The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages

A Dissertation
Submitted to the Department of Computer Science
and the Committee on Graduate Studies
of Stanford University
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

Craig Chambers

March 13, 1992

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David Ungar (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John Hennessy

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mark Linton

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Object-oriented programming languages promise to improve programmer productivity by supporting *abstract data types, inheritance*, and *message passing* directly within the language. Unfortunately, traditional implementations of object-oriented language features, particularly message passing, have been much slower than traditional implementations of their non-object-oriented counterparts: the fastest existing implementation of Smalltalk-80 runs at only a tenth the speed of an optimizing C implementation. The dearth of suitable implementation technology has forced most object-oriented languages to be designed as hybrids with traditional non-object-oriented languages, complicating the languages and making programs harder to extend and reuse.

This dissertation describes a collection of implementation techniques that can improve the run-time performance of object-oriented languages, in hopes of reducing the need for hybrid languages and encouraging wider spread of purely object-oriented languages. The purpose of the new techniques is to identify those messages whose receiver can only be of a single representation and eliminate the overhead of message passing by replacing the message with a normal direct procedure call; these direct procedure calls are then amenable to traditional inline-expansion. The techniques include a *type analysis* component that analyzes the procedures being compiled and extracts representation-level type information about the receivers of messages. To enable more messages to be optimized away, the techniques include a number of transformations which can increase the number of messages with a single receiver type. *Customization* transforms a single source method into several compiled versions, each version specific to a particular inheriting receiver type; customization allows all messages to **`self`** to be inlined away (or at least replaced with direct procedure calls). To avoid generating too much compiled code, the compiler is invoked at run-time, generating customized versions only for those method/receiver type pairs used by a particular program. *Splitting* transforms a single path through a source method into multiple separate fragments of compiled code, each fragment specific to a particular combination of run-time types. Messages to expressions of these discriminated types can then be optimized away in the split versions. The techniques are designed to coexist with other requirements of the language and programming environment, such as generic arithmetic, user-defined control structures, robust error-checking language primitives, source-level debugging, and automatic recompilation of out-of-date methods after a programming change.

These techniques have been implemented as part of the compiler for the SELF language, a purely object-oriented language designed as a refinement of Smalltalk-80. If only pre-existing implementation technology were used for SELF, programs in SELF would run one to two orders of magnitude slower than their counterparts written in a traditional non-object-oriented language. However, by applying the techniques described in this dissertation, the performance of the SELF system is five times better than the fastest Smalltalk-80 system, better than that of an optimizing Scheme implementation, and close to half that of an optimizing C implementation.

These techniques could be applied to other object-oriented languages to boost their performance or enable a more object-oriented programming style. They also are applicable to non-object-oriented languages incorporating generic arithmetic or other generic operations, including Lisp, Icon, and APL. Finally, they might be applicable to languages that include multiple representations or states of a single program structure, such as logic variables in Prolog and futures in Multilisp.

Acknowledgments

My heartfelt appreciation and thanks go to my advisor, David Ungar, for providing me with the great opportunity to participate in the SELF project. David always treated me as a colleague, promoted my work to others, and did his best to prepare me for independent research. He taught me much about research, writing, speaking, advising, and cartooning. I could never have had such an enjoyable and educational graduate experience without him. I will strive to be as good to my students as he was to me.

Besides serving on my thesis reading committee and providing much important feedback, Mark Linton also provided advice and direction during my first year at Stanford. His weekly research group meetings throughout my years at Stanford were interesting and educational. I appreciate Mark's strong support of my work. John Hennessy also served on my reading committee; his perceptive comments on my dissertation significantly improved its presentation, and in the process taught me a great deal about good research.

I also thank the other members of the SELF team for stimulating discussions and fun outings. Thanks to Elgin Lee for sharing an office and the birth of the SELF system with me. Bay-Wei Chang and Urs Hölzle worked and played with me as the SELF project matured. These patient people put up with my not-so-occasional coffee-induced tirades and continued to listen and discuss after the caffeine wore off. I will always remember fondly the times we've shared. More recently, Randy Smith, Ole Agesen, John Maloney, and Lars Bak have kept the SELF group a fun and stimulating collection of people.

Others at Stanford provided moral support and social diversions that helped me through. Ross Finlayson, Steve "Brat" Goldberg, Paul Calder, and John Vlissides were particularly good friends. My brother-in-common-law, Martin Rinard, made my time at Stanford interesting, to say the least.

I appreciate the support and patience of my new colleagues at the University of Washington as I finished this dissertation concurrently with my other responsibilities. By helping to make my transition from student to professor so smooth, they enabled me to finish this tome relatively quickly and painlessly. I also owe much to my early training as an undergraduate at MIT in Barbara Liskov's research group. Mark Day, Bob Scheifler, Paul Johnson, and Bill Weihl, my undergraduate thesis advisor, taught me about research and real system-building, and this experience enabled me to get started on research at Stanford right away.

Finally, I thank my family for their continued support and encouragement. I thank Bill McLaughlin, one of my real brothers-in-law, for his help in putting together the final version of this thesis and taking care of the last administrative details. To my wife, Sylvia, I owe my sincerest gratitude and give my deepest love.

