Chapter 3 Previous Work

In this chapter we describe previous software solutions for optimizing the execution speed of object-oriented languages. We also review some techniques originally developed for traditional non-object-oriented languages that relate to techniques for optimizing SELF.

3.1 Smalltalk Systems

Of all the commercial languages, Smalltalk is the closest to SELF. Consequently, efforts to improve the performance of Smalltalk programs are probably the most relevant to achieving good performance for SELF.

3.1.1 The Smalltalk-80 Language

Smalltalk-80 incorporates most of the language features we identified in the previous chapter as contributing to expressive power and implementation inefficiency: abstract data types, message passing, inheritance, dynamic typing, user-defined control structures, error-checking primitives, and generic arithmetic [GR83]. However, the designers of Smalltalk-80 included several compromises in the definition of the language and the implementation to make Smalltalk easier to implement efficiently.

Smalltalk (and most object-oriented languages) treats variables differently from other methods. Variables are accessed directly using a special mechanism that avoids a costly message send. Unfortunately, this run-time benefit comes at a significant cost: a subclass can no longer override a superclass' instance variable with a method, nor vice versa. This restriction prevents certain kinds of code reuse, such as a class inheriting most of the code of a superclass but changing part of the representation. A canonical example, described in section 2.2.3, is a programmer who wants to define a **`rectangle`** class as a subclass of the general **`polygon`** class but is stymied when he cannot provide a specialized representation for rectangles that differs from that used for polygons.

Another less visible compromise sacrifices the purity of the object-oriented model by introducing built-in control structures and operations. The definitions of many common methods are "hard-wired" into the compiler, including integer arithmetic methods such as **`+`** and **`<`**, the object equality method **`==`**, boolean methods such as **`ifTrue:`**, and block iteration methods such as **`whileTrue:`**. By restricting the semantics and flexibility of these "messages," the Smalltalk compiler can implement them efficiently using low-level sequences of special byte codes with no message sending overhead.

These compromises significantly improve run-time performance but sacrifice some of the purity and flexibility of the language model. For one, programmers are no longer able to change the definitions of the hard-wired methods, such as to extend or instrument them. The compiler assumes that there is only a single system-wide definition of **`==`**, for example, and any new definitions added by programmers are ignored. Programmers cannot define their own identity methods for their new object classes. Similarly, for messages like **`ifTrue:ifFalse:`** with block literals as arguments, the compiler assumes that the receiver will either be **`true`** or **`false`**, and no other object is allowed as the receiver, even if the programmer provides an implementation of **`ifTrue:ifFalse:`** for that object. Clearly these restrictions compromise the simple elegance and extensibility of the language, and programmers can be inflicted with completely unexpected behavior if they violate any of these assumptions.

Even worse, since there is such a large difference in performance between the handful of control structures hard-wired into the implementation and the remaining control structures written by the programmer, programmers are tempted to use the faster built-in control structures even if they are inappropriate. If succumbed to, this temptation will lead to programs that are less abstract, less malleable, and less reusable. A better solution would be to develop techniques that improve the performance of *all* user-defined control structures uniformly, including those written by the programmer, and encourage the programmer to maintain a high level of abstraction.

3.1.2 Deutsch-Schiffman Smalltalk-80 System

The definition of Smalltalk-80 specifies that source code methods are translated into *byte codes*, the machine instructions of a stack machine. Originally, Smalltalk-80 ran on Xerox Dorados implementing this instruction set in microcode [Deu83]. Subsequent software implementations of Smalltalk-80 on stock hardware supplied a *virtual machine* that interpreted these byte codes in software. Needless to say this interpretation was quite slow [Kra83].

Additionally, Smalltalk-80 activation records are defined and implemented as first-class objects, allocated in the heap and garbage collected when no longer referenced. This design further slowed the implementation of method call and return with the cost to allocate and eventually deallocate these activation record objects.

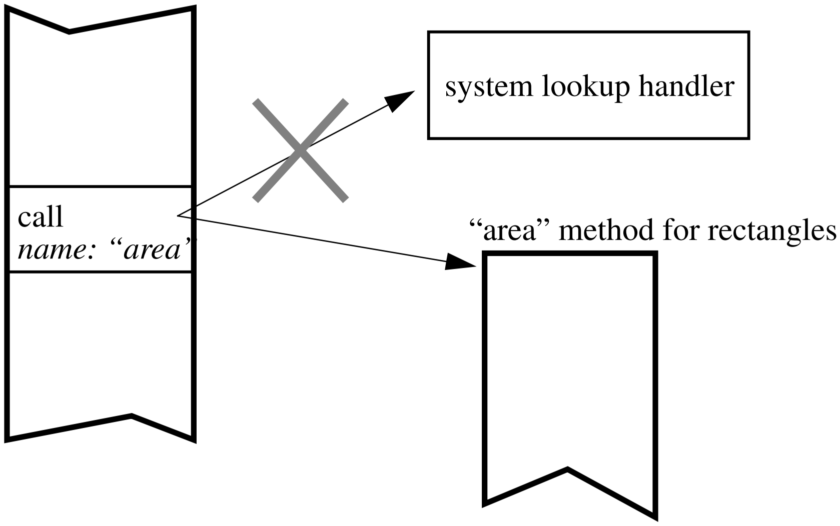
Peter Deutsch and Allan Schiffman developed several techniques for implementing Smalltalk-80 programs better than these interpreters without specialized hardware support [DS84]. They overcame the interpretation overhead by introducing an extra invisible translation step from virtual machine byte codes into native machine code, with the system directly executing the native machine code instead of interpreting the original byte codes. This translation primarily eliminates the overhead in the interpreter for decoding each byte code and dispatching to an appropriate handler.

The Deutsch-Schiffman system is based on *dynamic compilation* (called *dynamic translation* by Deutsch and Schiffman): byte codes are translated into machine code on demand at run-time rather than at compile-time. Dynamic compilation has a number of advantages over traditional static batch compilation. The compiler only has to compile code that actually gets used, reducing the total time to compile and run a program. Programming turn-around is minimized since the program can begin execution immediately after a programming change without waiting for the compiler to recompile all changed code; any recompilations will be deferred until needed at run-time.

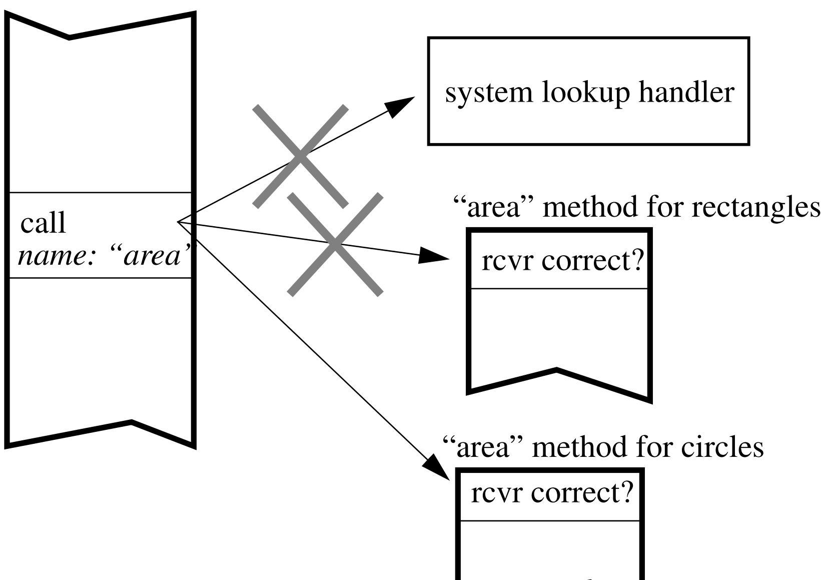
With dynamic compilation, the compiled code can be treated as a *cache* on the more compact byte-coded representation of programs. If a compiled method has not been used recently, its code can be thrown away (flushed from the cache) to save compiled code space. If a method is needed again later, it can simply be recompiled from the byte-coded representation. Deutsch and Schiffman's compiler is fast enough that recompiling a method from its byte code form is faster than paging in its compiled code from the disk, so this caching technique is especially useful for machines with only small amounts of main memory.

The Deutsch-Schiffman system optimizes method call and return by initially stack-allocating activation records. If the program begins manipulating stack-allocated activation records as first-class objects (e.g., by running the debugger) or if external references to the activation record still exist when the activation record is about to return (e.g., when a nested block outlives its lexically-enclosing activation record), then the system "promotes" activation records from the stack to the heap, preserving the illusion that activation records were heap-allocated all along. Most activation records never get promoted to the heap, so the performance of method call and return approaches the performance of procedure call and return in a traditional language implementation and the load on the garbage collector is greatly reduced.

