Chapter 2 Language Features and Implementation Problems

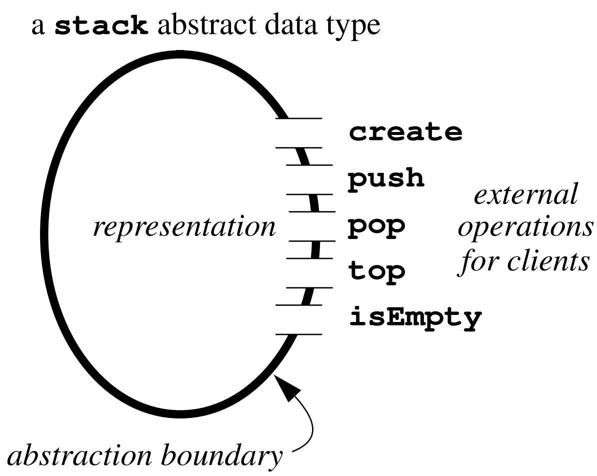
This chapter describes several desirable language features, all of which are included in SELF: abstract data types, message passing, inheritance, user-defined control structures, error-checking primitives, and generic arithmetic. For each feature, we describe its advantages for programmers, its adverse implementation consequences, and typical compromises made in other languages for the sake of efficient implementation. Those readers familiar with these language features and implementation challenges may choose to skim this chapter.

2.1 Abstract Data Types

2.1.1 Benefits to Programmers

The ability to describe and manipulate data structures is central to the expressive power of a language. Traditional programming languages such as C [KR78] and Pascal [JW85] include **`record`** and **`array`** data type declarations; Lisp [WH81, Ste84], Prolog [SS86], and many functional programming languages [MTH90, Wik87, Pey87] include **`cons`** cells. These type declarations build *concrete data types*. Manipulating concrete data structures is simply a matter of extracting fields from records or **`cons`** cells and indexing into arrays.

Abstract data types [LZ74, LSAS77, LAB+81, LG86] provide a more expressive mechanism for describing and manipulating data structures. An abstract data type abstracts away from a concrete data type by providing a set of operations (the *interface*) through which clients are to manipulate objects of the type. The abstract data type is implemented in terms of some lower-level data type (the *representation*), but this implementation is hidden from clients behind the abstract data type's abstraction boundary. For example, a canonical abstract data type is the **`stack`** data type, supporting **`create`**, **`push`**, **`pop`**, **`top`**, and **`isEmpty`** operations and represented using an array of stack elements and an integer top-of-stack index.



The enforced abstraction boundary provides advantages to both implementors and clients of abstract data types over traditional concrete data types. Implementors are free to change the representation of an abstract data type, and as long as the interface remains the same, clients of the abstract data type remain unaffected. For example, the **`stack`** data type could be reimplemented using a linked list in place of an array and an integer, and clients would be unaffected. Thus, abstract data types encapsulate design decisions that may change, especially those about the representation of critical data structures.

For clients, abstract data types provide a more natural interface for manipulating data structures than the language primitives used with concrete data structures. The operations on abstract data types can directly reflect the conceptual operations on the data type the programmer has in mind, rather than being translated into series of extraction and indexing operations. In the **`stack`** example, clients may use the more natural **`push`** and **`pop`** operations in place of array indexes and integer increments. These abstract operations also improve the reliability of the system, since adding an element to a stack is implemented in a single place and debugged once, rather than being repeated in every client at every call.

Abstract data types also provide a principle for organizing programs. When using abstract data types, the task of programming an application tends to revolve around identifying, designing, and implementing abstract data types. For many applications, this orientation is better than the more traditional orientation of top-down refinement of procedures and functions [Wir71]. In addition, libraries of common abstract data types are developed that may be reused in future applications, reducing development and maintenance costs.

2.1.2 Implementation Effects

Widespread use of abstract data types greatly increases the frequency of procedure calls over traditional programming styles using concrete data types. Each manipulation of a concrete data type, such as record field extraction or array indexing, is a built-in language construct, easily implemented by simple compilers in a few machine instructions. With abstract data types, however, each manipulation is conceptually a procedure call that invokes the programmer's implementation of the abstract operation. In a system with heavy use of abstract data types, many operations are implemented by the programmer to just call lower-level operations on the representation data type, magnifying the overhead of abstract data types.