This research has been generously supported by the National Science Foundation, Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and Digital Equipment Corporation.

Table of Contents

Chapter 1 Introduction

1.1 Introduction

Programming language designers have always been searching for programming languages and features that ease the programming process and improve programmer productivity. One promising current approach is *object-oriented programming*. Object-oriented programming languages provide programmers with powerful techniques for writing, extending, and reusing programs quickly and easily. Object-oriented languages typically involve some sort of *inheritance* of implementation, allowing programmers to implement new abstract data types in terms of implementations of existing abstract data types, and some sort of *message passing* to invoke operations on objects of unknown implementation. Several object-oriented languages have been designed and implemented, including Smalltalk-80* [GR83], C++ [Str86, ES90], Trellis/Owl [SCW85, SCB+86], Eiffel [Mey86, Mey88, Mey92], Modula-3 [Nel91, Har92], CLOS [BDG+88], and T [RA82, Sla87, RAM90]. Unfortunately, traditional implementations of object-oriented language features, particularly message passing, have been much slower than traditional implementations of their non-object-oriented counterparts, and this gap in run-time performance has limited the widespread use of object-oriented language features and hindered the acceptance of purely object-oriented languages.

Language designers have developed several other important language and environment features to improve programmer productivity. First-class *closures* allow programmers to define their own control structures such as iterators over collection-style abstract data types and exception handling routines. *Generic arithmetic* supports general numeric computation over a variety of numeric representations without explicit programmer intervention. A *safe*, *robust* implementation performs all necessary error checking to ensure that programs do not behave in an implementation-dependent way (e.g., get a "mystery core dump") when they contain errors such as array access out of bounds or stack overflow. Complete *source-level debugging* helps programmers get programs working quickly, significantly improving programmer productivity. Unfortunately, most languages and implementations do not support all these desirable features, again because they historically have had a high cost in run-time performance.

1.2 The SELF Language

To maximize the potential benefits of object-oriented programming, David Ungar and Randy Smith designed the SELF programming language [US87, HCC+91, UCCH91, CUCH91] as a refinement and simplification of the Smalltalk-80 language. SELF incorporates a purely object-oriented programming model, closures for user-defined control structures, generic arithmetic support, a safe, robust language implementation, and support for complete source-level debugging. (SELF will be described in more detail in Chapter 4.) Ungar and Smith strove to provide a simple, flexible language and environment that maximized the expressive power and productivity of the programmer. However, SELF's powerful features initially appeared to make its implementation prohibitively inefficient: the fastest implementation of Smalltalk-80, a language that does not include all the features of SELF, runs a set of small benchmark programs at only a tenth the speed of optimized C programs.

1.3 Our Research

The goal of the work described in this dissertation is to design and build an efficient implementation of SELF on stock hardware that does not sacrifice any of the advantages of the language or environment. Achieving our goal required us to develop new implementation strategies for message passing, closures and user-defined control structures, generic arithmetic, robust primitives, and source-level debugging. Our results have been surprisingly good: the same set of benchmarks used to measure the performance of Smalltalk-80 programs indicate that SELF programs run between a third and half the speed of the optimized C programs, roughly five times faster than the Smalltalk-80 implementation. These new techniques are practical, since SELF's compilation speed is roughly the same as an optimizing C compiler, and SELF's compiled code space usage is usually within a factor of two of optimized C.

Our new implementation strategies work well in overcoming the obstacles to good performance for SELF. Fortunately, these new techniques could be included in the implementations of other object-oriented languages to improve their

* Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

run-time performance. Even non-object-oriented languages incorporating user-defined control structures and/or generic arithmetic could benefit from our new implementation techniques. More importantly, we hope that the techniques we have developed will pave the way for other purely object-oriented languages to be designed and implemented without including compromises or restrictions solely for the sake of efficiency.

1.4 Outline

The next chapter of this dissertation describes the benefits of object-oriented programming, user-defined control structures, generic arithmetic support, and a robust implementation, as well as their associated costs in run-time performance. It also outlines various compromises and restrictions that other languages have included to achieve better run-time efficiency. Chapter 3 reviews this related work in detail. Chapter 4 describes the SELF language.

Chapters 5 through 13 contain the meat of the dissertation. Chapter 5 presents the goals of this work and outlines the organization of the compiler. Chapter 6 describes the framework in which the compiler functions, including the memory system architecture and the run-time system. (An early design and implementation of the memory system was described in Elgin Lee's thesis [Lee88].) Chapters 7 through 12 present the bulk of the new techniques developed to improve run-time performance. These techniques include *customization*, *type analysis*, *type prediction*, and *splitting*. Earlier designs and implementations of these techniques have been described in other papers [CU89, CUL89, CU90, CU91]. Chapter 13 describes compiler support for the SELF programming environment, in particular techniques that mask the effects of optimizations such as inlining and splitting from the SELF programmer when debugging; some of these techniques have been described in other papers [CUL89, HCU92]. Section 5.3 contains a more detailed outline of this part of the dissertation.

The performance of our SELF implementation is analyzed in Chapter 14. This analysis measures various configurations of our implementation and identifies the individual contributions to performance of particular techniques. The compile time and space costs of the system as a whole and of individual techniques are analyzed as well.