To reduce the cost of dynamic binding, Deutsch and Schiffman introduce a technique called *in-line caching*. They observe that for many message send call sites, the class of the receiver of the message remains the same from one call to the next. This is symptomatic of *monomorphic code*: code in which the polymorphism afforded by dynamic typing and dynamic binding is not being actively used. To take advantage of this situation, the call instruction that originally invoked the run-time message lookup system is overwritten or *backpatched* with a call instruction that invokes the result of the lookup. This effectively replaces the dynamically-bound call with a statically-bound call, eliminating the overhead of the run-time message lookup system.



Of course, this static binding is not necessarily always correct. The class of the receiver of the message might change, in which case the result of the message lookup might be different. To handle this situation, the Deutsch-Schiffman system prepends a *prologue* to each method's compiled code that first checks to see if the class of the receiver is correct for this method. If the class is not correct, then the normal run-time message lookup system is invoked, backing out of the mistakenly-called method. Thus the backpatched call instruction acts like a cache, one entry big, of the most recently called methods. If the hit rate is high enough and the cost of checking for a cache hit is low enough, overall system performance is improved.



Since a method may be inherited by more than one receiver class, whether or not the receiver's class is correct for the cached method may be an expensive computation to perform. Deutsch and Schiffman solve this problem by storing the receiver's class in a data word after the call instruction when the instruction is backpatched. The method's prologue then simply compares the current receiver's class with the one stored in the word after the call site (reachable using the return address for the call). If they are the same, the static binding is still correct and the method is executed. If they are different, the prologue calls the run-time lookup routine to locate the method for the new receiver class. This approach has a lower hit rate than is possible (since a different class might still invoke the same method), but allows a relatively fast check for a cache hit. Depending on the memory system organization, this technique requires two or three memory references and four or five extra instructions to verify an in-line cache hit.

These techniques, along with a faster garbage collection strategy called deferred reference counting [DB76], made significant improvements in the performance of Smalltalk-80 systems on stock hardware. The run-time performance of the Deutsch-Schiffman implementation is close to twice the speed of the interpreted version of Smalltalk. However, even with these techniques, plus the compromises in the language definition for commonly-used operations and control structures, the performance of Smalltalk-80 programs is still markedly slower than implementations of traditional statically-typed non-object-oriented languages. As described in Chapter 14, the current performance of the Deutsch-Schiffman implementation of Smalltalk-80 on a set of small benchmarks is roughly ten times slower than optimized C, measured on a Sun-4/260 workstation. This order of magnitude performance difference is unacceptable to many programmers and is certainly one of the major reasons that Smalltalk is not more widely used.

Deutsch and Schiffman's techniques are completely transparent to the user. Both dynamic translation of byte codes to native code and stack allocation of activation records are performance optimizations hidden from the user; Smalltalk programmers think they are simply getting a better interpreter. No user intervention is required to invoke the compiler or any optimizations, and the user's programming model remains at the level of interpreting, and debugging, the source code. The speed of the translation from byte codes to machine code is so fast that it is hard to notice any pauses for compilation at all. Future systems should attempt to achieve this level of unobtrusiveness.

3.1.3 Typed Smalltalk and TS Optimizing Compiler

Many researchers have noted that the chief obstacle to improving the performance of Smalltalk programs is the lack of representation-level type information upon which to base optimizations like procedure inlining [Joh87]. In an effort to improve the speed of Smalltalk programs, Ralph Johnson and his group at the University of Illinois at Urbana-Champaign have designed an extension to Smalltalk called Typed Smalltalk [Joh86, JGZ88, McC89, Hei90, Gra89, GJ90]. They added explicit type declarations to Smalltalk and built an optimizing compiler, called TS, that uses these type declarations to improve run-time performance.

A type in Typed Smalltalk is either a (possibly singleton) set of classes* or a *signature*. A variable declared to be of a set-of-classes type is guaranteed to contain instances of only those classes included in the set. To allow a variable to contain an instance of a subclass of one the listed classes, the subclass must also appear explicitly in the list. A signature type is more abstract, listing the set of messages that may legally be sent to variables of that type. Any object that understands the required messages can be stored in a variable declared with a signature type, independent of its implementation. A signature type can be converted into a set-of-classes type by replacing the signature with all classes compatible with the signature; this translation depends on the particular definition of the class hierarchy and may change if the class hierarchy is altered.

Both kinds of types are used to perform static type checking of Typed Smalltalk programs, but the TS optimizing compiler exploits set-of-classes types in several ways to improve run-time performance. If a variable is declared to contain instances of only a single class (the set of classes is a singleton set), the compiler can statically bind every message sent to the contents of the variable to the corresponding target methods. Similarly, if a primitive operation expects arguments of a particular class, this check may be performed at compile-time instead of run-time if the type of the argument is a singleton set. Methods that the user has marked as "inlinable" may be inlined if they are invoked from a statically-bound call site.

If the set of classes is not a singleton set but is still small (say, two or three members in the set), the TS compiler performs *case analysis*. The compiler generates type testing code to "case" on the class of the receiver at run-time, branching to one of several sections of code, one section for each member of the set of classes in the type declaration. The exact class of the receiver is known statically in each arm of the case, allowing the message to be statically bound and inlined (if the user has marked the target method "inlinable"). This case analysis usually takes more compiled code space than the original message send, since several potentially inlined versions of the message send are compiled, but the run-time performance of the message is improved, especially if the message is inlined in the case arms. Control flow rejoins after each arm of the case.

* Some class types in Typed Smalltalk can be parameterized by other classes, e.g., **`Array of: Integer`**.

Case analysis becomes too expensive in compiled code space and possibly even in execution time once the number of possible classes becomes large. The TS compiler falls back on the Deutsch-Schiffman technique of in-line caching beyond three or so possible classes. In-line caching is also used for messages sent to objects of signature types, since these types tend to match many different classes.

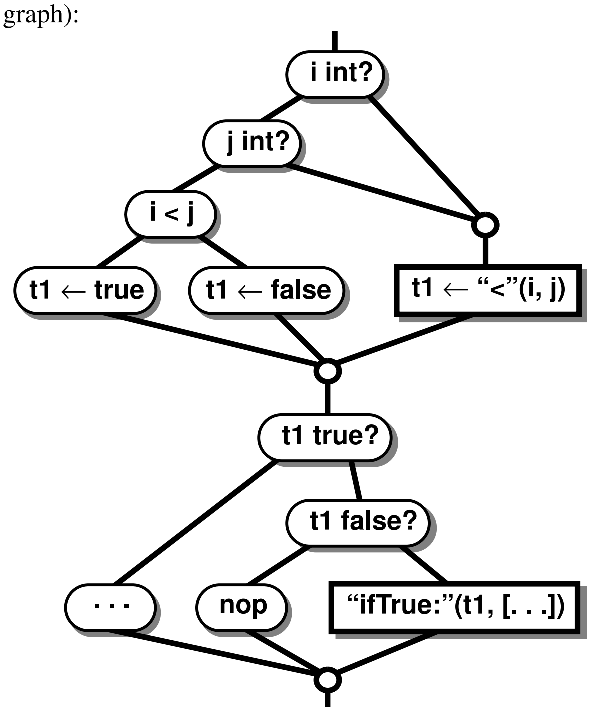
To reduce the burden on the programmer of specifying static type declarations, the Typed Smalltalk system includes a *type inferencer* that, when invoked, can automatically infer the appropriate type declarations for a routine. The programmer must provide type declarations for instance variables and class variables (although Justin Graver states in his dissertation that even these type declarations are not strictly necessary), and the inferencer will compute the appropriate type declarations for method arguments, results, and locals.

Unfortunately, this inferencing process is complicated by the more dynamic nature of control flow in languages with dynamic binding of messages to methods and by existing Smalltalk programming practice which virtually requires flow-sensitive type checking. To handle these problems, the Typed Smalltalk type checker uses *abstract interpretation* of top-level expressions to combine the inferred method signatures together based on the control flow required to evaluate the top-level expression. While quite powerful, this style of type checking can be very slow (since in the worst case their abstract interpretation takes time exponential in the size of the Smalltalk system), and the whole process may need to be repeated for each new top-level expression. Also, since the system infers signature types rather than set-of-classes types, the new type declarations are not very useful for the TS optimizing compiler. Manual type declarations are still the rule in Typed Smalltalk when it comes to improving run-time performance.