To eliminate the run-time cost of abstraction, implementations can expand the body of a called procedure in place of the procedure call; this technique is known as *procedure integration* or *inlining*. When an operation on an abstract data type is invoked, the compiler can expand the implementation of the operation for that abstract data type in-line, eliminating the procedure call. With aggressive use of inlining, the overhead of abstract data types can be virtually eliminated, removing a performance barrier that might discourage the use of an important program structuring tool. This inlining depends, however, on the fact that within a given program there is only a single implementation for a particular abstract data type (this condition does not exist for object-oriented programming with message passing, described next in section 2.2). If the implementation of an abstract data type changes, then the whole program may need to be recompiled to inline the new operation implementations. Even in non-inlining implementations of abstract data types, however, some amount of relinking after changing the implementation of an abstract data type is usually necessary.

By inlining the implementation of an operation in place of its call, the compiler has in some ways violated the abstraction boundary of the abstract data type. Fortunately, the compiler does not need to follow the same restrictions as the human programmers, and so this "violation" is quite reasonable. Abstraction boundaries are great for people to help organize their programs, but serve little purpose for the implementation.

2.2 Object-Oriented Programming

2.2.1 Benefits to Programmers

Object-oriented programming languages improve abstract data types by provide *objects* or *classes* instead [Weg87]. Object-oriented languages typically include two features not found in languages with only abstract data types: *message passing* and *inheritance*.

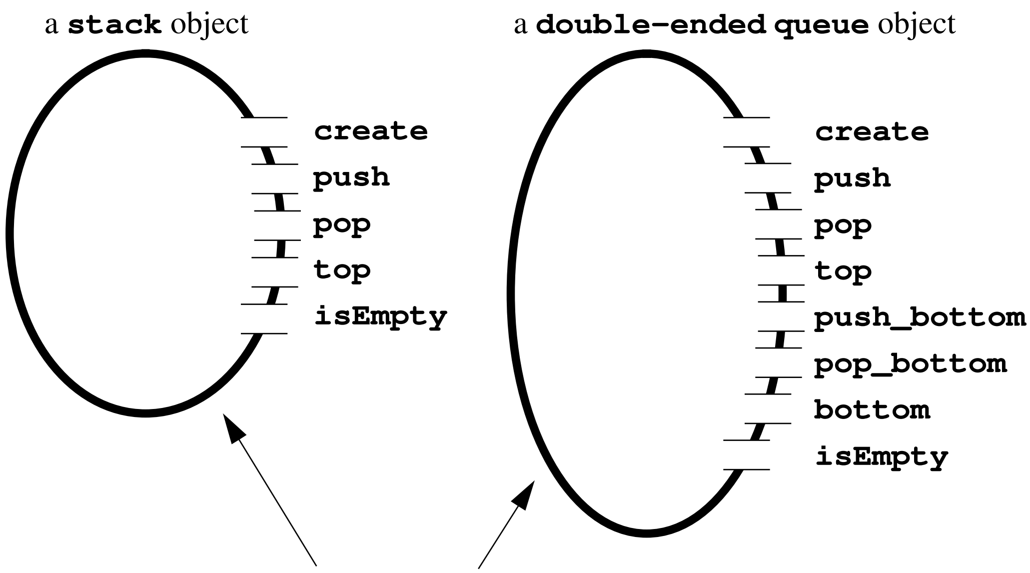
2.2.1.1 Message Passing

With abstract data types, clients are insulated from implementation details of abstract data types, allowing the implementor of an abstract data type to replace the implementation of the abstract data type with a new one without rewriting client code. Unfortunately, only one implementation of an abstract data type can exist in the system at a single time, and changing the implementation of an abstract data type is a compile-time operation that requires recompiling and relinking an application with the new implementation.