Finally, Chapter 15 concludes the dissertation and outlines some areas for future work.

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.

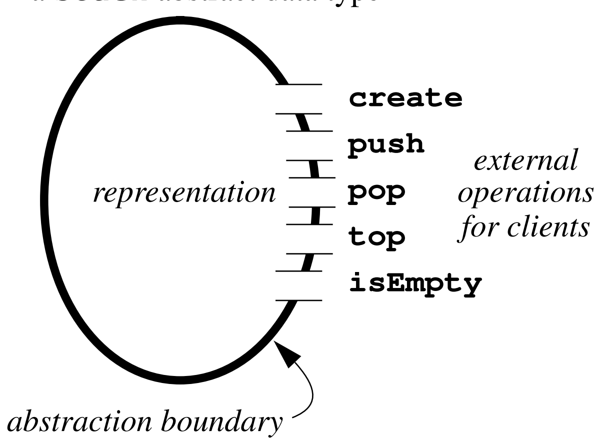
2.1 Abstract Data Types

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.

a **`stack`** abstract data type



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*.

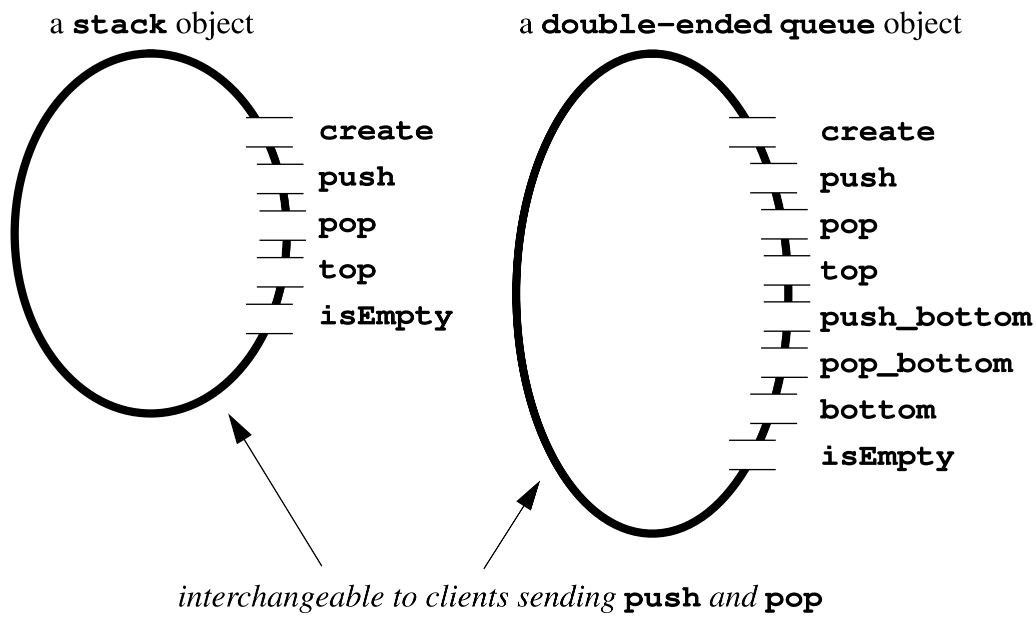
2.2.1.1 Message Passing

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.