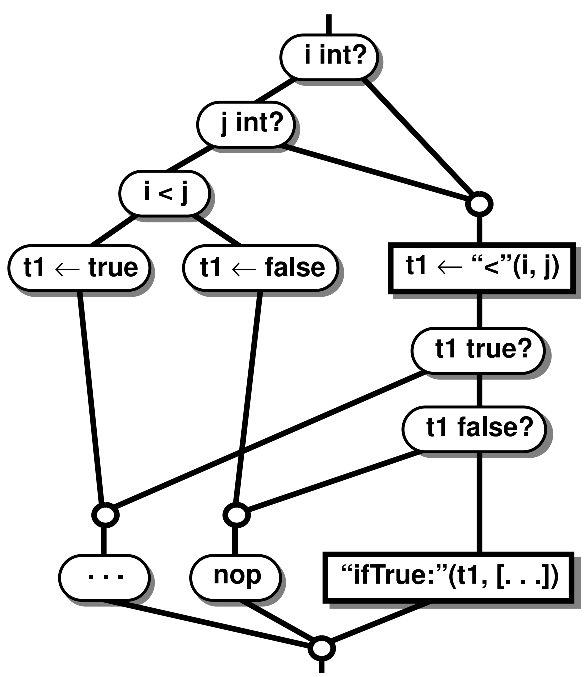
In the TS compiler, the front-end translates the Typed Smalltalk code into a relatively machine-independent *register-transfer language* (RTL); the back-end then optimizes these RTL instructions and converts them into native machine code. Primitive operations may be written by the programmer in the same register-transfer language, supporting a more user-extensible system and allowing inlining of calls to primitive operations using the same mechanisms as inlining of statically-bound message sends. Many standard optimizations are performed by the back-end of the compiler, including common subexpression elimination, copy propagation, dead assignment elimination, dead code elimination, and various peephole optimizations.

One interesting extension of constant folding included in the TS back-end is called *constant conditional elimination*; this optimization is related to our *splitting* technique described in Chapter 10. In conventional constant folding of conditionals, when a conditional expression can be evaluated to either **`true`** or **`false`** at compile-time, the compiler can eliminate the test and either the true or the false branch. The TS extension also handles cases where the conditional expression is not a compile-time constant within the basic block containing the test, but where the conditional expression is compile-time evaluable in some of the block's predecessors. In this situation, an algorithm called *rerouting predecessors* copies the basic block containing the test for each predecessor that can evaluate the conditional expression at compile-time, redirecting the predecessor to flow into the copy. The test can then be eliminated from the copied basic block, and perhaps from the original basic block, too, now that it has fewer predecessors.

Opportunities for this optimization occur frequently in Smalltalk code (and in SELF code, too). One common sequence in Smalltalk and SELF is something of the form `i < j ifTrue: [ ... ]`. This is Smalltalk's and SELF's version of a standard `if` conditional expression. Straightforward Smalltalk compilers would generate the following code (displayed as a control flow graph):



This sequence of code first loads either the `true` object or the `false` object (in the common case that the arguments are integers) and then immediately tests for the `true` object and the `false` object. Neither test can be eliminated directly, since the result of the `<` message is not a single constant. But after rerouting predecessors and copying the tests for each of the possible `<` outcomes, the tests can be turned into constant conditionals for the first two predecessors and eliminated:



Subsequently, the loads of `true` and `false` may be eliminated as dead assignments, leaving code that is comparable in quality to optimizing C compilers, once the two initial type tests have been executed.

Published performance results for an early version of the TS optimizing compiler indicated that adding explicit type declarations (of the set-of-classes variety) and implementing the optimizations described here led to a performance improvement of 5 to 10 times over the Tektronix Smalltalk interpreter for small examples on a Tektronix 4405 68020-based workstation. Rough calculations based on the speed of the Deutsch-Schiffman Smalltalk implementation on a similar machine indicate that the TS optimizing compiler runs Typed Smalltalk programs about twice as fast as the Deutsch-Schiffman system runs comparable untyped Smalltalk-80 programs. Unfortunately, this is still somewhere between 3 and 5 times slower than optimized C.

The implementation of Typed Smalltalk is not complete. One serious limitation is that support for generic arithmetic has been disabled: no overflow checking is performed on limited-precision integers (instances of **`SmallInteger`** in Smalltalk), and so the type checker assumes that the result of a limited-precision integer arithmetic operation is another limited-precision integer. Additionally, in-line caching is not currently implemented, and so message sends in which the type of the receiver may be one of more than a couple of classes cannot be executed by their system (although they can be type-checked). Finally, only a small amount of the Smalltalk system has been converted into Typed Smalltalk by adding type declarations, and so only small benchmarks may be executed; a full test of the type system and the implementation techniques has not been performed.

One disadvantage of the Typed Smalltalk approach is that users must add many type declarations to programs that work fine without them. To see real performance improvements, these type declarations must be of the set-of-classes variety, and users will be tempted to make the set as small as possible to achieve the best speed-ups, limiting the reusability of the code. In addition, users need to annotate all small, commonly-used methods as “inlinable” so that the TS compiler will inline calls to them.

Another disadvantage of the Typed Smalltalk system is that it compiles slowly. This disrupts the user’s illusion of sitting at a Smalltalk interpreter. Johnson notes that the implementation has not been fully tuned for compile-time performance yet, nor has it been annotated with type declarations and optimized with itself yet. Johnson speculates that these improvements might make the TS compiler run as fast as the Deutsch-Schiffman compiler; however, we are skeptical that an optimizing compiler written in Typed Smalltalk could ever run as fast as a simple compiler written in C, as is the Deutsch-Schiffman compiler.

3.1.4 Atkinson’s Hurricane Compiler

Robert Atkinson pursued an approach similar to the Typed Smalltalk project in attempting to speed Smalltalk-80 programs [Atk86]. He devised a type system very similar to Typed Smalltalk’s set-of-classes types and allowed Smalltalk programmers to annotate their programs with type declarations. He designed and partially implemented an optimizing compiler, called Hurricane, that uses these types in exactly the same way as the TS optimizing compiler: the compiler would statically bind and inline messages sent to receivers of singleton types, and statically bind and inline messages sent to receivers of small-set types after casing on the type of the receiver (the latter of these techniques was designed but not implemented).

Unlike the Typed Smalltalk approach, type declarations in Hurricane are *hints*, not guarantees, and no static type checking is performed to verify that type declarations are correct. This requires that all optimizations based on type declarations be prefixed with a run-time test to verify that the type declaration provided by the programmer is correct. In case the declaration is ever incorrect, the Hurricane compiler generates code to restart an unoptimized, untyped version of the method. Atkinson notes that this restarting severely limits the kinds of methods that can be optimized, presumably to just those that have no side-effects before any code that verifies the types of variables.

Few optimizations other than inlining statically-bound messages were implemented, and in fact some optimizations present in the Deutsch-Schiffman compiler, including inlining calls to common primitives such as integer arithmetic, were omitted. Even with these limitations in Hurricane’s implementation, Atkinson reports a factor of two speed-up over the Deutsch-Schiffman system for small examples running on a Sun-3 workstation. This speed-up appears to be similar to that obtained by the TS optimizing compiler, although precise comparisons are difficult since the two groups use different machines and compare against different baseline systems (a Deutsch-Schiffman dynamic compilation system versus a Tektronix interpreter). No information is available about the speed of the compiler itself. It is interesting to note that Atkinson implemented his Hurricane compiler in a single summer.

3.1.5 Summary

The Deutsch-Schiffman Smalltalk-80 system, with its dynamic compilation, in-line caching, and other techniques, represented the state of the art when we began our work in implementing dynamically-typed purely object-oriented languages. Unfortunately, even with several serious compromises in the purity of the language and the programming environment for the sake of better performance, the Deutsch-Schiffman Smalltalk-80 system runs small benchmark programs at just a tenth the speed of a traditional language such as C when compiled using an optimizing compiler. Attempts to boost the performance of Smalltalk on stock hardware rely on explicit user-supplied type declarations to provide more information to the compiler that enables it to statically-bind and inline away many expensive messages. This approach typically doubles the speed of small benchmark programs, but still leaves a sizable performance gap between type-annotated Smalltalk and a traditional language and sacrifices much of the ease of programming, flexibility, and reusability of the original untyped code.

3.2 Statically-Typed Object-Oriented Languages

Several statically-typed object-oriented languages have been implemented, and in this section we describe techniques used in these languages to achieve relatively good performance.

3.2.1 C++

C++ is a statically-typed class-based object-oriented language [Str86, ES90]. The original version of C++ included only single inheritance; recent versions (C++ 2.0 and later) have extended C++ to support multiple inheritance. C++ contains C as an embedded sublanguage and so incorporates all of C's built-in control structures and data types. Neither user-defined control structures nor generic arithmetic are supported directly. Operations on built-in data types are checked at compile-time for type correctness, but other checks, such as array access bound checks, are not, so the built-in operations in C++ are not robust. Current versions of C++ do not support parameterized data structures or exceptions, but future versions of the language probably will. Statically-bound procedure calls are available even when performing operations on objects; messages are dynamically-bound only when the target method is annotated with the **`virtual`** keyword. These language properties enable C++ programmers to reduce the performance penalty of object-oriented programming by skirting features that incur additional cost, such as dynamic binding. Of course, the benefits of object-oriented programming also are lost when non-object-oriented alternatives are selected.