Object-oriented programming languages rectify this problem, allowing multiple implementations of the same abstract data type to coexist in the same application *at run time*. Client code does not depend on which implementation of an abstract data type is being accessed, and in fact different implementations of the abstract data type can be manipulated at different times by the same client code. For example, both array-based and stack-based implementations of stacks can be manipulated by clients interchangeably.

To have this flexibility make sense, the operation invoked by some call must be determined dynamically based on the actual implementation used in the call. For instance, when invoking the **push** operation on a stack (in object-oriented terminology, *sending* the **push** *message*), the procedure that gets run (the *method* that implements the message) depends on which implementation of stacks is being operated on. If the stack passed as an argument to the **push** operation (receiving the **push** message) is a linked-list stack, then the linked-list-stack-specific **push** method should be invoked; if the stack is an array-based stack, then the array-based-stack-specific **push** method should be run. Since for different calls different stack implementations might be used, this determination of which **push** implementation to invoke must be determined dynamically at run-time. *Message passing* is arguably the key to the expressive power of object-oriented programming.

In a statically-typed language, variables are associated with types (typically by explicit programmer declaration but sometimes by automatic compiler inference), and the type of a variable describes the operations that may be performed on data values or objects stored in the variable. In traditional languages, including those with abstract data types, the only data values that can be stored in a variable are those of the same type as the variable. In object-oriented languages, where clients can manipulate objects of different implementations interchangeably via message passing, this restriction on the types of objects stored in a variable is relaxed: an object can be stored in a variable as long as the object supports at least the operations expected by the variable's declared type. The object stored in the variable can provide more operations that expected by the variable (can be a *subtype* of the variable's declared type), since these extra operations will be ignored by the client code. For example, client code that operates on stacks, say by sending the **push** and **pop** messages, will continue to operate correctly on any other object that supports the **push** and **pop** messages, such as a double-ended queue that supports both stack operations and additionally **push_bottom** and **pop_bottom** operations to add and remove elements from the opposite end of the stack.



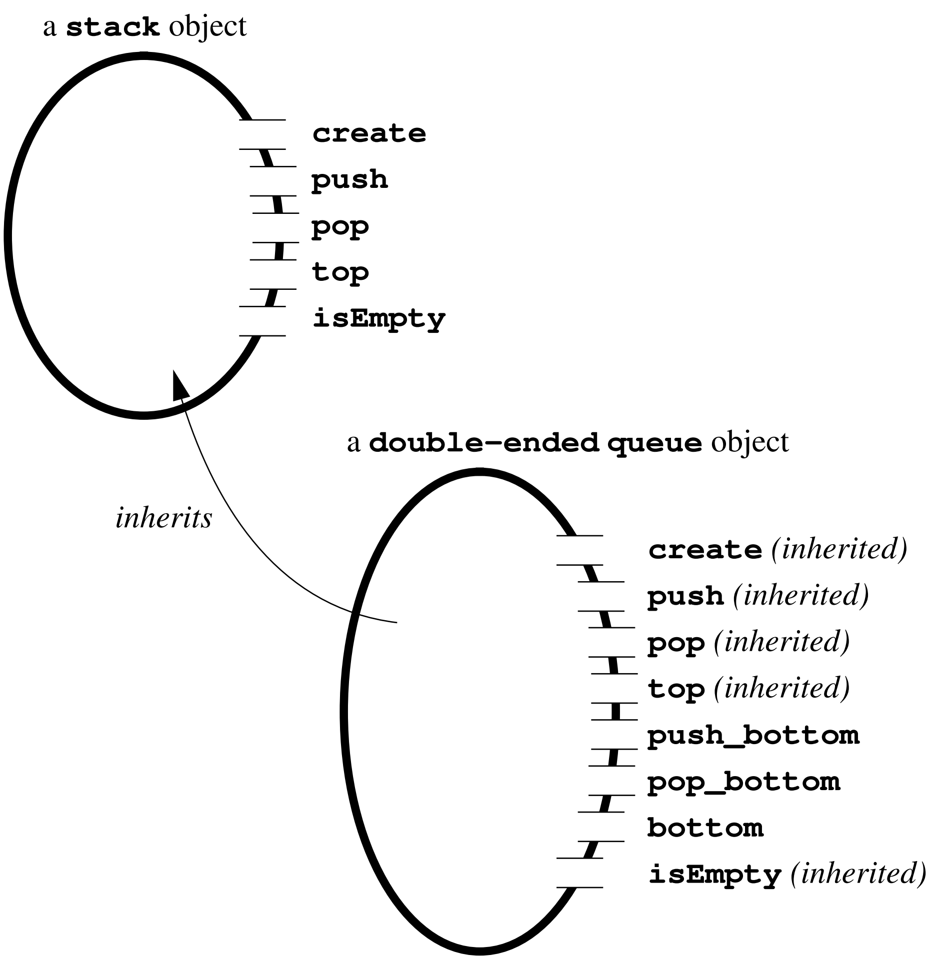
interchangeable to clients sending **push** *and* **pop**