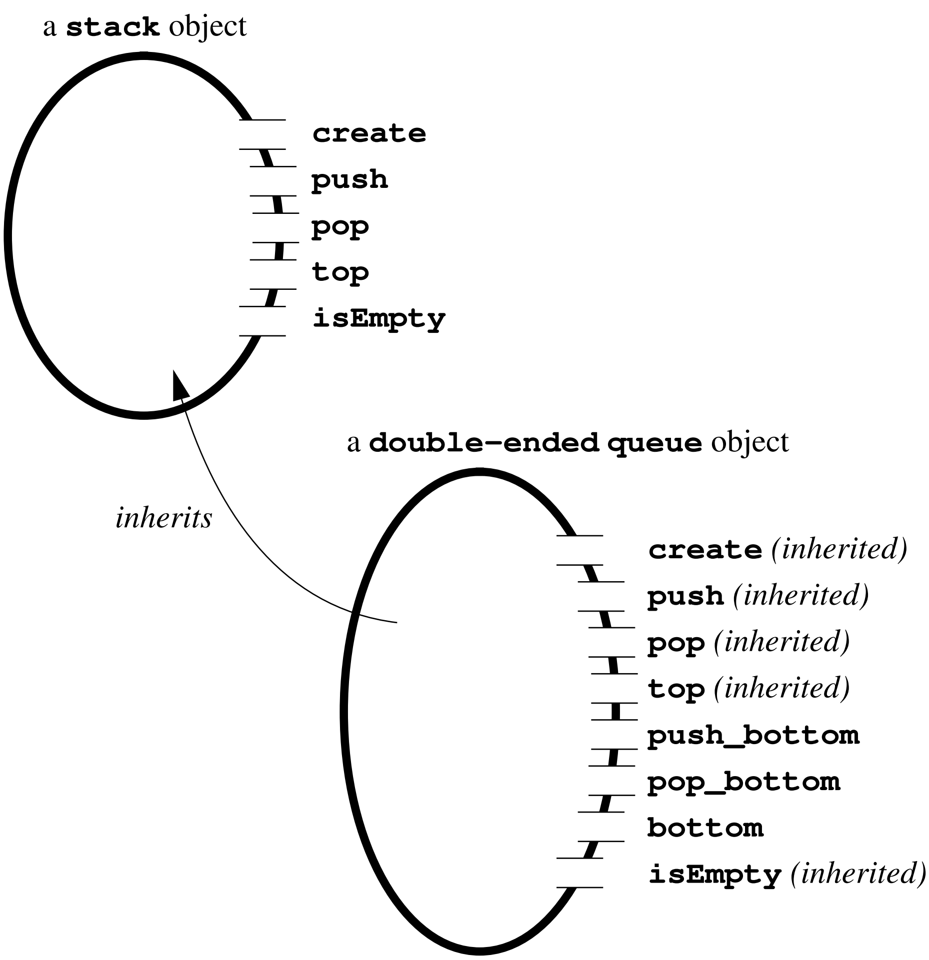


In general, the various types of objects in a system form a *lattice*, with more general types (i.e., types with fewer required operations) higher in the lattice and more specific types lower in the lattice. In most object-oriented languages this type lattice is restricted to be the same as the implementation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

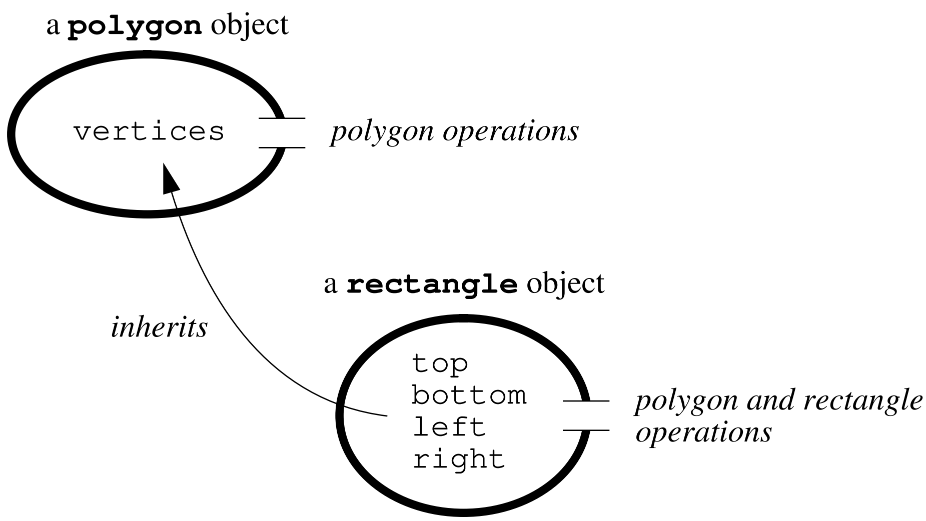
A pure object-oriented language, i.e., one that uses only message passing for computation and does not include non-object-oriented features such as statically-bound procedure calls or built-in operators, offers the maximum benefit from message passing and inheritance. Unfortunately, message passing slows 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.

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.

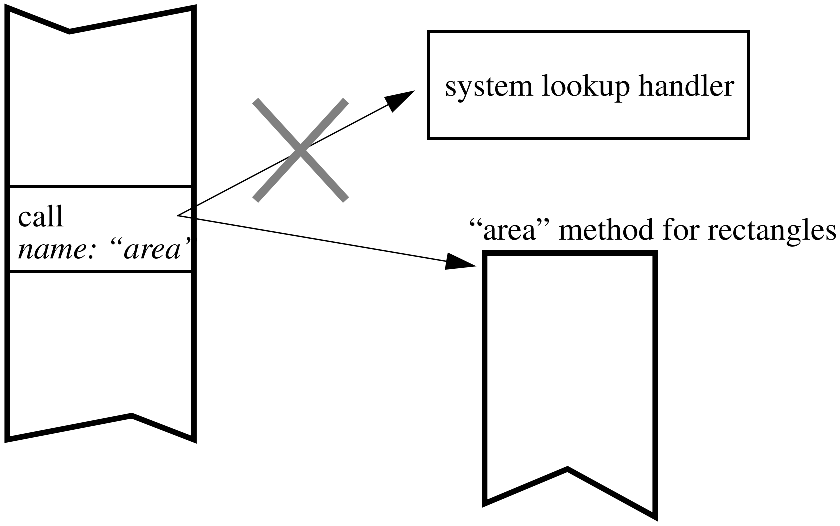
Peter Deutsch and Allan Schiffman developed several techniques for implementing Smalltalk-80 programs better than these interpreters without specialized hardware support [DS84]. They overcame the interpretation overhead by introducing an extra invisible translation step from virtual machine byte codes 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.