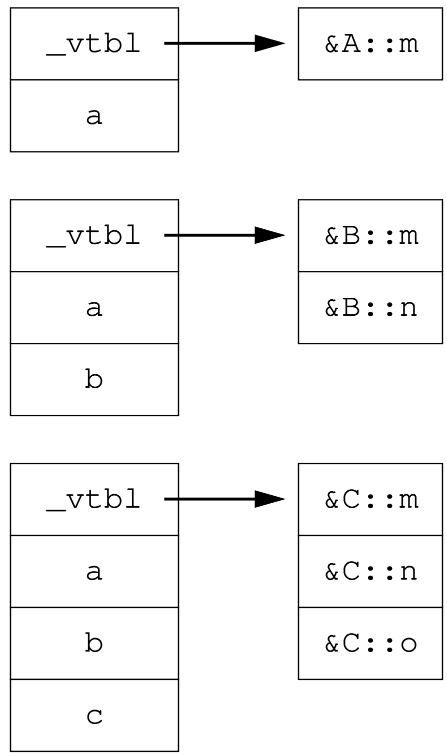
C++ equates types with classes. A subclass may specify whether or not it is to be considered a subtype of its superclass(es) (by inheriting from the superclass either **`public`**ly or **`private`**ly); if so, the C++ compiler verifies that the subclass is a legal subtype of the superclass(es). With only single inheritance, this static type system severely hampers reusability of code, since instances of two classes unrelated in the inheritance hierarchy cannot be manipulated by common code, even if both classes provide correct implementations of all operations required by the code. This deficiency is rectified with the addition of multiple inheritance, since the two previously unrelated classes may be extended with an additional common parent defining virtual functions for all the operations required by the common code, thus relating the two classes.

Most C++ implementations incorporate a special technique to speed dynamically-bound message passing enabled by the presence of static type checking. In the versions of C++ supporting only single inheritance, each object contains an extra data field that points to a class-specific array of function addresses (the **`_vtbl`** array). Each method that is declared **`virtual`** is given an index, in increasing order from base class to subclass; all overriding implementations are given the same index as the method they override. **`_vtbl`** arrays for each class are initialized to contain the addresses of the appropriate method, each method's address at its corresponding index in the **`_vtbl`** array.

```
struct A {
 int a;
 virtual int m();
};
```

```
struct B: A {
 int b;
 virtual int m();
 virtual int n();
};
```

```
struct C: B {
 int c;
 virtual int m();
 virtual int n();
 virtual int o();
};
```



C++ Virtual Function Implementation
(Single Inheritance Version)

To implement a dynamically-bound message send, the compiler generates a sequence of instructions that first loads the address of the receiver's **`_vtbl`** array, then loads the function address out of the **`_vtbl`** array at the index associated with the message being sent, and then jumps to this function address to call the method. Thus dynamically-bound calls have an overhead of only two memory indirections per call. Of course, additional overhead arises from the fact that dynamically-bound calls are not inlined, while a statically-bound call could be.

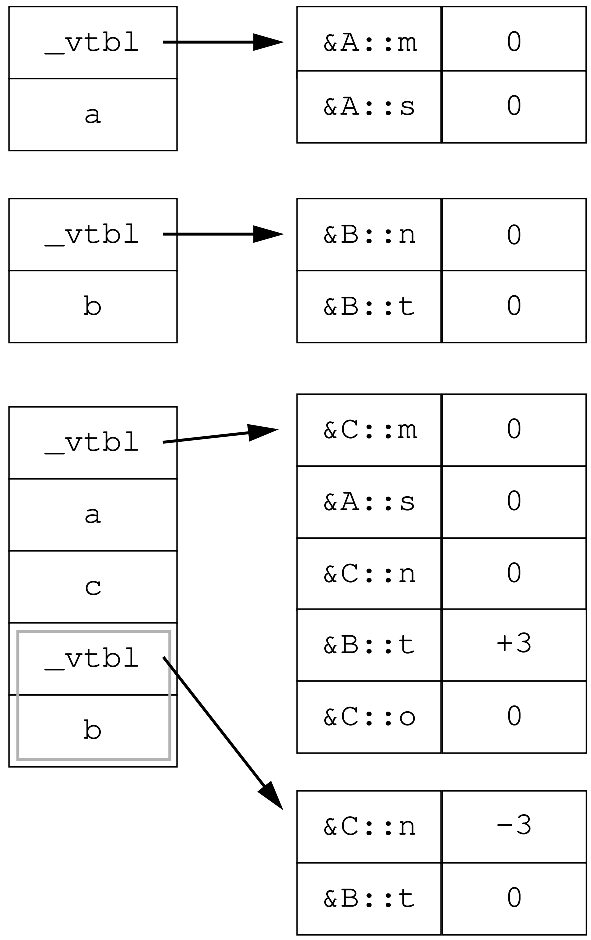
This **`_vtbl`** technique may be extended to work with the versions of C++ that support multiple inheritance. The trick is to embed in the object a complete representation, including the **`_vtbl`** pointer, of each superclass other than the first, and to pass the address of the embedded object whenever assigning a reference to the object to a variable that expects an object of the superclass (or when casting an expression from the subclass type to the superclass type). Users of an object reference do not know whether the object they manipulate is a "real" object, or if they are manipulating an embedded object instead.

Since this assignment goes on implicitly when invoking a virtual function defined on one of the object's superclasses (assigning the receiver of the message to the **this** argument of the virtual function), the **_vtbl** array and the implementation of message dispatch need to be extended. The **_vtbl** array is extended to be an array of <function address, embedded object offset> pairs.

```
struct A {
 int a;
 virtual int m();
 virtual int s();
};

struct B {
 int b;
 virtual int n();
 virtual int t();
};

struct C: A, B {
 int c;
 virtual int m();
 virtual int n();
 virtual int o();
};
```



C++ Virtual Function Implementation
(Multiple Inheritance Version)

The function address works the same as for single inheritance. The offset is added to the address of the receiver object to change it into a pointer into the embedded subobject if necessary. Thus, the overhead for invoking a dynamically-bound method is three memory indirections and an addition (again ignoring the overhead for not being able to statically-bind and inline away the message send entirely). This is roughly the same overhead as for the inline caching technique used in the fast Smalltalk-80 implementations when the inline cache gets a hit, but the **_vtbl** array technique does not slow down for message sends in which the receiver's class changes from one send to the next (there is no extra cache miss cost). In the multiple inheritance case, the **_vtbl** array technique requires additional add operations for some assignments from a subclass to a superclass.

Supporting *virtual base classes* in C++ adds even more complication. Since multiple inheritance in C++ can form a directed acyclic graph, a particular base class can be inherited by some derived class along more than one derivation path. If the base class is declared **virtual**, then only one copy of the virtual base class' instance variables is to exist in the final object; non-virtual base classes have independent copies of instance variables along each distinct derivation path. Since the embedding approach does not work directly with virtual base classes (in general, only one of the derivation paths can have the virtual base class embedded within it), C++ implementations include a pointer to the virtual base class in the representation of every subclass of the virtual base class, as shown in the diagram on page 24.

Assignments of the subclass to the virtual base class (or casts from the subclass to the base class) do not simply add a constant to the variable's address but instead must perform a memory indirection to load the virtual base class' address out of the object, increasing the overhead for these sorts of assignments from an add instruction to a memory indirection.

Virtual base classes should not be written off as an obscure C++-specific feature. In fact, most other class-based object-oriented languages define multiple inheritance of instance variables as if all superclasses are virtual base classes. Consequently, the extra overhead associated with virtual base classes in C++ also would be incurred by other object-oriented languages with multiple inheritance.

The space overhead for the **`_vtbl`**-based virtual function implementation includes the space taken up for the **`_vtbl`** function arrays and the extra words in each object to point to the **`_vtbl`** arrays and to virtual base classes. The **`_vtbl`** function arrays may be shared by all instances of a class, but not with instances of subclasses. With the single inheritance scheme, there is a single **`_vtbl`** array per class, of length equal to the number of virtual functions defined by (or inherited by) the class; subclasses thus may have longer **`_vtbl`** arrays than superclasses. With multiple inheritance, a particular class must define **`_vtbl`** arrays for itself and every superclass that isn't the first superclass in a class' list of superclasses; each of these **`_vtbl`** arrays are twice as big as in the single inheritance version since they include offsets in addition to function addresses. The space overhead for each instance includes a pointer to each of its class' **`_vtbl`** arrays (one in the single inheritance case, possibly more in the multiple inheritance case) and a pointer to a virtual base class for each subclass of all virtual base classes in the object. This space overhead is significantly more than the single extra word per object required for the in-line caching technique used in the fast Smalltalk-80 implementations.

This **`_vtbl`**-based implementation of message passing may work for C++, but could pose problems for other languages that need to support garbage collection (C++ does not support garbage collection). C++'s multiple inheritance layout scheme can create pointers into the middle of an object (e.g., to point to an embedded superclass or virtual base class), but without any easy way of locating the outermost non-embedded object. This can be problematic for some fast garbage collection algorithms. Slowing down the garbage collector could offset some (or perhaps all) of the extra performance advantage of this implementation approach over a simpler scheme such as in-line caching that never produces pointers into the middle of an object.