In general, the various types of objects in a system form a *lattice*, with more general types (i.e., types with fewer required operations) higher in the lattice and more specific types lower in the lattice. In most object-oriented languages this type lattice is restricted to be the same as the implementation inheritance graph (implementation inheritance is described in the next section).

This looser connection in object-oriented languages between the statically-declared type of a variable and the actual run-time type of the contents of the variable, enabled by the use of message passing to dynamically select the appropriate implementation for a call, dramatically increases the potential reusability and applicability of client code. Clients are further abstracted from implementation and representation issues by specifying only what operations are required of objects, either explicitly using static type declarations or implicitly by the operations actually invoked, not the implementation of the objects or even the precise interface or abstract data type of the objects. This level of abstraction limits dependencies between clients and implementations to just those strictly required for correctly performing the client's task, and allows the client code to be used with implementations that had not been written or even imagined at the time the client's code was written.

2.2.1.2 Inheritance

Frequently, two data types may have similar implementations. This commonality may be separated out (*factored*) into a third data type and then shared (*inherited*) by the two original data types. For example, the linked-list implementation of double-ended queues may be very similar to the linked-list implementation of stacks, and consequently the programmer could factor out the similar parts into a third module that is inherited by both linked-list-based stacks and linked-list-based double-ended queues. In fact, it is likely that the double-ended queue could inherit directly from the stack implementation without a third shared implementation being necessary. In this situation, the stack implementation would play the role of a reusable implementation and make the implementation and maintenance of the double-ended queue much easier.



Factoring enables programs to be modified more easily since there is only one copy of code to be changed; changes to factored code are automatically propagated to the inheriting data types. Factoring also facilitates extensions, since the shared objects provide natural places for new operations to be implemented and automatically inherited by many similar data types. For example, the programmer could add a **`size`** operation to stacks and double-ended queues would automatically receive the same capability via inheritance.

These hierarchies of related data types are a characteristic feature of object-oriented systems. Object-oriented programming extends abstract data type programming as an organizing principle for programs by supporting hierarchies of implementations; statically-typed object-oriented languages also support hierarchies or lattices of interfaces or types as described in the previous section. These hierarchies offer a focus for the initial design problem, a catalog of pre-designed, pre-implemented components upon which new applications can build, and a framework in which new components can be integrated and made available to other programmers.

2.2.2 Implementation Effects

A client manipulates an object by sending it messages listed in the object's interface. However, since object-oriented languages allow several implementations to co-exist for any given interface, the method invoked for a particular message send depends on the implementation of the receiver of the message. This implementation cannot always be determined statically and in fact frequently may vary from one invocation to the next. Thus the system must be able to determine the correct binding of the message send site to invoked method dynamically, potentially at each call. This *dynamic binding*, while key to the expressive power of object-oriented programming, is the chief obstacle to good performance of object-oriented systems.

Dynamic binding incurs the extra run-time cost needed to locate the correct method based on the implementation of the message receiver object. This lookup involves an extra memory reference in some implementations (e.g., C++ and Eiffel) and a hash table probe in others (e.g., Smalltalk-80 and Trellis/Owl), on top of the normal procedure call overhead.

A more disastrous problem, however, is that dynamic binding prevents the inlining optimization used to reduce the overhead of abstract data types. Inlining requires knowing the single possible implementation for an operation. This requirement directly conflicts with object-oriented programming which purposefully severs the links between client operation calls and the particular implementations they invoke. Consequently, in general dynamically-bound message sends cannot be inlined to reduce the call overhead.