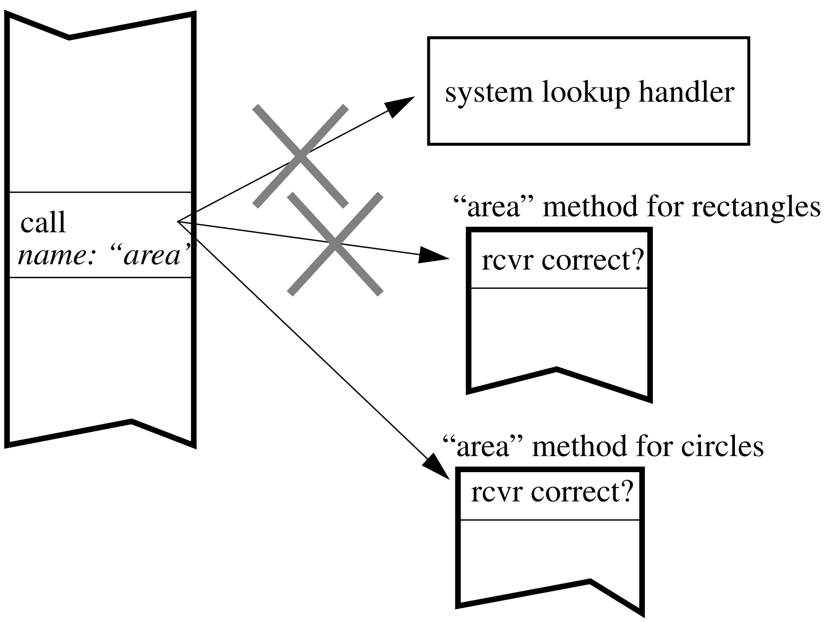
With dynamic compilation, the compiled code can be treated as a *cache* on the more compact byte-coded representation of programs. If a compiled method has not been used recently, its code can be thrown away (flushed from the cache) to save compiled code space. If a method is needed again later, it can 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.

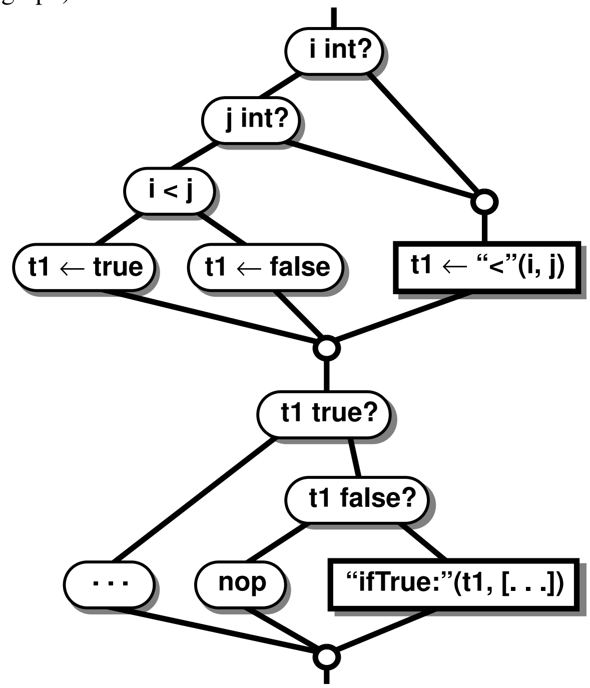
To reduce the burden on the programmer of specifying static type declarations, the Typed Smalltalk system includes a *type inferencer* that, when invoked, can automatically infer the appropriate type declarations for a routine. The programmer must provide type declarations for instance variables and 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.

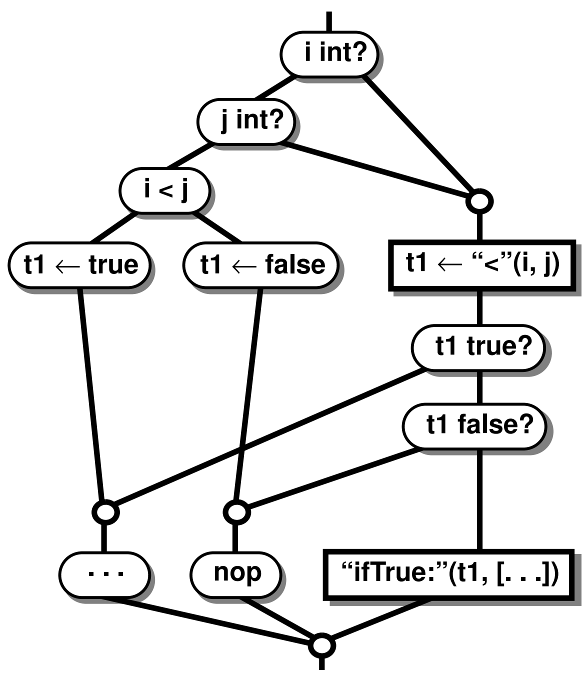
In the TS compiler, the front-end translates the Typed Smalltalk code into a relatively machine-independent *register-transfer language* (RTL); the back-end then optimizes these RTL instructions and converts them into native machine code. Primitive operations may be written by the programmer in the same 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.

