side, the structures of l and r must be recursively compared. To do this we use an algorithm similar to the recursive subtyping algorithm described in Amadio and Cardelli [1993]. For efficiency, our algorithm caches both positive and negative results of the conformance check. In a compiler, this cache would typically be maintained as part of the symbol table.

`conforms (`l`, `r`) =`

(1) If the pair `(`l`, `r`)` has been marked successful in the cache, then succeed.
(2) If the pair `(`l`, `r`)` has been marked unsuccessful in the cache, then fail.
(3) If l is the same type as r, then succeed.
(4) If r can be converted to l through built-in or user-defined conversions, then succeed.
(5) If l is a base type of r in the class hierarchy, then succeed.
(6) If l is a pointer type t_l`*` and r is t_r`*`, succeed if `conforms(`t_l`, `t_r`)` succeeds.
(7) If l is a reference type t_l`&`, succeed if `conforms(`t_l`, `r`)` succeeds.
(8) If both l and r are function, member function, member operator, or conversion operator types, succeed if
    (a) l and r have the same name,
    (b) r has the same number of parameters as l; or r has more parameters than l, and any additional parameters have default values,
    (c) for every parameter type a_l of l and the corresponding parameter type a_r of r, `conforms(`a_r`, `a_l`)` succeeds,
    (d) `conforms(`r_l`, `r_r`)` succeeds, where r_l is the return type of l and r_r is the return type of r, and
    (e) every exception specifier of l is listed as an exception specifier of r as well.
(9) If l is a union, succeed if there is a union data member of type t_l, such that `conforms(`t_l`, `r`)` succeeds.
(10) If l is a signature, and r is either a class (`class`, `struct`, or `union`) or a signature, then mark `(`l`, `r`)` as successful in the cache and recursively compare the structures of l and r. Succeed if for every member function, member operator, or conversion operator f_l in l there is a corresponding public member function, member operator, or conversion operator f_r in r, such that `conforms(`f_l`, `f_r`)` succeeds. If the recursive comparison of

l and r did not succeed, mark (`l, r`) as failed in the cache and fail.

(11) Otherwise fail.

When initializing an array of signature pointers, the conformance check has to be run for every element in the initialization list. Since arrays are passed to (member) functions by reference, the argument type has to be the same as (or can be converted to) the declared parameter type. Therefore, we do not need any special case for array types in the conformance check; the conformance of array arguments is handled in items (3) and (4).

In order to conform to C++'s rules for lexical scoping, type definitions in a signature `S`, such as local classes, unions, or enumerations, are ignored in the conformance check. A local type `t` can be referred to outside the signature using the syntax `S::t`. If a local type is used as parameter type or return type in signature member function declarations, classes either need to refer to the type as `S::t` in their member function declarations or use a conforming type. Since a `typedef` only defines an *alias* for a type, it is not necessary for the class to refer to it by name; the type it aliases can be used instead.

Data member declarations as well as private or protected member functions and constructors in a class `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 derived from 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 *);            // the types are the same
T * C::f (D *, D *);            // D is a base type of E
T * C::f (T *, E *);            // D conforms to T
T * C::f (T *, T *);            // D and E conform to T
D * C::f (D *, E *);            // D conforms to T
E * C::f (D *, E *);            // E conforms to T
U * C::f (D *, E *);            // U is a subtype of T
T * C::f (D *, E * = NULL);     // dflt. value is ignored
T * C::f (D *, E *, int = 0);   // 3rd arg. has dflt. value
T * C::f (D *, E *) throw (X);  // S::f allows any exception
```

Note that conformance is defined using contravariance [Cardelli 1984] of the parameter 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, which only allows covariance of the result types.

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 [Ellis and Stroustrup 1990]. 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 or to type `void*`.

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. That is, 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 programmers.

Observe that the conformance check does not treat classes and signatures symmetrically. While a class can conform to a signature, the reverse is not possible. The consequence of this is that, because of contravariance, a class that is recursive in a parameter type cannot conform to a signature that is recursive in the same parameter type. For example, given the declarations

```
signature S { int f (S *); };
signature T { int f (T *); };
class C { public: int f (C *); };
class D { public: int f (D *); };
```

the signature `T` conforms to `S`, but the classes do not conform to `S` (or `T`). For `C` and `D` to conform to `S`, contravariance would require the parameter type `S*` to conform to `C*` and `D*`, respectively. This asymmetry is necessary, since otherwise in the code

```
S * p = new C;          // illegal
S * q = new D;          // illegal
int i = p->f (q);
```

`C::f` would be called with an argument of type `D*`. Since C++ allows data members of arguments to be accessed directly, an argument has to have the same layout as the declared parameter type; conformance is not good enough. If the parameter is declared to be a signature pointer, and the argument is a signature pointer of a different signature type, there is no problem, since the object pointed to by the signature pointer cannot be accessed directly. In other words, two recursive signatures can be structurally *equivalent* [Amadio and Cardelli 1993], i.e., they can conform to each other. A class can only conform to but cannot be equivalent to a signature. To make the classes in the above example conform to `S`, the arguments would need to be of type `S*` or `T*`.

Our conformance algorithm is related to the notion of *matching* [Abadi and Cardelli 1996; Bruce 1997]. However, since C++ does not have a `selftype` construct to denote the run-time type of an object, conformance does not provide the full flexibility of matching. It would be possible to add `selftype` to C++, both for classes and for signatures. To gain the full flexibility of matching, a pointer of type `selftype` would always need to be represented as a signature pointer. Matching would be a useful mechanism for typechecking template arguments and inheritance in the

presence of `selftype`. However, because of the complexity of the interactions of `selftype` with subtyping defined by inheritance, structural subtyping defined by the conformance check, overriding of nonvirtual member functions, passing of objects by value, and overload resolution, we are not convinced that `selftype` would be useful as a conservative extension of C++.

4. EXAMPLE USES OF SIGNATURES

4.1 Signatures for Retroactive Type Abstraction

The solution to the X-Window object example using signatures is 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 collection of pointers to `XWindowObject`s

```
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 ();  // OpenLookCircle::display
    displayList[1]->display ();  // 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.

