Implementing Signatures for C++

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

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

Abstract

In this paper we overview the design and implementation of a language extension to C++ for abstracting types and for decoupling subtyping and inheritance. This extension gives the user more of the flexibility of dynamic typing while retaining the efficiency and security of static typing. We discuss the syntax and semantics of this language extension, show examples of its use, and present and analyze the cost of three different implementation techniques: a preprocessor to a C++ compiler, an implementation in the front end of a C++ compiler, and a low-level back-end based implementation.

1 Introduction

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

The remainder of the paper is structured as follows. First we present motivation for the addition of a type abstraction facility other than classes to C++. We then present in some detail the syntax and semantics of signatures relative to their implementation. We follow with examples which illustrate how signatures solve the problems presented in the motivation section. The final sections of the paper discuss and compare three different implementation possibilities, and analyze the costs of each. The primary intent of this paper is to detail these implementation techniques. For this reason, the motivation and language specifications are of necessity brief. The reader interested in a more detailed motivation and complete syntax and semantics is referred to [4].

2 Motivation

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

1. In some cases, it is difficult (if not impossible) to retroactively introduce abstract superclasses for the purpose of type abstraction.

2. Using the same construct (class inheritance) for type abstraction and code sharing limits the power of both and unnecessarily couples implementation to interface.

3. The hierarchy of type abstraction and the class hierarchy of implementation may not easily agree.

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 of the need to introduce type abstractions of existing class hierarchies is illustrated in [15]. Summarizing their presentation, suppose we have two libraries containing hierarchies of classes for X-Windows display objects. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. Further suppose all the classes in each hierarchy implement a `display()` and a `move()` member function, 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 descrimination 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 no source code is available for the two libraries (only header files and binaries are provided) retroactive code modification is not possible. The only choices remaining are to use a discriminated union for the display list elements, to use multiple inheritance to implement a new set of leaf classes in each hierarchy, or to use a hierarchy of forwarding classes[1]. The former solution is rather inelegant, the latter two clutter up the name space with 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 abstraction by inheriting the abstraction class. The nature of the restrictions in this example prevent us from doing this. What we would like is a type abstraction mechanism which does not rely on classes and, therefore, leaves classes free to be used for implementation specification. Likewise, the adherence of a particular class to a type abstraction 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 type abstractions at a later time without altering any implementations.

2.2 Separation of Type and Class Hierarchies

Another problem with a single class hierarchy defining both abstract data types and their implementations is that as the type hierarchy becomes more complex, it might become necessary to duplicate code, as an example from computer algebra [5, 4] demonstrates.

Consider the abstract type `general_matrix` with subtypes `negative_definite_matrix` and `orthogonal_matrix`. Both subtypes have additional functions, such as `inverse()`, which are not present in general matrices. Assume we have several different implementations of those abstract types, namely `dense_matrix`, which implements matrices as two-dimensional arrays, `sparse_matrix`, which uses lists of triples, and `permutation_matrix`, which is implemented as a special case of sparse matrices that takes advantage of permutation matrices only having one element in each row and column.

If we try to model this example with a class hierarchy, we end up either duplicating code or violating the type hierarchy. While `dense_matrix` can be made a subclass of the abstract virtual classes `general_matrix`, `negative_definite_matrix`, and `orthogonal_matrix` using multiple inheritance, we cannot do the same for `sparse_matrix`. Doing so would make `permutation_matrix`, which is a subclass of `sparse_matrix`, an indirect subclass of `negative_definite_matrix`. Since permutation matrices are positive definite, this would violate the type hierarchy. The alternative of having a separate class `sparse_negative_definite_matrix` is not satisfying either.

Similar arguments have been given in the literature to show that the `collection` class hierarchy of SMALLTALK-80 [14] is not appropriate as a basis for subtyping. While the problem does not arise with dynamic typing, it becomes an issue when trying to make SMALLTALK-80 statically typed while retaining most of its flexibility. The solution is to factor out the implementation aspect of classes into prototypical objects [18] or to factor out the type aspect into interfaces [7, 9].