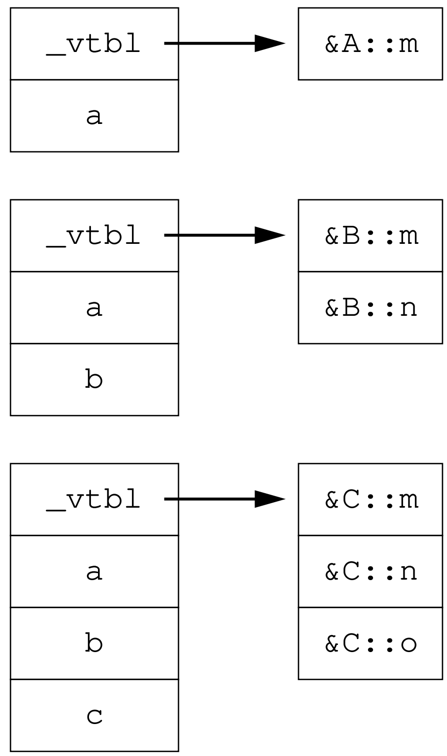
C++ equates types with classes. A subclass may specify whether or not it is to be considered a subtype of its superclass(es) (by inheriting from the superclass either **`public`**ly or **`private`**ly); if so, the C++ compiler verifies that the subclass is a legal subtype of the superclass(es). With 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.

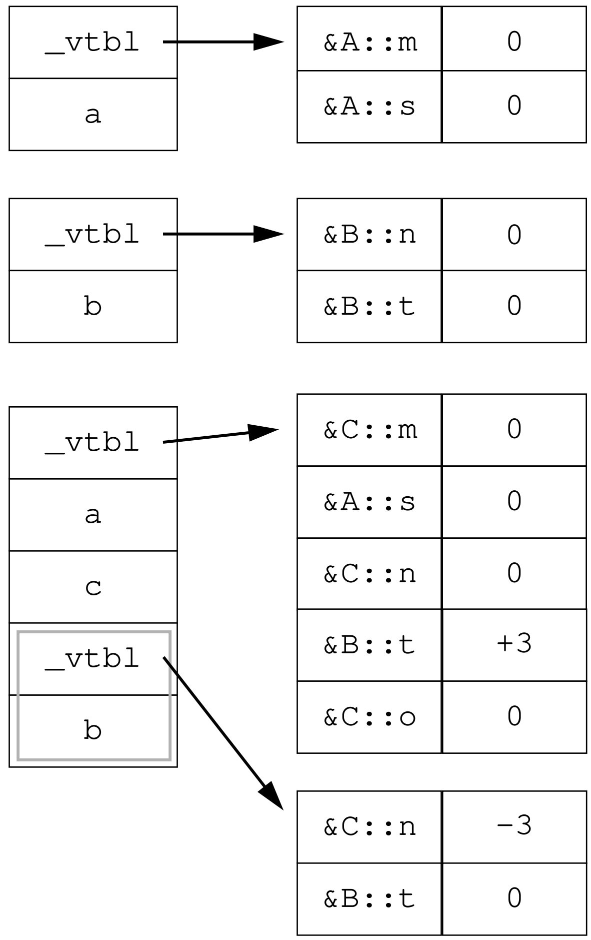
This **`_vtbl`** technique may be extended to work with the versions of C++ that support multiple inheritance. The trick is to embed in the object a complete representation, including the **`_vtbl`** pointer, of each superclass other than the first, and to pass the address of the embedded object whenever 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.