Inheritance can slow execution by requiring the run-time message dispatcher to perform a potentially lengthy search of the inheritance graph to locate the method matching a message name. Consequently, most implementations of message passing and inheritance use some form of cache to speed this search. Inheritance can also slow programs in a more subtle way by encouraging programmers to write well-factored programs, which have a higher call density than traditional programming styles. This overhead takes the form of messages sent to **`self`**, which would not have existed had the program not been factored using inheritance.

2.2.3 Traditional Compromises

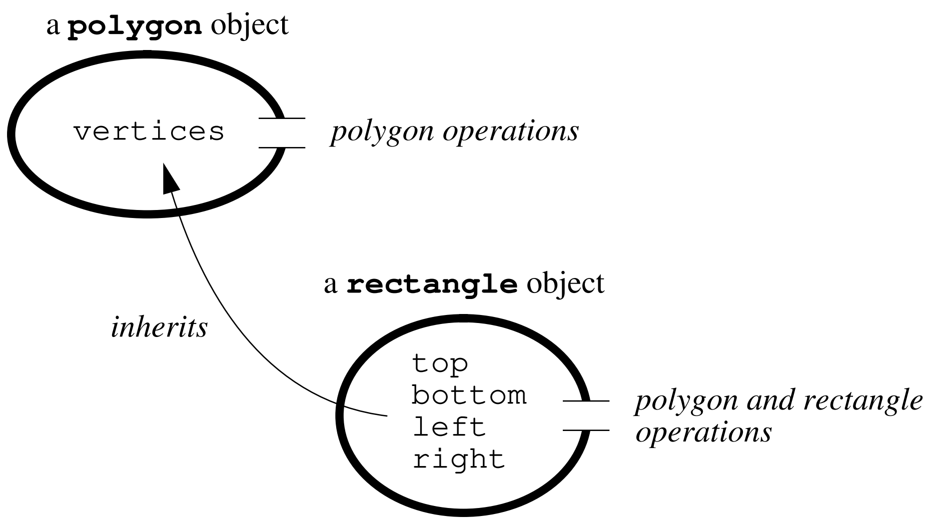
A pure object-oriented language, i.e., one that uses only message passing for computation and does not include non-object-oriented features such as statically-bound procedure calls or built-in operators, offers the maximum benefit from message passing and inheritance. Unfortunately, message passing slows down procedure calls with extra run-time dispatching and prevents the crucial inlining optimizations that are needed to reduce the overhead of abstraction boundaries. Since supporting a pure object-oriented language seems so inefficient, existing practical implementations of object-oriented languages do not support a completely pure object-oriented model, instead making various compromises in the name of efficiency.

Often language designers compromise by including non-object-oriented features or by extending an existing non-object-oriented language with object-oriented features, as with C++ and CLOS. These languages include all the built-in control structures and data types of the base non-object-oriented languages, and the base language features suffer from none of the performance problems associated with object-oriented features. For example, C++ includes all the built-in control structures available in C, and built-in data types such as integers, arrays, and structures may all be manipulated using traditional C operators without extra overhead. However, these mixed languages have the serious drawback that code written using the non-object-oriented features cannot benefit from any of the advantages of the object-oriented features. For example, programs written to manipulate standard fixed-precision (e.g., 32-bit) integers cannot later be used with arbitrary precision integers, even though both data types implement the same operations. Additionally, code for collections of objects cannot be used to create a collection of fixed-precision integers, since integers are not objects. Programmers of a hybrid language must choose between a well-written, reusable program and good run-time performance.

Even languages that are supposedly pure object-oriented languages in which all data structures are objects and all operations are dynamically-bound messages frequently "cheat" for the most common language features in an effort to improve performance. For example, Smalltalk-80, widely regarded as one of the purest object-oriented languages, hard-wires into the implementation the definitions of some common operations, such as **+** and **<** applied to integers, preventing programmers from changing their implementation. Other operations such as **==**, **ifTrue:**, and **whileTrue:** are treated specially by the implementation and are not dynamically-bound operations at all; the single implementation of each of these messages is built into the compiler and cannot be changed or overridden by the programmer. They are simply built-in operations and control structures for Smalltalk, albeit written in normal message sending syntax.