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

2.3 Implementation of Conflicting Type and Class Hierarchies

Often the abstract type hierarchy and the implementation class hierarchy cannot be made to agree. An example similar to one in [22] illustrates this point. Consider two abstract types `queue` and `dequeue` (doubly ended queue). The abstract type `dequeue` provides the same operations as `queue` and 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 [10] it is argued that in order for a type system to be sound it should not be possible to use inheritance for subtyping purposes and also allow the removal of operations. Most object-oriented languages choose instead to restrict the use of inheritance for code sharing to situations where there is also a subtype relationship, and to disallow inheriting only a portion of the superclass.

3 Syntax and Semantics of the Signature Language Extension

We term the key language construct we add to C++ to support type abstraction a *signature*. It is related to types in Russel [11], ML's signatures [19, 20], Haskell's type classes [16], definition modules in Modula-2 [26], interface modules in Modula-3 [8], abstract types in Emerald [6], type modules in Trellis/Owl [21], categories in Axiom [17] and its predecessor Scratchpad II [24, 25], and types in POOL-I [3].

The type system of C++ with signatures comes closest to those of Axiom and POOL-I. Russel, ML, Haskell, and Modula-2 don't have class types, Modula-3 only has interfaces for modules but not for classes. Emerald has first-class types instead of classes, and Trellis/Owl has a type hierarchy in which type information but no implementation is inherited. Domains in Axiom differ from classes by having method dispatch on all argument types and on the return type. Compared to C++, POOL-I doesn't have private and protected member functions and overloading. While both categories and domains in Axiom and types in POOL-I are first class, signatures and classes in our C++ extension are not, which makes the type system slightly less expressive but allows for a more efficient implementation and for complete type checking at compile time.

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

3.1 Signature Declarations

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

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

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

[2]The additional features of signature inheritance, the `sigof` construct (as in [15]), and opaque types are left out since they only affect the type checking phase of the compiler. For information on those constructs, as well as for more details on the semantics of signatures, see [4].

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 (`typedefs`), constant declarations, member function declarations, operator declarations, and conversion operator declarations are allowed within a signature declaration. Specifically:

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

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

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

3.2 Signature Pointers and References

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

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

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

Consider the following declarations,

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

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

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

3.3 The Conformance Check

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

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

1. For every member function, operator, and conversion operator declared in `S`, there must be a public declaration of the same function or operator in `C`. Furthermore, this declaration must have the same name and conforming return and argument types. Also, every signature contains an implicit destructor declaration. This destructor is matched with the class's destructor if defined or with the default destructor otherwise. Specifically, a signature member function `S::f` conforms to a class member function `C::f` iff
   - The type of every argument of `S::f` conforms to the type of the corresponding argument of `C::f` and
   - The return type of `C::f` conforms to the return type of `S::f`.
2. For every type definition in `S`, there is a public type definition of the same name and conforming structure in `C`.
3. For every constant declaration in `S`, there is a constant declaration of the same name and conforming type in `C`.

The conformance check for testing the conformance of one signature to another is exactly the same.

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

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

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

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

```
T * C::f (D *, E *);      // since the types are the same
T * C::f (D *, D *);      // since 'D' is a supertype of 'E'
T * C::f (T *, E *);      // since 'D' conforms to 'T'
T * C::f (T *, T *);      // since both 'D' and 'E' conform to 'T'
D * C::f (D *, E *);      // since 'D' conforms to 'T'
E * C::f (D *, E *);      // ...
U * C::f (D *, E *);
T * C::f (D *, E *, int = 0);
```

Note that conformance (and therefore subtyping) is defined using contravariance of the argument types of member functions unlike the subtype relationship defined by class inheritance. This is necessary to make subtyping sound and to avoid run-time type errors as they can occur in C++'s inheritance-based subtyping.

If several member functions of `C` conform to one member function of `S`, we find the one that conforms best using a variant of C++'s algorithm for finding the function declaration that best matches the call of an overloaded function [12].

If a member function of `C` conforms to several member functions of `S`, an error is reported by the compiler. This restriction 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. However, we feel that any such rule would be sufficiently complex to confuse users.