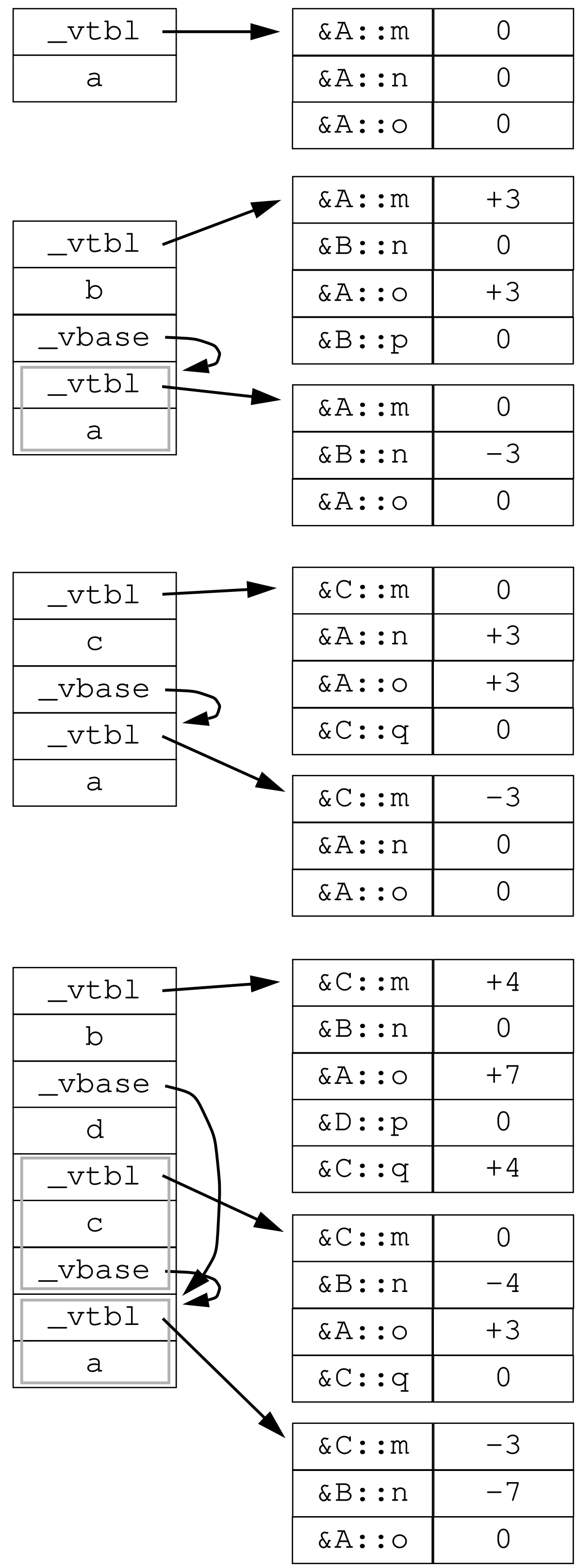
Unfortunately, the usefulness of these techniques is limited by their need to examine the complete class hierarchy prior to assigning indices to message names and compiling code. Additionally, adding a new class may require the system-

```
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

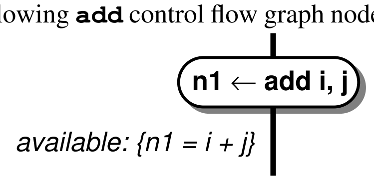
Partial evaluation is a technique for optimizing a program based on a partial description of its input [SS88]. A partial evaluator takes as input a program (say a circuit simulator), written for a large class of potential inputs, and a particular input to the program (say a circuit description), and 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.

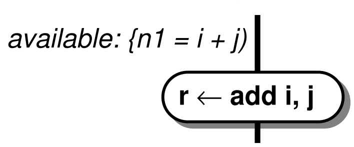
3.4.4 Common Subexpression Elimination and Static Single Assignment Form

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.

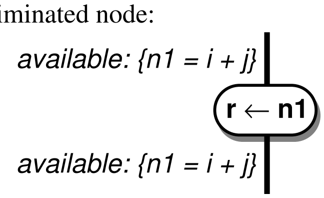
For example, the result of the following **`add`** control flow graph node would be added to the available expressions set:



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.

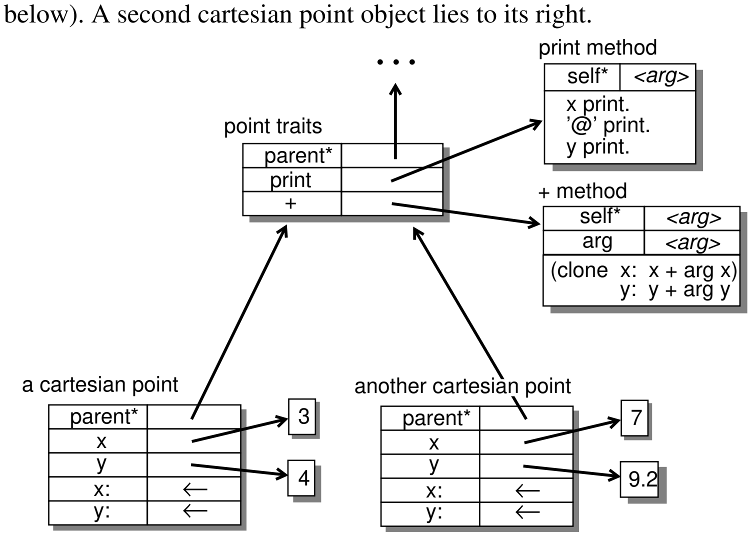
Chapter 4 The SELF Language

SELF is a dynamically-typed prototype-based object-oriented language with multiple, dynamic inheritance, originally designed by David Ungar and Randy Smith at Xerox PARC in 1986 [US87, HCC+91, UCCH91, CUCH91] as a successor to the Smalltalk-80 programming language. Like Smalltalk, SELF is intended for exploratory programming environments in which rapid program development and modification are primary goals. Hence SELF is dynamically-typed, affording greater flexibility and ease of development and modification, at the cost of reduced reliability and, given existing implementation technology, reduced run-time performance. Additionally, SELF includes the features described in Chapter 2 as desirable in an object-oriented language: abstract data types, a pure object-oriented model with dynamic binding on all messages (including all variable accesses), closures for user-defined control structures and exceptions, robust primitives, and support for generic arithmetic. Those readers familiar with SELF may choose to skim this chapter.

4.1 Basic Object Model

A SELF object consists of a set of *named slots*, each of which contains a reference to some other object. Some slots may be designated as *parent slots*. Objects may also have SELF source code associated with them, in which case the object is a *method*. To make a new object in SELF, an existing object (called the *prototype*) is simply *cloned* (shallow-copied) to produce a new object with the same name/value pairs as the prototype.

For example, the following picture portrays several SELF objects. The bottom-left object represents a cartesian point "instance" containing 5 slots: a parent slot named **`parent`** (identified as a parent slot by the asterisk next to the slot's name) containing a reference to the point traits object, two slots named **`x`** and **`y`** containing references to integer objects, and two slots named **`x:`** and **`y:`** that contain references to the assignment primitive method (notated using the ← symbol and described below). A second cartesian point object lies to its right.



The top-left object labeled **`point traits`** is inherited by all cartesian point objects. It also contains a parent slot named **`parent`** containing a reference to another object not shown in this diagram, a slot named **`print`** and **+** each containing a reference to a method object.

Method objects differ from other objects only in that they have attached SELF code in addition to slots. Each method object has a parent slot named **`self`** that is an argument slot; its contents in filled in with the receiver of the message when the method is invoked, as described below. The **+** method has an additional argument slot named **`arg`** that is filled in with the right-hand argument to the **+** message when the method is invoked. The two integer objects also have their own slots, but for conciseness we omit them from this diagram.

Two other kinds of objects appear in SELF: object arrays and byte arrays. Arrays are just like normal data objects, except that they additionally contain a variable number of array elements indexed by number instead of name. As their names suggest, object arrays contain elements that are arbitrary objects, while byte arrays contain only integer objects in the range 0 to 255, but in a more compact form suitable for interacting with external character- or byte-stream based systems. Primitive operations support fetching and storing elements of arrays as well as determining the size of an array and cloning a new array of a particular size.

4.1.1 Object Syntax

A programmer may describe a SELF object in textual form by listing the object's slots and its code inside parentheses. The slots are listed between vertical bars at the beginning of the object, with the code following afterwards; either of these components of an object may be omitted. A slot declaration begins with the slot's name, then an asterisk if the slot is a parent slot,* then either a left-arrow or an equal sign depending on whether or not, respectively, an assignment slot is desired, and then an expression which is evaluated to determine the slots contents. An assignable slot initialized to **`nil`** may be declared concisely by omitting the left-arrow and the initializer expression. Slots are separated by periods.

For example, the cartesian point object above could be defined as follows (comments are between double quotes):