3.2.2 Other Fast Dispatch Mechanisms

John Rose describes a framework for analyzing **`_vtbl`**-array-style message send implementations [Ros88]. His framework can describe several variations on the array-lookup implementation, and Rose carefully analyzes their relative performance. Not all variations are practical for all object-oriented languages; techniques such as that implemented in single-inheritance C++ are probably the fastest available short of statically binding and optionally inlining the message send.

Several researchers have proposed techniques for using a single **`_vtbl`** array even in languages with multiple inheritance (the implementation of multiple-inheritance C++ uses multiple **`_vtbl`** arrays per object and pointer adjusting for assignments and method calls). These proposed techniques potentially waste some space by having entries in **`_vtbl`** arrays that are unused for some classes, but only need a single **`_vtbl`** array pointer per object and a single **`_vtbl`** array per class. The central idea behind these techniques is to determine a system-wide mapping of message names to **`_vtbl`** array indices such that no two message names defined for the same object map to the same index. Dixon *et al* use a graph coloring technique to determine message names which must be given distinct indices [DMSV89]; they report that only a small amount of space is wasted in empty **`_vtbl`** array entries. Pugh and Weddell propose a novel extension that allows *negative* indices [PW90]. The extra degree of freedom in assigning indices without wasting space saves a significant amount of space. They report only 6% wasted space for a Flavors system with 564 classes and 2245 fields using their technique, versus 47% wasted space for a more conventional index assignment algorithm. Although Pugh and Weddell's technique is couched in terms of laying out the instance variables of an object to allow field accesses with a single memory indirection, their approach easily could be applied to laying out the entries in a class-specific array of member function addresses to allow virtual function calls with only two extra memory indirections above a normal direct procedure call, the same as for single-inheritance C++ implementations.

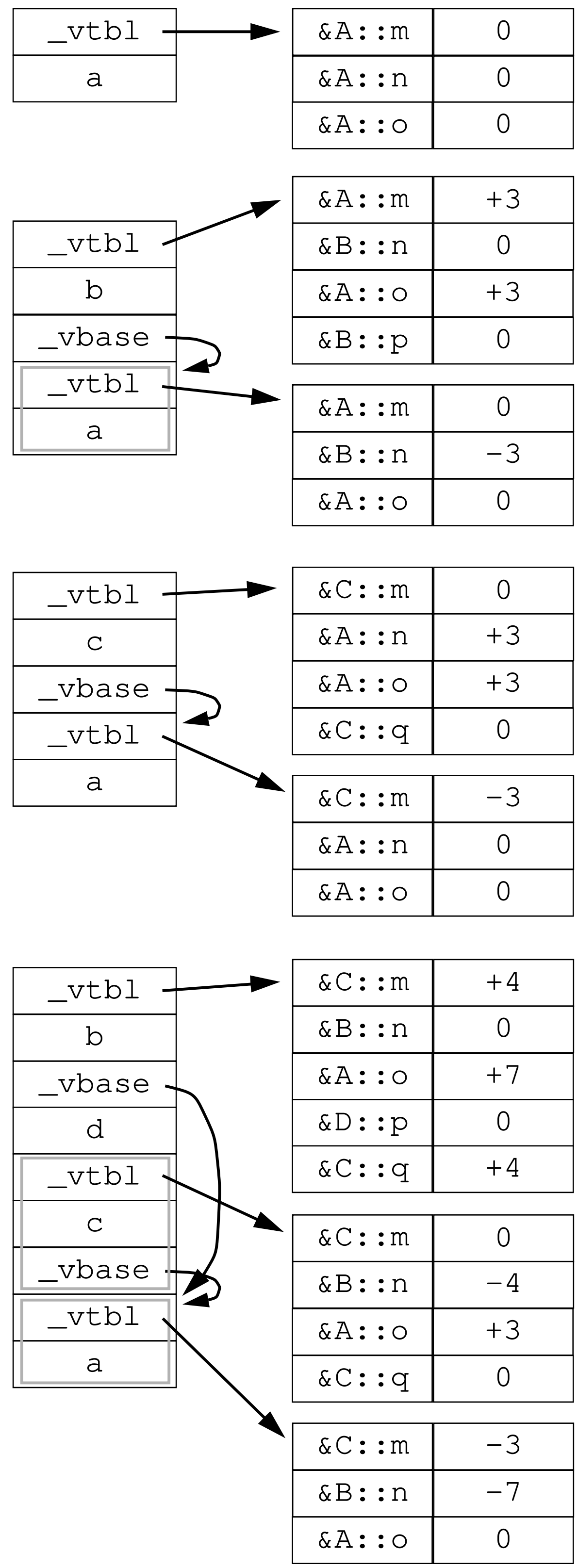
Unfortunately, the usefulness of these techniques is limited by their need to examine the complete class hierarchy prior to assigning indices to message names and compiling code. Additionally, adding a new class may require the system-

```
struct A {
 int a;
 virtual int m();
 virtual int n();
 virtual int o();
};
```

```
struct B: virtual A {
 int b;
 virtual int n();
 virtual int p();
};
```

```
struct C: virtual A {
 int c;
 virtual int m();
 virtual int q();
};
```

```
struct D: virtual A, B, C {
 int d;
 virtual int p();
};
```



C++ Virtual Function Implementation
(Multiple Inheritance with Virtual Base Classes)

wide assignment to be recalculated from scratch, since new conflicts may be introduced that invalidate the previous assignment. Dixon *et al* suggest a scheme whereby the index assignments are not treated as constants by the compiler but instead as global variables defined in a separately compiled module, thus limiting recompilation overhead to just the separate module that defines the assignments. However, this scheme slows down message lookup by at least an extra memory indirection to load the appropriate "constant" from its global variable location, thus largely eliminating the performance advantage of the global index allocation techniques.

The basic **`_vtbl`** array approach used in C++ implementations and the faster versions described above all rely on static type checking to guarantee that messages will not be sent to objects that do not expect them. The techniques may be extended to work with dynamically-typed languages by performing array bounds checking before fetching a function pointer out of a function array and also checking the actual message name against the name expected by the function extracted out of the array. If either of these tests fails, then the message is not understood by the receiver object, and a run-time type error results. The extra cost for these run-time checks may be significant, and when combined with the original cost of the two to four additional memory indirections may be much more than the expected average cost of in-line caching.

3.2.3 Trellis/Owl

Trellis/Owl is a statically-typed class-based object-oriented language supporting multiple inheritance [SCW85, SCB+86]. Trellis/Owl equates types with classes, and a subclass is required to be a legal subtype of its superclasses. Trellis/Owl includes the conventional kinds of built-in control structures, plus a **`type_case`** control structure that tests the run-time type of an expression. Trellis/Owl does not support user-defined control structures or closures, but does provide *iterators* and *exceptions*.* An iterator is a user-defined procedure invoked by the **`for`** built-in control structure. Unlike a normal procedure, an iterator may successively yield a series of values before returning. Each yielded value is assigned to the iteration variable local to the **`for`** loop, and the body of the **`for`** loop is executed once per yielded value. Control alternates between the iterator procedure and the **`for`**-loop body; the **`for`** loop exits when the iterator returns normally. Iterators provide a way for user-defined abstract data types (such as collections) to be iterated through conveniently, one element at a time, while preserving the abstraction boundary between the caller and the abstract data type. Exceptions provide a way for procedures to signal that a non-standard situation has occurred, and for callers to handle the exceptional situation at a point separate from the normal return point. Exceptions can help organize programs, streamlining the code for handling the normal common-case situations and clearly distinguishing the code handling unusual cases.

The **`type_case`** control structures, iterators, and exceptions are three common uses for closures. By building them into the language, the implementation of Trellis/Owl can include special techniques to make them relatively efficient. However, even though **`type_case`**'s, iterators, and exceptions extend traditional control structures in useful ways, there are still user-defined control structures that cannot be expressed easily using these built-in control structures. For example, exceptions as defined in Trellis/Owl automatically terminate the routine that raises the exception. Exceptions implemented using general closures could either terminate the routine, restart it, or continue it, at the discretion of the caller. As another example of the limitations of built-in control structures, Trellis/Owl's iterators provide a mechanism for performing some action uniformly on every element of a collection, but a user-defined control structure could treat the first, middle, and last elements of a collection differently, taking three closures as arguments. First-class closures and user-defined control structures are more expressive and simpler than any selection of built-in control structures.

Trellis/Owl supports object-oriented data abstraction very well. It is a pure object-oriented language, in that all operations on objects are performed using message sends, with no statically-bound procedure calls in the language,** and even variables are accessed solely via messages. The Trellis/Owl implementation has been concerned primarily with eliminating the overhead for these messages [Kil88]. As its principle implementation technique, Trellis/Owl automatically compiles a separate version of each source method for each inheriting subclass. Each version is used only for receivers whose class is exactly the same as the one assumed by the copy. This allows the compiler to know the exact type of the receiver within the copied method, enabling the compiler to statically bind all messages sent to the receiver. This static binding is not possible with only a single shared version of the source method, since each subclass is free to provide new implementations for all methods defined on the receiver. Additionally, to save compiled

* These control structures were pioneered in the CLU language [LAB+81], of which Trellis/Owl is a descendant.