3.4 Default Implementations

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

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

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

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

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

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

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

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

    /* ... */
}
```

In the function `foo` above it cannot be known whether `p` will use `T`'s default implementation or not. If the default implementation is used, there will be a run-time type error in the assignment to `q`.

Note that this is the only case where a run-time type check is necessary, in all other cases conformance can be fully checked at compile time. To warn of the possibility of a run-time type error, our compiler prints a warning message when generating the run-time test.

3.5 Views

The earlier presentation of the signature conformance check required that member functions declared by a signature be matched by class member functions of the same name and the same type. This may not be completely realistic as it might be the case that the implementor of the class simply chose a different name for the same function.

For example, suppose that in the X-Windows object manager example the function to display a window on the screen is called `display()` in `OpenLookObject` but `show()` in `MotifObject`. To build a display list of objects from both hierarchies, it would be necessary to rename the member function in one of those hierarchies.

Such renaming could be partially achieved through default implementations, but this is not sufficiently powerful to, for example, swap the names of two member functions. Instead, to rename class member functions, or to *view* a class to be an implementation of a signature type in a different way, we provide the following syntax:

```
S * p = (S *; foo = bar) new C;
```

This expression associates the class member function `bar` with the signature member function `foo`. Renaming expressions can be separated by commas in order to rename multiple member functions.

Conceptually, the renaming operations are performed in parallel to allow swapping of member function names. This allows, for example, a rational number class to be viewed as an implementation of the abstract type `Group` in two different ways, as a multiplicative group and as an additive group.

If the renamed member function is overloaded, all overloaded definitions are renamed in the same way. There is no syntax for selectively renaming functions depending on their return and argument types. While this would be possible, we feel it would make the syntax of views too complicated.

A similar renaming mechanism can be found in the computer algebra system VIEWS implemented in SMALLTALK [1, 2] or in the algebraic specification language OBJ3 [13].

3.6 Constants

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

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

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

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

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

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

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

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

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

4 Example Uses of Signatures

4.1 Signatures for Retroactive Type Abstraction

The solution to the `XWindowsObject` example using signatures is actually quite simple. All that is needed is to introduce a signature to define the abstract type `XWindowsObject`,

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

and to implement the display list as a list of pointers `XWindowsObject`'s,

```
XWindowsObject * displayList[NELEMENTS];
```

Given a pair of implementation hierarchies such as:

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

and

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

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

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

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

```
    return 0;
}
```

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

We can even make this example more compelling. Consider the possibility that the Motif class hierarchy used the name `show` rather than `display` for its rendering operation. We would simply need to add a view cast when assigning an object from the Motif hierarchy to our display list:

```
    displayList[1] = (XWindowsObject *; display = show) new MotifSquare;
```

4.2 Signatures to Separate Type and Class Hierarchies

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

```
signature general_matrix           { /* ... */ };
signature negative_definite_matrix { /* ... */ };
signature orthogonal_matrix        { /* ... */ };
```

Since `negative_definite_matrix` and `orthogonal_matrix` conform to `general_matrix` they are also subtypes of `general_matrix`. By using inheritance of signatures, as defined in [4], we can simplify the definition of the latter two signatures.

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

```
class dense_matrix  { /* ... */ };
class sparse_matrix { /* ... */ };
class permutation_matrix : sparse_matrix { /* ... */ };
```

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

4.3 Signatures to Implement Conflicting Type and Class Hierarchies

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

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

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

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

queue<int> * q1 = new DoublyLinkedList<int>;
dequeue<char *> * q2 = new DoublyLinkedList<char *>;
```

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

5 Implementation Techniques

In this section, we present three options for implementing signatures. The first method could be used in a compiler preprocessor (e.g., a `cfrontfront`) that translates C++ with signatures into C++ without signatures. The second is a compiler based implementation that produces a C-level code version of signatures and needs direct access to the type checking phases of a C++ compiler, but is independent of the compiler back-end and machine architecture. This method has been implemented in the GNU C++ compiler[23] 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 knowledge of 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 main idea of implementing signatures 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 can then be implemented as regular C++ pointers that point to those interface objects.

Consider the declarations

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

S * p = new C;
```

and assume `C` conforms to `S`. The signature declaration itself is simply a type declaration, we do not need to generate any code for it. The code for the interface object is generated when compiling the assignment to the signature pointer `p`.

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

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

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

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

For creating the classes of interfaces 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 (void)  { delete optr; };
    int f (void)           { return optr->f (); };
    int g (int x, int y)   { return optr->g (x, y); };
};
```

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

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

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

is translated to

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

Since a signature pointer is a standard C++ pointer in this scheme, we don't need to do anything special to compile a signature member function call. The call `p->f ()` will execute `S_C_Interface<C>::f`, which in turn calls `C::f`.