4.2 Signatures to Implement Conflicting Type and Class Hierarchies

The solution to the Queue/DEQueue problem presented earlier is also easy using signatures. We simply define an implementation class and two signatures to specify 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 renaming mechanism [Baumgartner and Russo 1995] 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 imple-

mentation of other classes that might implement an abstract stack type, while the latter needlessly clutters the implementation of `DoublyLinkedList`.

5. IMPLEMENTATION TECHNIQUES

In this section, 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 an intermediate representation version of signatures and needs direct access to the typechecking 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 [Stallman 1995] 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 the preprocessor-based 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, and as such, no code needs to be generated. In the particular case above, an interface object must be created to redirect the signature member functions `S::f` and `S::g` to the corresponding class member functions `C::f` and `C::g`. The code for this interface object is generated when compiling the assignment to the signature pointer `p`.

To create such interface objects for any class `C` that conforms to a signature `S`, we first generate an abstract 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 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:
    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`.

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

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

is translated to

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

Assuming the GNU C++ compiler's default layout for virtual function tables, 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 to a pointer of type `void*`, the (implicit) cast expression `(void *)p` needs to be translated to `(void *)*p`, which results in the conversion operator call `(*p).operator void*()`.

Translating a signature pointer to a pointer to an interface object has the advantage that it is straightforward to implement in a preprocessor for a C++ compiler. However, it requires interface objects to be allocated on the heap. Another disadvantage is that assignments with a signature pointer on the RHS can result in

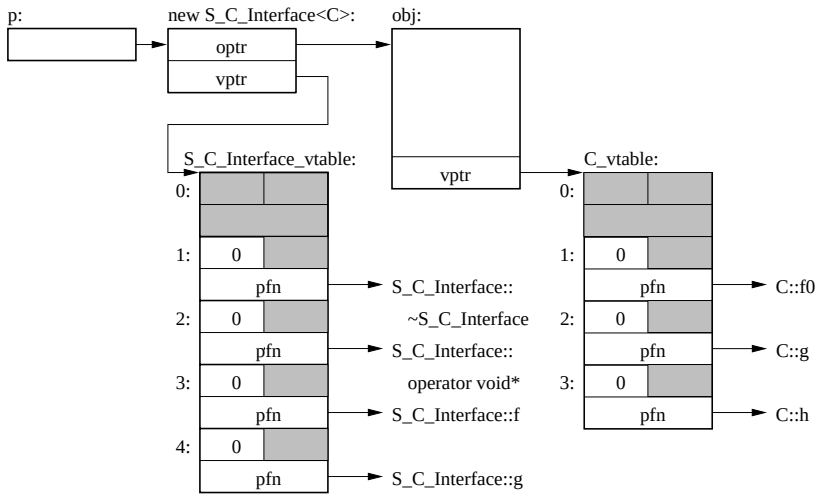

Fig. 1. Preprocessor-based implementation.

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 to `p.f()`. Signature references are implemented exactly the same way as signature pointers.

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. Assigning to a signature pointer, therefore, requires now only two pointer assignments instead of allocating memory followed by three pointer assignments. Assigning one signature pointer to another of the same type now also requires copying two words of storage. Unfortunately, this optimization is not possible for signature pointers that are assigned (to) signature pointers of a different type. In this case, heap allocation is still required.

5.2 Compiler Front-End Implementation

As with the preprocessor-based implementation, the compiler front-end implementation is centered around the basic idea of encapsulating class objects with interface objects. However, by translating signatures to the level of abstraction of a compiler intermediate representation instead of C++ code, we are able to produce more efficient executable code. Although the description of the compiler front-end implementation relies on details of how the GNU C++ compiler [Stallman 1995] compiles C++ classes, the same ideas can be easily implemented in other compilers as well.

The inefficiency of the preprocessor-based implementation is caused by the im-

plementation of a signature member function call as a virtual member function call. When calling a signature member function, two member function calls have to be performed in the generated code: the virtual call to the interface object's member function and the call to the actual class member function. In addition, assigning a signature pointer to another signature pointer of a different type leads to a chain of heap-allocated interface objects, since the information from the RHS virtual function table cannot be copied to the LHS virtual function table. In this case, a signature member function call results in multiple virtual member function calls.

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.

This optimization of inlining the forwarding member functions is only possible if no conversions of argument and result types are necessary. In the general case, if we need to convert arguments or the return value in a signature member function call, we need to generate a conversion function and store a pointer to this function in the signature table. This means that, as in the preprocessor-based implementation, we need two member function calls to perform one signature member function call in the presence of conversions.

5.2.1 *Outline of the Implementation.* In order to outline the structure of the compiler front-end implementation, we initially ignore classes with virtual member functions and multiple and virtual inheritance of classes. Also, we assume no conversions of arguments or the result are necessary.

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 `void`. Since neither C nor C++ allow us to express this, the compiler must generate appropriate casts when using `optr`.