Most object-oriented languages limit the power of *instance variables* (parts of the representation of objects, like fields of records). In these languages, instance variables are accessed directly by the methods in the object's implementation, rather than by sending messages to **self** to access instance variables. Accesses to them may then be implemented by just a load or store instruction, significantly faster than normal dynamically-bound operations. Unfortunately, this practice reduces the potential reusability of abstractions by preventing instance variables to be overridden by inheriting abstractions in the same manner as dynamically-bound methods.

For example, a **polygon** data type might define a **vertices** instance variable containing a list of vertices making up the polygon. The programmer might wish to define a **rectangle** data type as inheriting from the **polygon** data type, but with a new representation: four integers defining the **top**, **bottom**, **left**, and **right** sides of the rectangle.



The programmer could make rectangles compatible with polygons by overriding the **vertices** instance variable with a **vertices** method that computed the list of vertices from the four integer instance variables. Unfortunately, in most object-oriented languages overriding an instance variable is not possible, and so **rectangle** cannot inherit directly from **polygon** as pictured. Trellis/Owl and SELF are two notable exceptions to this unfortunate practice.

Finally, object-oriented languages with static typing usually restrict the type lattice to be the same as the inheritance graph: if one object inherits from another, then the child object must be a subtype of the parent, and if one type is a subtype of another then it must inherit from the other. This restriction may perhaps be justified as a language simplification. Some languages go even further, however, by restricting the inheritance graph to form a tree; an object may inherit from only one other object. This restriction to *single inheritance* simplifies the implementation, allowing relatively efficient implementations of dynamic dispatching using indirect procedure calls (e.g., the implementation of *virtual function calls* in versions of C++ supporting only single inheritance). Unfortunately, when subtyping is tied to inheritance of implementation, single inheritance can be very limiting to programmers. General abstract types such as **comparable** and **printable** cannot be easily defined and used as supertypes of the appropriate objects, since any particular object cannot be a subtype of more than one such abstract type. Supporting multiple supertypes for one object imposes significant extra run-time overhead on message sends given existing implementation technology, as described in Chapter 3.

2.3 User-Defined Control Structures

2.3.1 Benefits to Programmers

Programs can be smaller and more powerful if the language allows arbitrary chunks of code to be passed as arguments to operations. These chunks of code are called *closures* or *blocks* [SS76, Ste76] and enable programmers to implement their own iterators, exception handlers, and other sorts of control structures. For example, the **`stack`** data type could provide an operation called **`iterate`** that would take a closure as its argument. The operation would iterate through the elements of the stack, invoking the closure on each element in turn. This arrangement would be similar to a traditional **`for`** loop but could be defined entirely by the programmer using only abstract data types and closures. Like the body of a **`for`** loop, a closure is *lexically-scoped*, meaning that it has access to the local variables of the scope in which it is defined (e.g., the caller of the **`iterate`** operation). The **`stack`** data type might also provide an operation named **`popHandlingEmpty`** that would take a closure as an argument and either pop the stack (if not empty) or invoke the closure to handle the empty-stack error. In this situation, the closure would act like an exception handler.

Closures typically provide a way to prematurely exit computations, either using first-class *continuations* as in Scheme [AS85, RC86, HDB90] or using *non-local returns* as in Smalltalk-80 or SELF. When returning non-locally, the closure returns not to its caller (e.g., the **`popHandlingEmpty`** operation) but from its lexically-enclosing operation (e.g., the caller of **`popHandlingEmpty`**). Thus non-local returns have an effect similar to **`return`** statements in C.

Closures can support all of the traditional control structures, including **`for`** loops, **`while`** loops, and **`case`** statements. In an object-oriented language, even basic control structures such as **`if`** statements may be completely implemented using closures and messages; the implementation of the **`if`** message for the **`true`** object is different than that for the **`false`** object, for instance. Thus closures enable pure object-oriented languages to be defined without *any* built-in control structures other than message passing, non-local returns, and some sort of primitive loop or tail-recursion operation, simplifying the language and moving the definition of control structures into the domain of the programmer.