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

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

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

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

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

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

    // code for default implementation
}
```

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

The above solution has the advantage that it is straightforward to implement in a preprocessor for a C++ compiler. The disadvantage is that it requires interface objects to be allocated on the heap. 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 = new C;
```

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

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

5.2 Compiler Front-End Implementation

In the above implementation, there are two sources of inefficiency, which we try to overcome in our GNU C++ implementation. One of the problems is that for calling a signature member function, we have two member functions calls, the call to the interface object's member function and the call to the class member function. The other problem is that interface objects can become rather large.

In order to optimize signature member function calls we can inline the member function calls of class `S_C_Interface<C>` by storing some of the information contained in those member functions in a special table called the *signature table*. A signature table, which is similar in structure to a virtual function table, depends only on a signature and conforming class pair, and can, therefore, be made static.

The key to optimizing the space requirements of interface objects is to observe that signature constants, as well as the default implementation flags, can be stored in static memory as well. The values of both signature constants and default implementation flags can be determined in the class conformance check, they don't depend on the actual class object. The obvious place to store them is, again, in the signature table.

To keep the structure of signature tables in our implementation simpler, we impose a slight restriction on the expressiveness of the language by requiring *strict conformance* between a signature and a class. Strict conformance means that a signature member function and the corresponding class member function need to have the same number of arguments, exactly the same argument types, and exactly the same return type. Without imposing this restriction, we might need to convert argument types or the return type in a signature member function call, but we don't have place in a signature table to store the conversion code. In a following section, we will show how to lift this restriction without causing run-time overhead.

Simplified Version

Let us ignore default implementations, signature constants, classes with virtual member functions, and multiple inheritance of classes for now. For the above signature declaration with member functions `f` and `g`, the compiler generates an internal representation of the following structure of member function pointers:

```
struct S_Table {
    const void * _S_destr;
    const int    (S::*_f) (void);
```

```
        const int    (S::*_g) (int, int);
    };
```

where the field `_S_destr` represents the destructor that is implicitly declared in every signature. The type `S_Table` will be the type of signature tables for signature `S`.

In the previous solution, an interface object contained 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, and we store the interface object in the signature pointer. This leads us to the following type declaration for signature pointers:

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

Actually, the type of `optr` can be a pointer to *any* object. When using `optr` the compiler must generate appropriate casts.

We now generate code for the declaration '`S * p = new C;`' as follows:

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

To initialize the signature table `S_C_Table`, we need to cast `C::f` and `C::g` to member functions of `S`. If `C` doesn't have a destructor, we use the default destructor. Since C++ doesn't allow us to cast to a member function type or to take the address of a destructor, this has to be 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 was in an inner scope, the signature table declaration needs to be moved out of this scope into the file scope.

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

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 field `_g`, which is `C::g` in our example. We need to pass the value of `p`'s `optr` field as first argument, so that `C::g` gets a pointer to the right object passed for its implicit first parameter called `this`.

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

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

Fine Details

As noted earlier, we need to store signature constants and flags to test for default implementations in the signature table. Similarly, we need additional information to handle multiple inheritance of classes and classes with virtual member functions.

If a signature member function is implemented by a virtual class member function, we don't know the address of the function to call until run time. It is, therefore, not possible to store the class member function's address in the signature table. What we do instead is store the offset into the class's virtual function table together with a flag that tells us to perform a virtual member function call. Together with calls to default implementations, we have now three different kinds of signature member function calls. We store the two flags needed to test for those cases in a short integer, which will be tested for a zero, positive, or negative value.

In case of multiple inheritance, an object in GCC might use several virtual function tables, one for each parent class. We need to find the right virtual function table to use for each member function call. The solution is to add another field to the signature table entry that contains the offset into the object where we can find a pointer to the proper virtual function table.

Also in GCC, member functions are implemented as regular functions that take a pointer to the object, called `this`, as first argument. To pass the object correctly in the presence of multiple inheritance of member functions, an offset has to be added to `this` that depends on the place of the member function in the class hierarchy. As in virtual function tables, we need to store this offset with each entry in a signature table.