** Operations on classes may only be invoked on class constants (variables cannot contain classes), and so are effectively statically-bound. Operations on classes primarily create new instances.

code space, if the compiled code of a method for a subclass is the same as for a superclass, then the subclass' method will share the compiled code of the superclass; the Trellis/Owl compiler only generates a new compiled method when it differs from all other compiled methods. Our customization technique, described in Chapter 8, is similar to Trellis/Owl's "copy-down" compilation strategy.

Trellis/Owl can statically bind messages in two other situations. If the receiver of a message is a compile-time constant, the message sent to the constant can be statically bound. If the static type of the receiver of a message is annotated with **`no_subtypes`**, thus preventing the programmer from defining subclasses and overriding methods, the compiler again can statically bind the message. Unfortunately, the **`no_subtypes`** declaration violates the pure object-oriented model by explicitly preventing any future inheritance from classes declared to have **`no_subtypes`**, and therefore the declaration is a classic example of implementation efficiency concerns compromising an otherwise clean object model. In fact, the same efficiency benefits could be achieved without compromising the language definition by simply checking at link-time whether a class has any subclasses, thus inferring the **`no_subtypes`** declaration without modifying the source code.

If a message send has been statically bound, the compiler uses a direct procedure call to speed message dispatch. If the target of the message is an instance variable accessor (remember that all instance variables are accessed through messages in Trellis/Owl), then the accessor method is inlined, further increasing the speed of these messages. Currently, Trellis/Owl does not inline any other methods, even common methods such as integer arithmetic.

If a message remains dynamically bound, then Trellis/Owl uses the same in-line caching technique pioneered by the Deutsch-Schiffman Smalltalk-80 system. If this cache misses, then an external hash table specific to the class of the receiver is consulted for the target method. Trellis/Owl's calling overhead for dynamically-bound messages should compare favorably to fast Smalltalk-80 systems.

Unfortunately, no performance data is available for Trellis/Owl. The implementation does very little traditional optimization and does not even do as much optimization as fast Smalltalk-80 systems, other than copying methods down for each receiver class. The implementors openly admit that Trellis/Owl's performance is not near that of traditional languages, but report that performance is "good enough" for their users.

3.2.4 Emerald

Emerald is a statically-typed pure object-oriented language for distributed programming [BHJL86, Hut87, HRB+87, JLHB88, Jul88]. Emerald is unusual in lacking both classes and implementation inheritance: Emerald objects are completely self-sufficient. Emerald does include a separate subtyping hierarchy, however, and recent versions include a powerful mechanism for statically-type-checked polymorphism [BH90]. All Emerald data structures are objects, and the only way to manipulate or access an object is to send it a message. Thus, Emerald is just as pure as Trellis/Owl. Unfortunately, Emerald sacrifices complete purity and elegance for the sake of efficiency in a manner similar to the **`no_subtypes`** declaration in Trellis/Owl. Several common object types such as **`int`**, **`real`**, **`bool`**, **`char`**, **`vector`**, and **`string`** are built-in to the implementation and can be neither modified nor subtyped. This allows the compiler to statically bind and inline messages sent to objects statically declared to be one of the built-in types to improve efficiency, at the cost of reduced reusability and extra temptation of programmers to misuse lower-level data types. Also, some messages such as **`==`** are not user-definable; there is a single system-wide definition of **`==`**, and programmers cannot redefine or override this definition. This restriction allows the compiler to generate in-line code for **`==`** rather than generating a full message send.

Most of the implementation techniques developed for Emerald address distributed systems, such as including run-time tests to distinguish references to local objects from references to remote objects and optimizations to eliminate many of these tests. The Emerald compiler also includes a flow-insensitive type inferencer which can determine the representation-level concrete type of an expression if that expression is a literal, a variable known to be assigned only expressions of a particular concrete type, or a message send whose receiver is known to be a particular concrete type (enabling the compiler to statically-bind the message send to a particular method) and whose bound method's result is a literal. This simple interprocedural concrete type inference is used to statically-bind message sends, eliminating the run-time message lookup, and also to support the analyses for determining whether an object is guaranteed to remain local to the current node. The Emerald compiler performs no inlining of user-defined methods, however, even when statically-bound.

3.2.5 Eiffel

Eiffel is a statically-typed class-based object-oriented language supporting multiple inheritance [Mey86, Mey88, Mey92]. Like C++ and Trellis/Owl, Eiffel equates classes with types, and treats subclassing as subtyping. However, Eiffel does not verify that subclasses are legal subtypes of their superclasses, and in fact provides many commonly-used features that violate the standard subtype compatibility rules assumed by other statically-typed object-oriented languages. Eiffel provides no support for user-defined control structures, but does include an exception mechanism and an assertion mechanism that optionally checks assertions at run-time, generating an exception if an assertion is violated.

Eiffel uses dynamically-bound message passing for all procedure calls, but includes explicit variables that cannot be overridden in subclasses (Eiffel does allow methods to be overridden by variables, in which case they are accessed using dynamically-bound messages as in Trellis/Owl). This speeds variable accesses at the cost of reduced reusability; the **`polygon`** and **`rectangle`** example used in section 3.1.1 to illustrate problems with Smalltalk-80 applies equally to Eiffel.

Unfortunately, no information has been published on implementation techniques or run-time performance for Eiffel.

3.2.6 Summary

C++ is widely described as an efficient object-oriented language. Much of this efficiency, however, comes from its embedded non-object-oriented language, C. A C++ program achieves good efficiency primarily by avoiding the object-oriented features of C++, relinquishing both the performance overhead and the programming benefits associated with these features. Trellis/Owl, Emerald, and Eiffel, while being much more purely object-oriented than C++, compromise their purity with restrictions designed to enable a more efficient implementation.

C++ implementations use a virtual function table implementation that reduces the overhead of message passing over normal direct procedure calls to between two and four memory indirections and in some cases an add instruction, depending on whether the system supports single or multiple inheritance, whether virtual base classes are involved, and whether function table indices are assigned using only local information or globally using system-wide information. Unfortunately, the performance of these implementations relies on static type checking to verify that all messages will be sent to objects that understand them, and so they are not directly applicable to a dynamically-typed language such as SELF; adding in extra run-time checking to detect illegal messages would sacrifice any performance advantage held by the virtual function table implementation over a system like in-line caching.

3.3 Scheme Systems

Scheme is a dynamically-typed function-oriented language descended from Lisp [AS85, RC86]. Scheme supports several language features described in Chapter 2 as desirable, including closures and generic arithmetic. Consequently, insights into the construction of efficient Scheme implementations may help in building efficient implementations of object-oriented languages, particularly ones with user-defined control structures and generic arithmetic. Conversely, techniques developed for implementing object-oriented languages may be useful for implementing Scheme. This section describes work on efficient Scheme implementations.

3.3.1 The Scheme Language

Scheme includes a number of built-in control structures, data types, and operations. In addition, Scheme supports lexically-scoped *closures* with which Scheme programmers may build a wide variety of user-defined control structures. Closures are first-class data values which may be passed around Scheme programs, stored in data structures, and invoked at any later time, much like blocks in SELF and Smalltalk. Scheme closures and Smalltalk blocks are defined to be "upwardly mobile": they may be invoked after their lexically-enclosing scope has returned, even if they refer to local variables defined in the enclosing scope. Closures are a key ingredient of the functional programming style based on higher-order lexically-scoped functions.

Scheme also supports first-class *continuations*. A continuation is a function object that encapsulates "the rest of the program" at the time it is created. Continuations may be used by programmers to build powerful control structures, such as exception handlers, coroutines, and backtracking searches. Non-local returns in SELF and Smalltalk are a specialized form of continuation creation and invocation.

Like most Lisps, Scheme supports generic arithmetic. Therefore, although Scheme is not directly object-oriented, it includes a sizable object-oriented subsystem (albeit not extensible by the programmer). Primitive operations in Scheme are robust, performing all necessary error checking to detect programmer errors. Since variables are untyped, this checking cannot in general be performed statically, thus incurring additional run-time cost. Unchecked versions of many common primitives are also available to the speed-conscious programmer, with a corresponding loss of safety. Also, type-specific unchecked versions of arithmetic functions exist for programmers who are willing to sacrifice both safety and reusability in the quest for speed.