The intermediate code generated for the declaration `S * p = new C;` is equivalent to what would be generated from

```
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. The value of `p`'s `optr` is passed

as the 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 of `C::g`.

5.2.2 *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 information in the signature table to perform a signature member function call correctly.

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 the 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 subobject 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 two flags in a signature table entry to determine whether a nonvirtual member function or a virtual member function 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 delta;        // 'this' adjustment
    short index;        // vtable index if positive
    union {
        void * pfn;     // pointer to function
```

```
        short  vt_off;  // offset to vtable pointer
    };
    short vb_off;       // offset to vbase pointer if pos.
};
```

The data member `delta` contains the value to be added to `this`; `pfn` contains a function pointer in case of a nonvirtual member function, 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 flag to discriminate between nonvirtual and virtual member functions is the sign bit of `index`; for nonvirtual member functions `index` is negative. The data member `vt_off` occupies the same memory location as `pfn`. For typechecking purposes, the compiler needs to cast `pfn` to the appropriate function pointer type.

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 second flag mentioned above is the sign bit of `vb_off`.

Conceptually, a signature table entry is a member function pointer. 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 equivalent declaration for signature table entries would, therefore, be

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

In `vtable_entry_type`, the name `delta2` is used instead of `vt_off`.

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 many 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 many 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 would be necessary to add

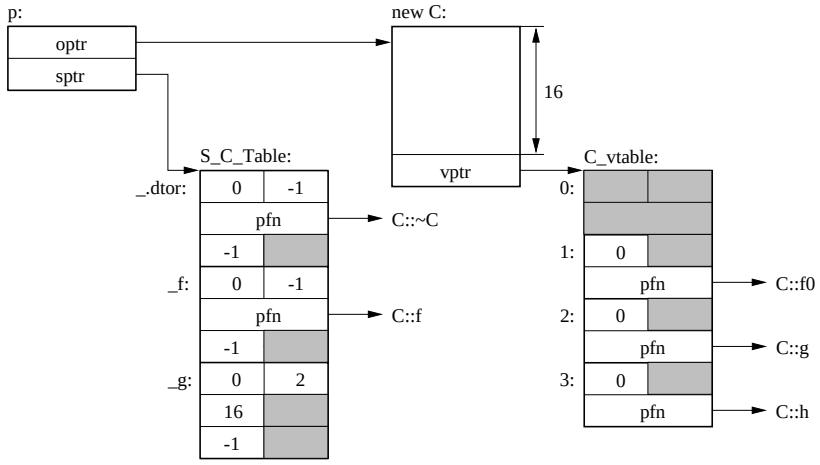

Fig. 2. Compiler front-end implementation.

a `vb_off` data member to `vtable_entry_type` as well and to include additional virtual base pointers in the object.

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

```
struct S_Table {
    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. All the information for initializing the data members of a signature table entry 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 signature pointer `p` results in the data structure displayed in Figure 2.

5.2.3 *Signature Member Function Call.* To call a signature member function, we need to generate a conditional expression that tests the sign of the data member `index` of the signature table entry and, depending on its value, reads the pointer to a nonvirtual member function from the signature table or the pointer to a virtual member function from the virtual function table. 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 to

```
int i = (s    = &(p.sptr->_g),
```

```
base = (s->vb_off < 0)
       ? p.optr
       : *(p.optr + s->vb_off),
this = base + s->delta,
pfn  = (s->index < 0)
       // address of nonvirtual member function
       ? s->pfn
       // address of virtual member function
       : ((**(base + s->vt_off))[s->index]).pfn,
pfn (this, 7, 11)
);
```

where `s`, `base`, `this`, and `pfn` are compiler-generated temporary variables. The pointer `s` contains the address of 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. The function pointer `pfn` is assigned a pointer to the nonvirtual or virtual member function depending on the sign of `s.index`.

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 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.

5.2.4 *Conversions.* So far we have assumed that no conversions of arguments and/or the return value are necessary when calling a signature member function. In the following, we lift this restriction and discuss what 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 `S::f`'s argument, `D*`, is a subtype of `C::f`'s argument type `T*`. 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.

These two 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 `C*`. For our example, the compiler would need to generate the conversion functions

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

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

Since they have a pointer of type `C*` as first argument, we can treat them like nonvirtual member functions declared in class `C`. We, therefore, install pointers to these conversion functions in the signature table entries `S_C_Table._f` and `S_C_Table._g`, respectively. In these signature table entries, `delta` is set to zero, and `index` and `vb_off` are set to `-1`.

The signature member function call `p->f()` now results in the conversion function `S_C_f` being executed. If `C::f` were virtual or declared in a base class of `C`, any adjustments to the `this` pointer and the virtual function dispatch would be generated as part of the call sequence for `this->f()` in `S_C_f`.

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

5.2.5 *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 a class 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 flag in the signature table.

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 a 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 could print a warning message whenever signature table entries have to be copied.

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. This solution can be implemented in any compiler that allows segments of assembly language code generated outside of functions.

When 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. All the work of following the virtual base pointer if necessary, adjusting the object pointer, and possibly looking up the member function's address in the virtual function table is performed at the call site.

The key idea in this solution is to customize the calling sequence for calling a particular class member function and to perform any necessary `this` adjustment and virtual member function call on the callee side. To do this, a customized piece of code, called *thunk*, is generated for each signature table entry that performs all the necessary adjustments and then branches to the class member function. Instead of flags and offsets, the signature table now contains only pointers to these thunks. A signature member function call can now be translated into simply calling the thunk. Such an implementation was proposed in Granston and Russo [1991]. The same idea is used in some compilers for implementing a virtual member function call.

The advantage of this solution is that there is no unnecessary overhead. In particular, it is not necessary to test any flags, and if the `this` adjustment is zero, the addition does not have to be performed. In the common case of calling a nonvirtual class member function that does not need any `this` adjustment, we do not need any thunk. Instead, the signature table entry can point to the member function directly.

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.

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 nonvirtual member function and `C::g` is a virtual member function, both of which require a nonzero offset to be added to the `this` pointer. The thunk needed for calling `C::f` is the following

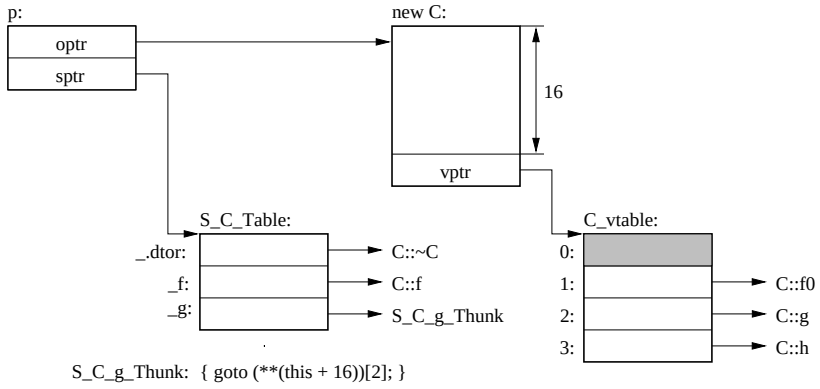

Fig. 3. Thunk-based implementations of signature table and virtual function table.

short piece of code:

```
S_C_f_Thunk:
{
    this = this + DELTA;
    goto C::f;
}
```

where `DELTA` is a compile-time constant determined from `C`'s base classes. 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 data member `optr` from the signature pointer. If `DELTA` were zero, no thunk would be necessary.

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

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

The values `DELTA`, `VT_OFF`, and `INDEX` are constants that can be determined at compile time and are hard-coded into the thunk. Here we assume that virtual function tables are implemented using thunks as well. Otherwise, we would need to select the data member `pfn` from the virtual function table entry. 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 nonzero offset to be added to `this`, the thunk would be

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

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

Also `VB_OFF` would be a constant hard-coded into the thunk.

When compiling an assignment of an instance 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 = { &C::~C, &C::f, &S_C_g_Thunk };
```

The signature table is initialized to contain pointers to the appropriate class member functions whenever possible and to the thunks otherwise.

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.optr, 7, 11);
```

as it did in the simplified outline of the front-end implementation.

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 lightweight function call sequence. A thunk for the nonvirtual member function call would then look as follows:

```
S_C_f_Thunk:
{
    this = this + DELTA;
    // convert argument types
    call C::f;
    // convert return type
    return;
}
```

There is no run-time penalty compared to the front-end implementation if a signature member function does not require conversions.

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 Granston and Russo [1991], 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 nonlocal signature pointer, the solution in Granston and Russo [1991] has to be corrected by allocating signature pointers on the heap.

Observe that the information in thunks that do not contain conversion code 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 functions that are virtual and/or inherited.

6. PERFORMANCE ANALYSIS

In this section, we compare the performance of the three proposed implementation techniques. Since detailed cost analysis in terms of instruction counts and timings are architecture and compiler specific, we first analyze space requirements in terms of words of memory and time requirements in terms of logical operation counts. Following this high-level cost comparison, we present machine instruction counts and measurements of signature member function calls in the front-end and thunk-based implementations and compare them to nonvirtual and virtual class member function calls on the SPARC architecture.

6.1 High-Level Comparison

The memory required for interface objects in the preprocessor implementation of signatures is two words, one for the pointer to the object (`optr`) and one for the pointer to the interface object's virtual function table. This is the same as the size of signature pointers in the other two implementations, where we have the pointer to the signature table (`sptr`) instead of the virtual function table pointer. In the preprocessor implementation, since the interface object is allocated dynamically, an additional word is needed for the pointer to the interface object.

The space needed for the signature table in the compiler front-end implementation is 50% 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 a 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. A correct implementation of virtual function tables would need three words per entry as well. In the thunk implementation, the signature table takes only one third the space, since we only need one pointer per table entry. But additional static storage for the thunks is required.

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 the compiler-based solutions, only two pointer assignments are required.

In the compiler-based implementations, assigning a signature pointer of a different type than the LHS signature pointer takes time proportional to the LHS signature table if signature table entries have to be copied. Since in the preprocessor implementation the LHS interface object simply 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 nonvirtual 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 `index` of a signature table entry.

In the thunks implementation, we do not need to perform any tests when calling a signature member function. This makes a signature member function call as efficient as a standard virtual member function call in the case of calling a nonvirtual member function. When calling a virtual member function through a signature pointer, we have to perform an additional virtual function table lookup.

6.2 Measured Performance

In order to compare our implementation of subtype polymorphism with the existing virtual function implementation, this section compares the execution times of the signature member function dispatch mechanism with the virtual function dispatch mechanism. We only consider the front-end and thunk-based implementations of signature tables and compare them with GCC's standard and thunk-based implementations of virtual function tables, respectively. We also include virtual member function calls (no dispatch) and calls of nonvirtual member functions through a signature pointer (two dispatches) for reference.