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

```
struct sigtable_entry_type {
    short  code;
    short  offset;
    union {
        void * pfn;
        struct {
            short vpoffset;
            short vtoffset;
        };
    };
};
```

The `code` field contains the flags mentioned above, the `offset` field contains the value to be added to `this`, `pfn` contains a function pointer in case of a non-virtual member function or a default implementation, and, in case of a virtual member function, `vpoffset` and `vtoffset` contain the offset of the virtual function table pointer in the object and the offset into the virtual function table, respectively. The fields `vpoffset` and `vtoffset` occupy the same memory location as `pfn`. For type checking purposes, the compiler needs to cast `pfn` to the appropriate function pointer types.

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

```
struct S_Table {
    const sigtable_entry_type _S_destr;
    const sigtable_entry_type _f;
    const sigtable_entry_type _g;
};
```

In addition, for each uninitialized constant in the signature, we insert a field declaration into the signature table type. All the information for initializing the fields of a signature table entry and for initializing constants can be obtained at compile time from the class of the object on the RHS of a signature pointer assignment or initialization.

When assigning a signature pointer to another signature pointer of the same type, we simply copy the two fields. If the types are not the same, we may need to initialize the signature table for the LHS signature pointer at run time by copying the corresponding fields from the signature table to which the RHS signature pointer points. If the signature table entries to be copied form a contiguous block of data in the RHS signature table and the order of the table entries is the same for both signature tables, the compiler can avoid copying by letting the LHS `sptr` point into the RHS signature table.

If copying is unavoidable, the compiler should print a warning message. Independent of whether copying table entries is necessary, if the RHS signature contains a default implementation where the LHS signature only has a member function *declaration*, the compiler needs to generate a run-time test and should print a corresponding warning message.

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

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

from our example above is now translated into

```
int i = (this = p.optr,
         s = p.sptr->_g,
         s.code == 0
             ? s.pfn (this + s.offset, 7, 11)        // non-virtual call
             : (v = (*p.optr[s.vpoffset])[s.vtoffset],
                v.pfn (this + s.offset, 7, 11)       // virtual call
               )
        );
```

where `this`, `s`, and `v` are compiler generated temporary variables to hold the pointer to the object, the signature table entry, and the virtual function table entry, respectively. These temporary variables can be kept in registers.

In case the signature member function `g` has a default implementation, the signature member function call becomes

```
int i = (this = p.optr,
         s = p.sptr->_g,
         s.code == 0
             ? s.pfn (this + s.offset, 7, 11)        // non-virtual call
             : (v = (*p.optr[s.vpoffset])[s.vtoffset],
                s.code > 0
                    ? v.pfn (this + s.offset, 7, 11) // virtual call
                    : s.pfn (p, 7, 11)               // default impl call
               )
        );
```

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

5.3 Implementation with Back-End Support

A place that leaves room for optimization in the previous solution is the conditional expression for calling a signature member function. A possible way to optimize signature member function calls is to store (pointers to) pieces of code, or *thunks*, in the signature table instead of flags and offsets. The thunks contain the appropriate code to set the `this` pointer correctly and branch to the class member function or perform a virtual function call. Such an implementation was proposed in [15].

Each thunk only contains the code necessary to call one specific class member function. We do not need to test any flags but just branch to the thunk, which does the right thing for the member function we want to call. Signature table entries are now reduced to a single function pointer again.

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

```
struct S_Table {
    void * _f;
    void * _g;
};
```

Given a class `C` conforming to `S`, assume that `C::f` is a non-virtual member function that doesn't need any offset to be added to `this` and that `C::g` is a virtual member function that requires adding a non-zero offset of `this`. The thunk needed is the following short piece of code:

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

Before branching to the thunk, the compiler will have set up the activation record correctly for calling `C::f`. In particular, all the arguments were either pushed onto the stack or are in registers. The value passed for the first argument, `this`, is the signature pointer. Before branching to `C::f`, we need to extract the `optr` field so that `this` points to the object. With the right layout of the activation record in registers or on the stack, no work needs to be done for adjusting `this`. In that case we can get rid of the thunk altogether and store a direct pointer to `C::f` in the signature table entry.