2.3.2 Implementation Effects

Unfortunately, straightforward implementations of control structures using closures introduces more run-time overhead than traditional built-in control structures. Allocation and deallocation of closure objects bog down such user-defined control structures when compared to built-in control structures which can usually be implemented by a few compare and branch sequences. This allocation and deallocation cost is especially significant for extremely simple control structures such as **`if`** statements. For looping statements such as **`while`** and **`for`**, the allocation and deallocation cost can be amortized over the iterations of the body of the loop, but the extra procedure calling cost for invoking the methods comprising the user-defined control structure and for invoking the closure object each time through the loop still incurs a significant amount of overhead over the few instructions execution for a comparable built-in control structure. In the SELF system a traditional **`for`** loop runs more than 20 methods during the execution of the control structure, many of which are invoked for every iteration of the loop. In pure object-oriented languages in which these procedure calls are really dynamically-bound messages, the cost becomes even greater, especially since inlining of the user-defined code implementing the control structure becomes much harder.

2.3.3 Traditional Compromises

Because of the difficulty of efficient implementation, few languages support closures and user-defined control structures. Most include only built-in control structures and require programmers to build their own iterator data structures. Some, such as Trellis/Owl, provide built-in iterators and exceptions, supporting two of the most common uses for closures. However, many kinds of user-defined control structures go beyond simple iteration and exception handling, and these control structures cannot be implemented directly in Trellis/Owl.

Scheme provides first-class closures and continuations, but also provides a number of built-in control structures; these built-in control structures are implemented more efficiently (and invoked more concisely) than are general user-defined control structures using closures. Most Scheme programs rely heavily on these built-in control structures to get good performance. Smalltalk-80 nominally relies entirely on user-defined control structures and blocks (Smalltalk's term for closures). Unfortunately, as mentioned in section 2.2.3, Smalltalk-80 restricts some common control structures such as **`ifTrue:`** and **`whileTrue:`** so that the compiler can provide efficient implementations that do not create block objects at run-time. The primary disadvantage of such restrictions, from the point of view of the Smalltalk programmer, is that the large performance differential between the restricted control structures optimized by the compiler and control structures defined by the user tempts the programmer to use the fast control structures even if they are less appropriate than some other more abstract control structures. This implementation compromise thus discourages good use of abstraction.

2.4 Safe Primitives

2.4.1 Benefits to Programmers

At the leaves of the call graph of a program are the *primitive operations* built into the system, such as object creation, arithmetic, array accessing, and input/output. Frequently a primitive operation is defined only for particular types of arguments. For example, arithmetic primitives are defined only for numeric arguments, and array access operations are defined only for arrays and integer indices within the bounds of the corresponding array. Even procedure calls could be considered primitive operations that are legal only as long as there is enough stack space for new activation records.

In many environments, especially compiled, optimized environments, the programmer is responsible for ensuring that primitive operations are only invoked with legal arguments. If the program contains an error that leads to a primitive being invoked illegally, the system can become corrupted and probably crash mysteriously sometime later in execution. For example, C implementations do not check for array accesses out-of-bounds, and so an out-of-bounds store into an array can corrupt the internal representation of another object. Subsequent behavior of the system becomes unpredictable. Programs developed on such unsafe systems are extremely difficult to debug, since some programming errors can lead to seemingly random behavior far away in time and space from the cause of the errors.

On the other hand, a *safe* or *robust* system always verifies for each invocation of a primitive operation that its arguments are legal and that the primitive can be performed to completion without error. If the primitive call is illegal, then a robust system either halts gracefully (for example by entering a debugger) or invokes some user-definable routine or closure, thus enabling the programmer to handle the error. Robust programming systems make program development much easier by catching programming errors quickly, giving the programmer a much better chance at identifying the cause of the illegal invocation. Since a robust system never becomes internally corrupted as a result of a programming error, the penalty for such errors is greatly reduced, speeding the debugging process.