3.3.2 The T Language and the ORBIT Compiler

T is an object-oriented extension to Scheme [RA82, Sla87, RAM90]. It includes all the features of Scheme, including first-class closures and continuations, and adds the ability to declare dynamically-bound generic operations and object structure types. An operation called with an object as its first argument invokes the implementation of the operation associated with the object; a default implementation of the operation may be provided for built-in data types and objects that do not implement the operation. T is not a pure object-oriented language, since all the built-in data types and procedures of Scheme are still available in T, but T is better than most hybrid languages in that many built-in procedures inherited from Scheme are defined as dynamically-bound operations in T, and the T programmer can override any statically-bound procedure with a dynamically-bound version. Thus, a T programmer may turn his T system into a nearly pure object-oriented language; common practice, however, still relies heavily on the traditional built-in data types and operators from Lisp, such as **`cons`**, **`car`**, and **`cdr`**, which are not redefined as dynamically-bound operations.

The ORBIT T compiler by Kranz *et al* is a well-respected Scheme compiler [KKR+86, Kra88]. The ORBIT compiler analyzes the use of closures and continuations and attempts to avoid heap allocation of closures whenever possible. Early measurements indicate that ORBIT's performance is comparable both to other Lisps without closures and even to traditional languages such as Pascal. However, as seems to be the rule with Lisp benchmarks, the unsafe type-specific versions of arithmetic operators are used to achieve fast performance; the compiler does not optimize the performance of true generic arithmetic [Kra90]. Additionally, no user-defined control structures are used in the benchmarks measured, only the built-in control structures. Finally, ORBIT does not optimize the object-oriented features of T [Kra89].

3.3.3 Shivers' Control Flow Analysis and Type Recovery

Olin Shivers has developed a set of algorithms for constructing relatively large control flow graphs from Scheme programs using interprocedural analysis, even in the presence of higher-order functions and closures [Shi88]. The resulting large control flow graph is more amenable to traditional optimizations than the original small control flow graphs, thus potentially boosting the performance of Scheme programs to that achieved by optimizing compilers for traditional languages.

Shivers also developed a technique for *type recovery* in Scheme that attempts to infer the types of variables in programs [Shi90]. His algorithm begins with assignments from constants (which have a known type) and propagates this information though his extended interprocedural control flow graph along subsequent variable bindings, much like traditional data flow analysis (described in section 3.4.1). He proposes using this type information to eliminate run-time type tests around type-checking primitive operations. Our type analysis, described in Chapter 9, has much in common with this proposal. Unfortunately, his techniques have not yet been developed into a practical, working system; for example, his system does not yet generate machine code, only small examples have been examined, and the compiler is quite slow. Also, while Scheme programs contain higher-order functions, they are not object-oriented, so the interprocedural analysis used in the construction of the extended control flow graph upon which the type recovery is based cannot be performed easily and accurately in the presence of the dynamically-bound message passing that is characteristic of object-oriented systems.

3.3.4 Summary

While Scheme supports several of the same seemingly expensive features that SELF does, such as closures, generic arithmetic, and robust primitives, it also includes enough inexpensive alternatives for programmers to avoid the expensive features. Scheme includes built-in control structures to avoid much of the overhead of closures, and most Scheme systems include unsafe primitives and type-specific arithmetic operators to avoid the overhead of generic arithmetic and robust primitives. Of course, by avoiding the overhead of the expensive features the programmer also loses their advantages in flexibility, simplicity, and safety. Some techniques used Scheme systems, however, are relevant to SELF, including analysis of the use of closures to avoid heap allocation and analysis of type information to avoid run-time tests.

3.4 Traditional Compiler Techniques

Since we are attempting to make SELF competitive with the performance of traditional languages and optimizing compilers, we will likely need to include and possibly extend traditional optimizing compiler techniques in the SELF compiler. In this section we discuss several conventional techniques that relate to techniques used for SELF.

3.4.1 Data Flow Analysis

Much work has been done on developing techniques to optimize traditional statically-typed non-object-oriented imperative programming languages such as Fortran [BBB+57], C, and Pascal [ASU86, PW86, Gup90]. Many of these techniques revolve around *data flow analysis*, a framework in which information is computed about a procedure by propagating information through the procedure's control flow graph. The information computed from data flow analysis may be used to improve both the running time and the compiled code space consumption of programs. Some examples of optimizations performed using the results of data flow analysis include constant propagation, common subexpression elimination, dead code elimination, copy propagation, code motion (such as loop-invariant code hoisting), induction variable elimination, and range analysis optimizations such as eliminating array bounds checking and overflow checking. Several of the techniques used in the SELF compiler are akin to data flow analysis, in particular type analysis described in Chapter 9.

Data flow analyzers propagate information, usually in the form of sets of variables or values, around the control flow graph, altering the information as it propagates across nodes in the control flow graph that affect the information being computed. The analysis may propagate either forwards or backwards over the control flow graph, depending on the kind of information being computed. Data flow analysis that examines only a single piece of straight-line code (a *basic block*) is called *local*; data flow analysis that examines an entire procedure is called *global*.

Data flow analysis is complicated by join points in the control flow graph (merge nodes for forward propagation, branch nodes for backward propagation). At join points, the information derived from each of the join's predecessors may be different. Data flow analysis algorithms combine the information from the predecessors in a *conservative approximation*, meaning that no matter what path execution actually takes through the program, the information data flow analysis computes will be correct (i.e., it is conservative), although it might not be as precise as possible (i.e., it may be only an approximation). The conservativeness of data flow analysis algorithms is required in order that the optimizations performed using the computed information preserve the semantics of the original program. Ideally, the approximations are as close to the "truth" as can be achieved without too much compile-time expense.

Data flow analysis gets even more complicated in the presence of loops. The loop entry point (the head of the loop in forward data flow analysis) acts like a join point, but the first time the loop entry point is reached the information for the looping "backwards" branch has not yet been computed. One common approach to handling this problem first assumes the best possible information about the loop branch, analyzes the loop under this assumption, and keeps reanalyzing the loop until the information computed for the looping branch matches the information assumed for the looping branch. This *iterative data flow analysis* finds the best *fixpoint* in the information computed for the loop, but can be a relatively expensive operation since the body of the loop can be reanalyzed many times before the fixpoint is found. Our iterative type analysis technique works similarly, as described in Chapter 11.

Iterative data flow analysis extracts information from arbitrary control flow graphs. For certain restricted kinds of control flow graphs called *reducible* control flow graphs typically produced by programmers of traditional languages using "structured programming," asymptotically faster but more complex techniques such as *interval analysis* can be used to extract similar kinds of information. Unfortunately, the control flow graphs manipulated by the SELF compiler are not always reducible, especially after splitting loops as described in Chapter 11.

3.4.2 Abstract Interpretation

Abstract interpretation is a more semantics-based approach to the data flow analysis problem [CC77]. In this framework, the semantics of the original programming language is abstracted to capture only relevant information; this new abstract semantics is called a *non-standard semantics*. The program can then be analyzed by interpreting the program using the non-standard semantics. Of course, some conservative approximation is usually required to be able to interpret the non-standard semantics of the original program in a bounded amount of time; this approximation frequently can be captured formally in the way the non-standard semantics is defined. Abstract interpretation and data flow analysis are both techniques for statically analyzing programs, but a particular analysis problem can sometimes be described more elegantly using abstract interpretation.

3.4.3 Partial Evaluation

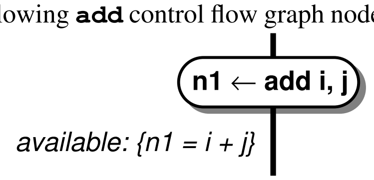
Partial evaluation is a technique for optimizing a program based on a partial description of its input [SS88]. A partial evaluator takes as input a program (say a circuit simulator), written for a large class of potential inputs, and a particular input to the program (say a circuit description), and produces as output a new program optimized for the particular input (a simulator optimized for a particular circuit). Partial evaluation is intended to allow a programming style in which a single general program is written that can be optimized for particular cases, improving on the alternative style of writing many specialized programs.

Partial evaluators produce these optimized programs using a form of interprocedural analysis, propagating the description of the input program through the entire program call graph and taking advantage of this extra information through heavy use of constant folding, procedure inlining, and the like. Partial evaluation systems also commonly produce multiple versions of a particular procedure (called *residual functions*), each optimized for a particular calling environment; procedure calls then branch to the appropriate optimized residual function instead of the more general and presumably slower original function. Our customization technique, described in Chapter 8, can be viewed as partial evaluation based on run-time information, as is discussed in section 8.4.

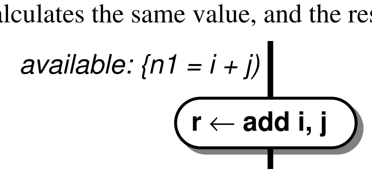
3.4.4 Common Subexpression Elimination and Static Single Assignment Form

Global common subexpression elimination is one of the most important of the traditional optimizations. The standard approach to common subexpression elimination uses data flow analysis to propagate sets of *available expressions*, which are computations that have been performed earlier in the control flow graph [ASU86]. After computing the expressions available at a control flow graph node such as an arithmetic instruction node, the compiler can eliminate the node if the result computed by the node is already available.