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

```
S_C_g_Thunk:
{
    temp = this;
    this = this.optr + OFFSET;
    goto (*(temp.optr)[VPOFFSET])[VTOFFSET].pfn;
}
```

The values `OFFSET`, `VPOFFSET`, and `VTOFFSET` are constants that can be determined at compile time and are hard-coded into the thunk.

When a default implementation is used, the entry in the signature table can point to the code of the default implementation directly; we don't need a thunk in this case.

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

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

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

Instead of a conditional expression, the signature member function call

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

now reduces to

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

A big advantage is that we could include code for converting argument types in a thunk. This code would simply go into the thunk 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 either have a second thunk which does nothing else but perform the necessary conversion, or we could call, instead of branching to, the class member function from the thunk using a light-weight function call sequence. The thunk for the non-virtual member function call could look then as follows:

```
{
    this = this->optr + offset;
    // convert argument types
    temp = ret_addr;
    ret_addr = L;
    goto C::f;
 L: // convert return type
    ret_addr = temp;
    return;
}
```

By making signature tables contain thunks we can implement the conformance check completely, i.e., we are not limited to strict conformance. There are no run-time penalties compared to the front-end implementation if a signature member function doesn't require conversions. On the contrary, by not having to test a `code` field as in our implementation, a few instructions will be saved. The only disadvantage of this solution is that it requires generation of low-level code, which complicates or even prohibits its use in a compiler that generates C code, such as AT&T's `cfront` compiler.

As in the front-end implementation, assigning a signature pointer to another signature pointer might require copying entries of the RHS signature table to the LHS signature table. In most cases we can 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 some further conversions.

In the thunk implementation described in [15], copying of signature table entries is avoided by having the `optr` field of the LHS signature pointer point to the RHS signature pointer instead of pointing to the object. This makes assignment more efficient but requires multiple indirections in a signature member function call. Furthermore, to allow assigning a local signature pointer to a non-local signature pointer the solution in [15] has to be corrected and signature pointers have to be allocated on the heap.

There is one more detail in assigning a signature pointer to another signature pointer. If the RHS signature table contains a default implementation that is not allowed to be copied to the LHS signature table, an error has to be reported. To allow this run-time test, we have to reintroduce a flag that tells us whether a default implementation is used or not. This flag can be stored in the low-order bit of the function/thunk pointer in the signature table. We just have to make sure that class member functions and thunks are aligned on half-word or word boundaries, which is required on most RISC-based architectures anyway. When calling a signature member function that might use a default implemention, we have to mask out this bit. If the architecture allows, we could omit the mask instruction by starting default implementations at odd addresses. The only time this flag needs to be tested is in the code for an assignment when performing the run-time error check.

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

6 Cost Comparison

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

The space needed for the signature table in the compiler front-end implementation is the same as the space needed for the virtual function table in the preprocessor implementation, two words for each signature member function and an additional one for the implicitly declared destructor. Additional space is needed in signature tables for constants. In the thunk implementation, the signature table takes only half the space since we only need one pointer per table entry. But in addition we need static storage for the thunks.

Assigning a class object to a signature pointer requires two pointer assignments in the compiler-based solutions. 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.

Assigning a signature pointer of a different type than the LHS signature pointer can become expensive in the compiler based implementations if signature table entry fields have to be copied. In the preprocessor implementation, we have the same cost as for assigning a class object.

In the preprocessor implementation, a signature member function call takes as much time as two class member function calls, one of which is virtual.

When calling a non-virtual member function in the front-end based implementation, we need one test in addition to the time needed for a virtual member function call. If the `optr` field of the signature pointer is in the wrong register, we also need a register-to-register move. 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 addition, there are one or two tests, depending on whether a default implementation might be called or not. The cost of calling a default implementation is two tests added to the cost of a virtual member function call.

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

To optimize calling a virtual class member function through a signature pointer, we could copy the information from the virtual function table into the signature table when the member function is called for the first time. This would add the cost of copying in the first call but would make subsequent calls more efficient. We could use this optimization for both compiler based implementations.