We chose not to measure the dispatch mechanism of the preprocessor implementation, since its performance is both poor and very predictable: the time for a virtual function call followed by an additional class member function call. We also did not measure the time for assignments to signature pointers, since in the common case it is very predictable: the time for two regular pointer assignments. Furthermore, as we discuss below, this cost is quickly amortized across signature member function calls.

Since on the SPARC processor the cache cannot be flushed easily, we did not measure cache behavior. Note, however, that the cache behavior of signature member function calls should be better than that of a virtual function calls, since the object does not have to be brought into the cache to get the pointer to the dispatch table.

6.2.1 *Experimental Setup.* All measurements were performed on a SPARCstation 5 with an 85MHz processor, 32MB of main memory, and a write-through, virtually addressed cache. The computer was running Solaris 2.4 in single-user

mode with all network interfaces turned off to minimize the number of random interrupts. The compiler we used was based on the developers' snapshot 960323 of GCC-2.7.2. The test programs were compiled with optimization level `-O2`.

We improved the code generation for signature member function calls and implemented signature tables using thunks. These changes are not in the released version of GCC-2.7.2; they are available from the authors on request. Currently, the compiler is only able to generate thunks for calling nonvirtual class member functions. For measuring calls of virtual member functions through a signature pointer in the thunk-based implementation, we hand-coded the thunks in assembly language.

To get precise measurements with a hardware clock granularity of 500ns, we measured the time needed for one million member function calls: 100,000 loop iterations of 10 calls per iteration. We measured the time needed for such loops of nonvirtual and virtual class member functions called through a regular object pointer and nonvirtual and virtual member functions called through a signature pointer. We also measured the time needed for an empty loop. The member functions did not contain any code other than a return instruction and did not need any adjustment of the `this` pointer. To relate the cost of calling a member function to the cost of argument passing, we measured the calls of member functions with zero to eight parameters. All arguments passed were small integer constants. To prevent the optimizer from generating different loop instructions for the different loops, we generated the assembly instructions for incrementing and testing the loop variable by hand using `asm` statements.

Each measurement was repeated 101 times. The first iteration was used to prime the cache and, thus, discarded. We then calculated the average and the standard deviation for the remaining 100 measurements and threw out samples more than four standard deviations away from the average. This process of discarding samples was repeated twice to eliminate any influence of operating system interrupts. The number of discarded samples ranged from zero to six out of 100 measurements. Finally, we calculated the average of the remaining measurements, subtracted the average time for the empty loop, and divided the result by one million to obtain the time for one member function call.

The average times for the different member function calls are in the range of 50ns to 500ns. The standard error for the average, computed for 90 degrees of freedom (i.e., at least 91 valid samples) is less than 0.75ns for all measurements with a confidence of 99.9%. This means that the true value of the time a member function call takes is within the line width in the plots that follow.

6.2.2 *Compiler Front-End Implementation.* Our GCC front-end implementation still supports the experimental default implementations of signature member functions proposed in Baumgartner and Russo [1995]. The layout of a signature table entry is, therefore, slightly different than described earlier. In particular, the flag to distinguish between nonvirtual and virtual member function calls is in the additional data member `tag` instead of encoded in `index`. This results in one additional load instruction when calling a virtual function through a signature pointer, since both `tag` and `index` have to be loaded instead of just `index`. An additional change from the proposed implementation described earlier is that member func-

tion inherited through virtual inheritance are not called correctly in our current GCC implementation. However, this does not bias the comparison in our favor since GCC's virtual function table implementation fails to handle this case as well.

In the presence of these changes, the pseudocode resulting for the signature member function call `p->f()` is

```
s    = &(p.sptr->_f);
base = p.optr;
this = base + s->delta;
pfn  = (s->tag == 0)
       // address of nonvirtual member function
       ? s->pfn
       // address of virtual member function
       : ((**(base + s->vt_off))[s->index]).pfn;
pfn (this);
```

On the SPARC architecture, GCC compiles the above call into the following assembly language code, where `OFFSET` is the offset of the signature pointer `p` relative to the frame pointer (`%fp`).