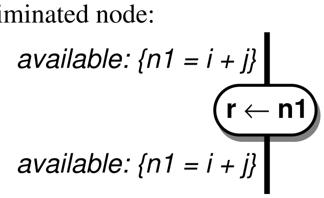
For example, the result of the following **`add`** control flow graph node would be added to the available expressions set:



If some later node in the graph calculates the same value, and the result of the earlier node is still available:



then the compiler can replace the second redundant calculation with a simple assignment node from the result of the earlier node to the result of the eliminated node:



Common subexpression elimination via comparing against available expressions relies heavily on determining when two expressions (such as the two **`i + j`** expressions above) are equivalent. When the two expressions include the same variable names, one would expect the expressions to be the same. However, any assignment to a variable referenced by an available expression would cast this equivalence into doubt, and so an expression must be removed from the available expression set whenever one of the variables in the expression are assigned. For example, if **i** were assigned a new value between the first and second computations above, then the expression must be considered unavailable, and the second expression would not be eliminated. On the other hand, if the value of **i** were assigned to third variable, **k**, and the second calculation referenced **k** instead of **i**, then the two computations would compute the same result, but the available expression system would need to be more sophisticated to track such assignments and detect that the two computations produce the same result.

Several researchers have developed techniques to improve the effectiveness of determining when two expressions are equal. Perhaps the best current approaches are based on *static single assignment* (*SSA*) form [AWZ88]. The invariant maintained by a program in SSA form is that each variable is assigned at most once, and exactly one definition of a variable reaches any use of that variable. Arbitrary programs can be transformed into an equivalent program in SSA form by replacing each definition of a variable in the original program with a definition of a fresh new variable (pedagogically named by adding a subscript to the original variable name). Each use of the original unsubscripted variable also is changed to use the appropriate subscripted variable. Finally, to preserve the invariant that exactly one definition reaches any use, at merge points reached by different subscripted variables of the same original unsubscripted variable, a fresh new subscripted variable is created for the merge and assigned the result of a ϕ-*function* of the incoming subscripted variables; such pseudo-assignments do not generate any code, but simply preserve the SSA invariant.

SSA form supports optimizations such as common subexpression elimination better than the traditional approach based on the original variable names because SSA form does not need to take into account assignments to variables; SSA's renamed subscripted variables are only assigned once. Assignments to variables do not kill existing available expressions, since the assignments are guaranteed to be to different variables after the renaming. In addition, the ϕ-functions of SSA form can track expressions as they flow through control structures, supporting better identification of constant expressions and equivalent expressions, which in turn can enable more common subexpressions to be eliminated.

As will be described in section 9.6, the SELF compiler also performs global common subexpression elimination. While the SELF compiler does not use precisely SSA form, its *values*, described in section 9.1.3, are quite similar.

3.4.5 Wegman's Node Distinction

Mark Wegman describes a generalization of several standard code duplication and code motion optimizations called *node distinction* [Weg81]. Given some differentiating criterion computable through data flow analysis, Wegman's technique splits nodes downstream of a potential merge point if the merging paths have different values of the differentiating criterion. Several traditional code motion techniques, such as code hoisting, and some novel techniques, such as splitting nodes based on the value of some boolean variable, can be expressed in this framework.

Node distinction is remarkably similar to our splitting technique, described in Chapter 10. One drawback of node distinction as described by Wegman is that the differentiating criterion must be known in advance, prior to data flow analysis and node distinction. Our splitting, on the other hand, does not require any such advance knowledge and so is more suitable for a practical compiler. The relationship between splitting and node distinction will be explored further in section 10.2.5.

3.4.6 Procedure Inlining

Many researchers have worked on improving the performance of procedure calls through inline expansion of the bodies of the callees in place of the calls; this technique is also known as *inlining*, *procedure integration*, and *beta reduction*. Some languages, including C++, provide mechanisms through which the programmer can tell the compiler to inline calls to particular routines. More sophisticated systems attempt to determine automatically which routines should be inlined [Sch77, AJ88, HC89, RG89, McF91]. Inlining itself is not a particularly difficult transformation, but it is harder to devise a set of good heuristics to control automatic inlining, balancing compiled code space and compilation time increases against projected run-time performance improvements and operating correctly in the presence of recursive routines. Chapter 7 will describe the heuristics used in the SELF compiler to guide automatic inlining. Also, inlining is possible only when the target of a call is known at compile-time, and so inlining is not directly applicable to purely object-oriented systems where messages are always dynamically bound. Many of the techniques included in the SELF compiler are designed to enable many common messages to be inlined away.

3.4.7 Register Allocation

One of the most important techniques, included in virtually all optimizing compilers, is *global register allocation*. Many modern register allocators treat the problem of allocating a fixed number of registers to a larger number of variables as an instance of *graph coloring* of the *interference graph*. The nodes of the interference graph are the variables that are candidates for allocation to registers. The interference graph contains an arc between two nodes if the corresponding variables are simultaneously live at some point in the procedure being allocated. Coloring the graph (i.e., assigning each node a color such that no adjacent nodes have the same color) corresponds to a register allocation, where each color represents a distinct register.

Since only a fixed number of registers can be used for register allocation, the goal of the coloring process is to find a coloring of the interference graph with no more colors than allocatable registers. Unfortunately, determining whether a graph is colorable with *k* colors is an NP-complete problem, so much of the work in implementing graph-coloring-based register allocators in real compilers involves developing heuristics that can usually find a coloring of the interference graph in a reasonable amount of time, and in handling spilling of variables to memory if no coloring can be quickly found [CAC+81, Cha82, CH84, LH86, BCKT89, CH90].

In practice, the nodes in the interference graph may be portions of a variable's lifetime, particularly if these portions represent disconnected regions with no real data flow between regions (i.e., separate "def-use chains"), thus allowing different parts of a variable's lifetime to be allocated to different registers. Additionally, two variables may be coalesced into a single node if the two variables are not simultaneously live and one variable is assigned to the other. This *subsumption* process can eliminate unnecessary register moves, but might also make the graph harder to color.

Section 12.1 will describe the implementation of global register allocation in the SELF compiler, discussing its current strengths and weaknesses.

3.4.8 Summary

Several implementation techniques have been developed and exploited in various language implementations of object-oriented languages that are relevant to our quest for an efficient implementation of SELF. Many of these techniques speed dynamic binding. The most effective techniques reduce a dynamically-bound message send to a statically-bound procedure call by determining the class of the receiver at compile-time. The TS Typed Smalltalk compiler and the Hurricane compiler both use type declarations to determine the types of receivers of messages, and statically-bind and possibly inline away messages sent to receivers known to be of a single type; these compilers use case analysis if the receiver may be one of a small set of types. Trellis/Owl uses a copy-down scheme to additionally statically-bind and sometimes inline away messages sent to **`self`**, in particular instance variable accesses. This static binding and inlining away of extra layers of abstraction is especially important in pure object-oriented languages.

Other techniques have been developed for cases where the type of the receiver cannot be determined statically. Smalltalk systems and Trellis/Owl use an in-line caching technique to speed message sends where the class of the receiver remains fairly constant; the resulting speed of a message send is only 3 to 4 times slower than a normal procedure call. In cases where the in-line cache misses, a hash table is used to locate the target method quickly.

Implementations of C++ implement message passing using an indirect array accessing technique. This approach exploits information present in the class hierarchy to produce a mapping from message name to array index, reducing the cost of message passing to around 2 to 3 times the cost of a normal procedure call in the single-inheritance case, 3 to 5 times the cost for the multiple-inheritance case. Some extensions to this approach can reduce the cost of the multiple inheritance scheme to just the cost of the single inheritance scheme, at the cost of some wasted space and significant extra compile time.

Most languages include built-in control structures, data types, and operations to ease the burden on the implementation of message passing. Even Smalltalk, supposedly a pure object-oriented language with no built-in control structures, includes several critical compromises in the language design to speed performance. Other languages, such as C++ and T, make no attempt at purity, and much of the processing that takes place in such languages is in the base non-object-oriented sublanguage. Consequently, the speed of the object-oriented features has not been heavily optimized, since programmers concerned with speed may "code around" the problems using the faster non-object-oriented facilities.

In summary, with existing technology the overhead of dynamic binding can be eliminated only in a limited number of cases, namely those in which the type of the receiver can be identified precisely. If a message send cannot be statically bound to the target method, then current techniques imply a direct overhead of at least 2 to 3 times slowdown in the speed of a message send in a statically-typed language, more in a dynamically-typed language. These techniques impose the additional indirect overhead of preventing inlining to reduce the cost of the extra abstraction boundaries introduced in well-designed, well-factored code and in user-defined control structures; in many cases this "indirect" overhead is much more damaging than the direct slowdown of procedure calls. Finally, the other impediments to good performance, user-defined control structures, generic arithmetic, and robust language primitives, are not significantly optimized in any of the implementations described. Consequently, the performance of existing object-oriented languages lags far behind the performance of conventional languages.