7 Conclusion

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

While we presented the ideas of such a language extension as an extension of C++, they would equally well apply to any statically typed object-oriented programming language. Having decoupled subtyping from inheritance, it would be possible to change the semantics of inheritance and make it more versatile for code reuse by allowing to inherit only parts of a superclass while giving up its use for subtyping. For pragmatical reasons, however, such a change is undesirable as it might affect the behavior of existing programs.

8 Availability

Parts of the language extension have already been implemented in GCC; the implementation is available by anonymous `ftp` from `ftp.cs.purdue.edu`, directory `pub/gb`. We expect our extension to become part of the GCC distribution starting with GCC version 2.6.

References

[1] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. "An object-oriented approach to algebra system design." In Bruce W. Char (ed.): *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC '86)*, Waterloo, Ontario, Canada, 21–23 July 1986, pp. 24–30. Association for Computing Machinery, 1986.

[2] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. "A Smalltalk system for algebraic manipulation." In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 277–283.

[3] Pierre America and Frank van der Linden. "A parallel object-oriented language with inheritance and subtyping." In *Proceedings of OOPSLA '90 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Ottawa, Canada, 21–25 October 1990. *SIGPLAN Notices*, Vol. 25, No. 10, October 1990, pp. 161–168.

[4] Gerald Baumgartner and Vincent F. Russo. *Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism.* To appear in *Software: Practice & Experience*, 1994.

[5] Gerald Baumgartner and Ryan D. Stansifer. *A Proposal to Study Type Systems for Computer Algebra.* RISC-Linz Report 90-87.0, Research Institute for Symbolic Computation, Linz, Austria, March 1990.

[6] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. "Object structure in the Emerald system." In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 78–86.

[7] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. "Interfaces for strongly-typed object-oriented programming." In *Proceedings of OOPSLA '89 Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans, Louisiana, 1–6 October 1989. *SIGPLAN Notices*, Vol. 24, No. 10, October 1989, pp. 457–467.

[8] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. "Modula-3 Language Definition." *ACM SIGPLAN Notices*, Vol. 27, No. 8, August 1992, pp. 15–43.

[9] William R. Cook. "Interfaces and specifications for the Smalltalk-80 collection classes." In *Proceedings of OOPSLA '92 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada, 18–22 October 1992. *SIGPLAN Notices*, Vol. 27, No. 10, October 1992, pp. 1–15.

[10] William R. Cook, Walter L. Hill, and Peter S. Canning. "Inheritance is not subtyping." In *Proceedings of 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, 17–19 January 1990, pp. 125–135.

[11] James Donahue and Alan Demers. "Data types are values." *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 426–445.

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

[13] J.A. Goguen and T. Winkler. *Introducing OBJ3.* Technical Report CSL-88–9, SRI International, 1988.

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

[15] Elana D. Granston and Vincent F. Russo. "Signature-based polymorphism for C++." In *Proceedings of USENIX C++ Technical Conference*, Washington, D.C., 1991.

[16] Paul Hudak et al. "Report on the programming Language Haskell: A non-strict, purely functional language, version 1.2." *ACM SIGPLAN Notices*, Vol. 27, No. 5, May 1992, Section R.

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

[18] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. "An exemplar based Smalltalk." In *Proceedings of OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 322–330.

[19] David B. MacQueen. "Modules for Standard ML." *Polymorphism*, Vol. 2, No. 2, 1985.

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

[21] Craig Schaffert et al. "An introduction to Trellis/Owl." In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September–2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 9–16.

[22] Alan Snyder. "Encapsulation and inheritance in object-oriented programming languages." In *Proceedings of OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 38–45.

[23] Richard M. Stallman. *Using and Porting GNU CC.* Cambridge, Massachusetts: Free Software Foundation, V. 2.3, 16 December 1992.

[24] Robert S. Sutor and Richard D. Jenks. "The type inference and coercion facilities in the Scratchpad II interpreter." In *Proceedings of SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, 24–26 June 1987, St. Paul, Minnesota. *SIGPLAN Notices*, Vol. 22, No. 7, 1987, pp. 56–63.

[25] Stephen M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. "The Scratchpad II Type System: Domains and Subdomains." In Alfonso M. Miola (ed.): *Computing Tools for Scientific Problem Solving.* London: Academic Press, 1990, pp. 63–82.

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