2.4.2 Implementation Effects

Implementing safe primitives requires type checking and sometimes range checking (such as for array accesses out of bounds) for arguments to primitives. With statically-typed non-object-oriented languages, the type checking of primitive arguments can be done at compile-time. However, with object-oriented languages and dynamically-typed languages, this type checking cannot in general be performed statically, thus incurring extra run-time overhead. Run-time range checking in general cannot be optimized away even in statically-typed non-object-oriented languages.

2.4.3 Traditional Compromises

Few languages provide completely robust primitives. Most languages check the types of arguments to primitives, either at compile-time (for statically-typed languages) or at run-time (for dynamically-typed languages), and some check that array references are always in bounds (at least as an option). Few systems handle procedure call stack overflow gracefully.

2.5 Generic Arithmetic

2.5.1 Benefits to Programmers

Most languages incorporate multiple numeric representations, such as integers and floating point numbers of various ranges and precisions. These representations offer different trade-offs between accuracy and efficiency. Some languages allow these numeric representations to be freely mixed in programs, and support automatic conversion from one numeric representation to another. For example, a language supporting this kind of *generic arithmetic* might include arithmetic primitives that handle overflows and underflows by returning results in representations with greater range or precision than the original arguments to the primitives (or providing the means for programmers to implement their own conversion routines). Languages with generic arithmetic relieve the programmer of the burden of dealing with numeric representation issues. Code written with one numeric representation in mind becomes automatically reusable for all other numeric representations without any explicit programmer interactions.

2.5.2 Implementation Effects

Generic arithmetic imposes significant run-time overhead. The system must perform extra run-time dispatching to select an implementation of the numeric operation appropriate for the representation of the arguments. This dispatching overhead is similar to that imposed by message passing; in fact, generic arithmetic can be viewed as an object-oriented subpart of a language, albeit one that in an otherwise non-object-oriented language may not be user-extensible. Generic arithmetic also requires extra run-time checking for overflows and underflows.

Overflows and underflows impose a serious indirect cost that is often overlooked when calculating the cost of generic arithmetic. Since the representation of the result of an arithmetic operation may be different than the representation of the operation's arguments, the compiler cannot in general statically determine the representation of the result of a numeric operation even if the compiler has determined the representation of the arguments. For example, even if the compiler knows that the type of the arguments to an operation are represented as standard machine integers, the result may be represented as an arbitrary-precision integer if an overflow occurs. Thus overflow checking limits the effectiveness of traditional flow analysis to track the representations of numeric quantities.

2.5.3 Traditional Compromises

Because of these costs, few languages support generic arithmetic. Of those that do, several also provide alternative representation-specific arithmetic operations that avoid the run-time overhead associated with generic arithmetic, but also sacrifice the safety and expressiveness of generic arithmetic.

2.6 Summary

Object-oriented languages provide a number of important enhancements over traditional procedural programming languages, among them abstract data types, message passing, and inheritance. User-defined control structures enhance the abstract data type model, and when coupled with object-oriented features eliminates the need for built-in control structures. Safe primitives are a must for an effective development environment. Support for generic arithmetic increases both the programmer's power and the program's reliability.

Unfortunately, these desirable language features don't come cheap. They impose significant implementation costs, particularly in run-time execution speed. Abstract data types and user-defined control structures conspire to dramatically increase the frequency of procedure calls, and dynamic binding both increases the cost of these procedure calls and prevents direct application of traditional optimizations such as procedure inlining. Generic arithmetic and safe primitives increase the expense of the basic operations at the leaves of the call graph.

The standard approach to solving these problems in existing language implementations is to cheat. Abstract data types are compromised by distinguishing variables and functions in interfaces. Common control structures, operations, and data types are built into the language definition, forcing programmers to choose between reusable, malleable programs and execution speed. Generic arithmetic support is either non-existent or too expensive to use, and error-checking of primitives is forgone in the name of execution speed.

SELF includes all the features described in this chapter as important, desirable language features. (The SELF language will be described in detail in Chapter 4.) However, we were unwilling to cheat to get good performance. This dilemma was the driving force that led to the research described in this dissertation.