```
        ldd [%fp-OFFSET],%o2  ! load p into %o2 and %o3
        ldsh [%o3],%o0        ! load p.sptr->_f.tag
        ldsh [%o3+4],%o1      ! load p.sptr->_f.delta
        cmp %o0,0             ! tag = 0?
        be nonvirt
        add %o1,%o2,%o4       ! p.optr + delta
        ldsh [%o3+8],%o0      ! load p.sptr->_f.vp_off
        ldsh [%o3+6],%o1      ! load p.sptr->_f.index
        ld [%o2+%o0],%o0      ! load vptr from p.optr[vp_off]
        sll %o1,3,%o1         ! index *= 8
        add %o0,%o1,%o0       ! calculate &vptr[index]
        b done
        ld [%o0+4],%o1        ! load vptr[index].pfn
nonvirt:
        ld [%o3+8],%o1        ! load p.sptr->_f.pfn
done:
        call %o1,0            ! call class member function
        mov %o4,%o0           ! this = p.optr + delta
```

The path length through this code for calling a nonvirtual member function is ten instructions, four of which are load instructions. For calling a virtual member function, the path length is 15 instructions, seven of which are load instructions.[3]

Note that there are possibilities for further optimizing the above code. Since one branch of the conditional expression consists of a single instruction, this instruction could be moved into the delay slot of the branch instruction by enabling the annul bit of the branch instruction [SPARC International 1991]. This would eliminate the

[3]The instruction immediately following a branch or call instruction, i.e., the instruction in the *delay slot*, is executed prior to the branch and call, respectively.

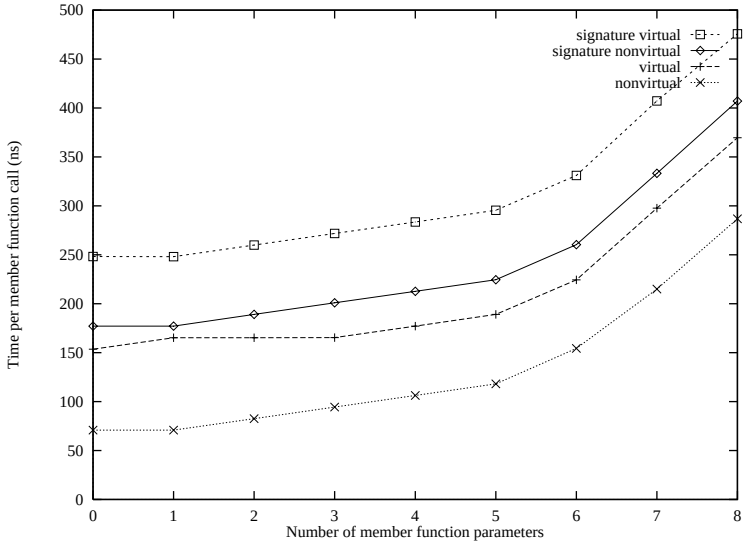

Fig. 4. Cost of member function calls in the front-end implementation.

unconditional branch to label `done`. The last instruction in the sequence, which is executed before the call instruction, only moves the `this` pointer into the proper register. A better register allocator could eliminate this instruction, freeing up the delay slot for loading an argument. Finally, the shift instruction (`sll`) used in indexing the virtual function table could be eliminated by storing the word offset to the virtual function table entry in `index` rather than the array index. Even if it is not eliminated, a better instruction scheduler could move the shift instruction before the preceding load (`ld`) instruction to improve pipeline performance.

The additional `ldsh` instruction to load the data member `tag` does not cost anything on a SPARC, since, due to pipelining, the two `ldsh` instructions to load `vp_off` and `index` take approximately the same time as one `ldsh` instruction, since they fetch from consecutive memory locations.

Figure 4 shows, from bottom to top, the measured times for nonvirtual calls, virtual calls, nonvirtual calls through a signature pointer, and virtual calls through a signature pointer. Note that even in this straightforward implementation, calling a nonvirtual member function through a signature pointer is only 7% (for one argument) to 22% (for three arguments) slower than a virtual function call. Note also that the execution time does not increase linearly with the number of arguments. The flat areas in the plots result from instruction scheduling and pipelining. After five arguments the plots become steeper, since additional arguments cannot be passed in registers but have to be stored on the stack.

6.2.3 *Implementation with Back-End Support.* In the thunk-based implementation, the calling sequence for the signature member function call `p->f()` is simply

```
p.sptr->_f (p.optr);
```

which GCC compiles to

```
        ldd [%fp-OFFSET],%o0    ! load p into %o0 and %o1
        ld [%o1],%o2            ! load p.sptr->_f
        call %o2,0              ! call member fnct. or thunk
        nop                     ! empty delay slot
```

This calling sequence is very similar to that of a virtual function call. Instead of a single `ldd` instruction to load both the object pointer and the pointer to the signature table into a register pair, two `ld` instructions would be needed for a virtual function call to load the object pointer and the pointer to the virtual function table.

Since in our test cases no `this` adjustment or argument conversion is necessary, there is no thunk for calling a nonvirtual member function. For calling a virtual member function, we use the following (currently hand-coded) thunk:

```
thunk:
        ld [%o0],%g5            ! load vptr from *this
        ld [%g5+8],%g5          ! load vptr[2]
        jmp %g5                 ! jump to virt. member fnct.
        nop                     ! empty delay slot
```

For calling a nonvirtual member function, the path length is three instructions (excluding the `nop` instruction in the delay slot), two of which are load instructions. In the case of a virtual member function, the path length is six instructions, four of which are load instructions.

To be able to compare the performance of properly optimized call sequences, we hand-optimized the generated assembly code for all member function calls (including class member function calls). The optimizations we performed manually could easily be done by an optimizing compiler. Since GCC's instruction scheduler does not always interleave load instructions with register instructions, we manually changed the order of some instructions. We also eliminated the register-to-register `mov` instructions to get the object pointer into register `%o0`. With flow analysis based on registers instead of on declared types, the compiler's register allocator would be able to remove these `mov` instructions as well.

Figure 5 shows the times for the thunk-based implementation. Neither the virtual function call nor the nonvirtual signature member function call used a thunk. The tables contained direct pointers to the member function.

Note that the performance of calling a nonvirtual member function through a signature pointer is almost identical to the performance of a virtual function call and is actually 8% better when passing less then four arguments. This is not surprising, given that the code for both cases is almost identical.

Since in our dispatch mechanism the pointer to the dispatch table is kept in the signature pointer instead of in the object, we can keep the pointer in a register across multiple signature member function calls. In the virtual function dispatch mechanism, this optimization is not as straightforward and is not even always possible, since an object can have multiple virtual function tables. Figure 6 shows the times for the thunk-based implementation with all object pointers and signature pointers in registers. The calling sequences for signature member function calls are the same as above except that the `ldd` instruction for loading the signature pointer has been eliminated. This optimization results in a speedup of 4% (for eight arguments) to 20% (for zero or one arguments) over virtual function calls.

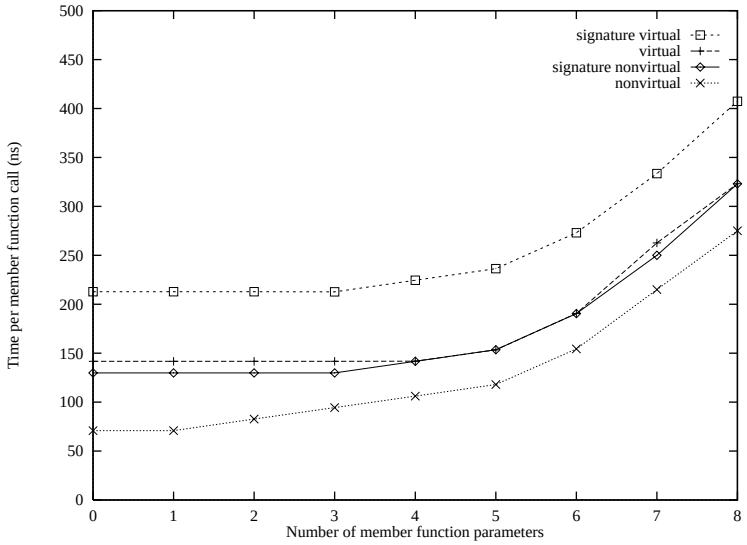

Fig. 5. Cost of member function calls in the implementation with back-end support.

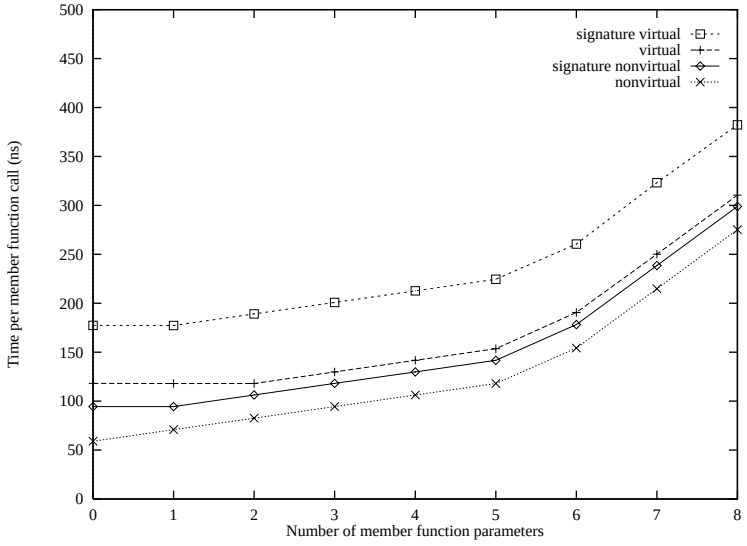

Fig. 6. Cost of member function calls in the implementation with back-end support and all pointers in registers.

As a final observation, note that the added cost of the signature pointer assignment over a class pointer assignment is quickly amortized by keeping the signature pointer in registers across multiple signature member function calls. Not counting the improved pipeline behavior of the `ldd` instruction over two `ld` instructions, the cost of assigning to the data member `sptr` in a signature pointer is amortized after only two signature member function calls per assignment.

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. Not only is our language extension more flexible for type abstraction and subtyping than classes and inheritance, it also allows for a more efficient implementation of the dispatch mechanism necessary for subtyping, as we have shown by our performance measurements.

A signature is a language construct that allows the separation of the concepts of abstract and concrete types. Using signature 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++).

The signature language construct is very similar to interfaces in Java [Gosling et al. 1997] and is also related to types in Russell [Donahue and Demers 1985], ML's signatures [MacQueen 1985; MacQueen 1988], Haskell's type classes [Hudak et al. 1992], definition modules in Modula-2 [Wirth 1985], interfaces in Modula-3 [Cardelli et al. 1992], abstract types in Emerald [Black et al. 1986], type modules in Trellis/Owl [Schaffert et al. 1986], categories in Axiom [Jenks and Sutor 1992] and its predecessor Scratchpad II [Sutor and Jenks 1987; Watt et al. 1990], and types in POOL-I [America and van der Linden 1990].

While interfaces in Java are syntactically closely related to signatures, Java uses named conformance (i.e., a class must declare to implement an interface using the `implements` keyword) instead of structural conformance inferred by the compiler. The type system of C++ with signatures actually comes closest to those of POOL-I and Axiom. Unlike C++, POOL-I does not have overloading and private and protected member functions, and Axiom is an abstract datatype language. 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. Similarly to C++ with signatures, ML, Modula-2, and Modula-3 allow a clean separation of specification and implementation. However, ML and Modula-2 do not have classes but only modules, while Modula-3 has both classes and modules but provides interfaces only for modules but not for classes. Russell and Haskell have notions related to signatures, but both lack classes and other object-oriented features. Emerald has first-class types instead of classes, and

Trellis/Owl has a class hierarchy with only type inheritance but no implementation inheritance.

Syntactically, signatures are also very similar to interfaces in the Object Management Group's CORBA Interface Definition Language (IDL) [Object Management Group 1995], which allows the description of interfaces for distributed objects independent of the programming language. However, IDL interfaces are simply translated into abstract classes in C++. Implementations of these interfaces still need to inherit from the generated abstract classes. This does not solve the problem of retroactive abstraction, nor does it solve the problem of decoupling subtyping from inheritance.

Note that the subtype relationship defined by multiple inheritance in C++ is subsumed by signature conformance. Assume class `D` is a derived class of `C` defined by C++-style public inheritance and overriding of virtual member functions. Since `D` conforms to any signature that `C` conforms to, an instance of `D` can be assigned to a signature pointer wherever an instance of `C` was assigned before. The runtime dispatch for signature member function calls gives the same polymorphism as a virtual function dispatch. Therefore, signature conformance subsumes any subtype relationship defined by inheritance in which only virtual member functions are redefined. The only case where signature conformance cannot be used instead of class inheritance to define a subtype relationship is when the derived class redefines a *nonvirtual* member function inherited from a base class. This use of inheritance, however, is usually considered bad programming style, since semantically it breaks the subtype relationship between the derived class and the base class. In addition, when multiple inheritance was used only for subtyping purposes, with signature conformance we do not need to pay the cost of adjusting the *this* pointer and of following pointers to virtual bases.

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. With subtype polymorphism defined by signature conformance and implemented through signature pointers and references, it would no longer be necessary for inheritance to define a subtype relationship at all. Therefore, virtual function tables would no longer be needed as a dispatch mechanism to achieve subtype polymorphism. While virtual member functions are still useful for code reuse, they could be implemented differently if inheritance does not define a subtype relationship. For example, by trading space for execution speed, we could implement inheritance by copying and recompiling the code from the base class into the derived class. 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 not possible, as they would affect the behavior of existing programs, future programming languages could take advantage of this separation.

8. AVAILABILITY

Parts of the language extension have been implemented in G++ as a compiler extension [Baumgartner 1995]. The implementation is included as part of the GCC

distribution starting with GCC version 2.6.0.

ACKNOWLEDGEMENTS

We would like to thank Konstantin Läufer and Michal Young for reading parts of the article and for providing many valuable comments, and Andy Muckelbauer for numerous discussions about the implementation. The referees' comments resulted in a greatly improved presentation.

REFERENCES

Abadi, M. and Cardelli, L. 1996. On subtyping and matching. *ACM Trans. Program. Lang. Syst. 18,* 4 (July), 401–423.

Amadio, R. M. and Cardelli, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15,* 4 (Sept.), 575–631.

America, P. and van der Linden, F. 1990. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the OOPSLA '90 Conference on Object-Oriented Programming Systems, Languages, and Applications*. DELETE, DELETE. *SIGPLAN Not. 25,* 10 (Oct.), 161–168.

Baumgartner, G. 1995. Type abstraction using signatures. In *Using and Porting GNU CC*, R. M. Stallman, Ed. Free Software Foundation, Cambridge, Mass, Section 7.6, 180–182. Available as part of the GCC-2.7.2 distribution.

Baumgartner, G. and Russo, V. F. 1995. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Softw. Pract. Exper. 25,* 8 (Aug.), 863–889.

Black, A., Hutchinson, N., Jul, E., and Levy, H. 1986. Object structure in the Emerald system. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*. DELETE, DELETE. *SIGPLAN Not. 21,* 11 (Nov.), 78–86.

Bruce, K. B. 1997. Typing in object-oriented languages: Achieving expressibility and safety. *ACM Comput. Surv.* To be published.

Cardelli, L. 1984. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds. Lecture Notes in Computer Science, vol. 173. Springer-Verlag, New York, 51–67.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. 1992. Modula-3 language definition. *SIGPLAN Not. 27,* 8 (Aug.), 15–43.

Cook, W. R., Hill, W. L., and Canning, P. S. 1990. Inheritance is not subtyping. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 125–135.

Donahue, J. and Demers, A. 1985. Data types are values. *ACM Trans. Program. Lang. Syst. 7,* 3 (July), 426–445.

Ellis, M. A. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass.

Gosling, J., Joy, B., and Steele, G. 1997. *The Java Language Specification*. Addison-Wesley, Reading, Mass.

Granston, E. D. and Russo, V. F. 1991. Signature-based polymorphism for C++. In *Proceedings of the 1991 USENIX C++ Conference*. USENIX Assoc., Berkeley, Calif., 65–79.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. 1992. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *SIGPLAN Not. 27,* 5 (May), Section R.

Jenks, R. D. and Sutor, R. S. 1992. *AXIOM: The Scientific Computation System*. Springer-Verlag, New York.

MacQueen, D. B. 1985. Modules for Standard ML. *Polymorphism 2,* 2 (Oct.), PAGES.

MacQueen, D. B. 1988. An implementation of Standard ML modules. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*. ACM, New York, 212–223.

Object Management Group 1995. *The Common Object Request Broker: Architecture and SpecificationDOT*, Rev. 2.0ED ed. Object Management Group, Framingham, Mass.

Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. 1986. An introduction to Trellis/Owl. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*. DELETE, DELETE. *SIGPLAN Not. 21,* 11 (Nov.), 9–16.

Snyder, A. 1986. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*. DELETE, DELETE. *SIGPLAN Not. 21,* 11 (Nov.), 38–45.

SPARC International 1991. *The SPARC Architecture ManualDOT*, Ver. 8, SAV080SI9106ED ed. SPARC International, Menlo Park, Calif.

Stallman, R. M. 1995. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Mass. Available as part of the GCC-2.7.2 distribution.

Sutor, R. S. and Jenks, R. D. 1987. The type inference and coercion facilities in the Scratchpad II interpreter. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*. DELETE, DELETE. *SIGPLAN Not. 22,* 7 (July), 56–63.

Watt, S. M., Jenks, R. D., Sutor, R. S., and Trager, B. M. 1990. The Scratchpad II type system: Domains and subdomains. In *Computing Tools for Scientific Problem Solving*, A. M. Miola, Ed. Academic Press, London, 63–82.

Wirth, N. 1985. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin.

Received July 1995; revised May 1996; accepted July 1996