```
( |
  parent* = traits point. "evaluates to the point traits object"
  x <- 3. "left-arrow creates corresponding assignment slot"
  y <- 4.
| )
```

This example illustrates the use of **=** to define a single data slot and **<–** to define a data slot/assignment slot pair; the name of the assignment slot is computed by appending a colon to the name of the data slot.

The point traits object could be defined as follows:

```
( |
  parent* = ... "code to evaluate to the parent of point traits"
  print = ( x print. '@' print. y print ).
  + = ( | :arg | (clone x: x + arg x) y: y + arg y ).
| )
```

The **`print`** and **+** method objects are defined directly as contents of slots. Method objects look just like other object declarations, except that they specify code in addition to any slots. Argument slots are prefixed with colons and may not be initialized. SELF defines a syntactic sugar for argument slots that allows them to be written in as part of the slot name; the **+** slot declaration could also have been written as follows:

```
+ arg = ( (clone x: x + arg x) y: y + arg y ).
```

SELF includes a few other forms for object literals, including integer and floating point literal expressions that evaluate to the corresponding integer and floating point objects and string literals delimited by single quotes.

4.2 Message Evaluation

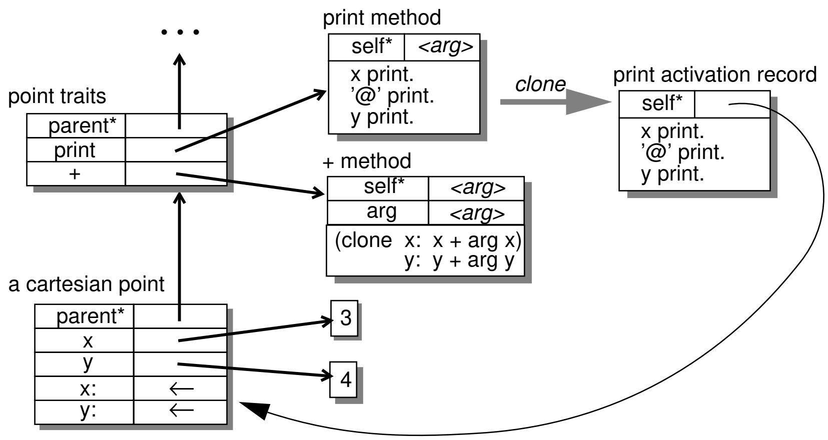
When a message is sent to an object (called the *receiver* of the message), the receiver object is scanned for a slot with the same name as the message. If a matching slot is not found, then the contents of the object's parent slots are searched recursively, using SELF's multiple inheritance rules to disambiguate any duplicate matching slots. For example, if the **`x`** message were sent to the cartesian point object pictured above, the system would search the cartesian point for a slot whose name is **`x`**, locating the slot referring to the 3 object. If instead the **`print`** message were sent to the cartesian point object, the system would first scan the cartesian point object for a slot named **`print`**, unsuccessfully. The system would then search each object stored in a parent slot of the cartesian point, which in this example would be the point traits object, and the system would find the matching **`print`** slot in this parent object.

Once a matching slot is found, the object referred to by the slot is *evaluated* and the result is returned as the result of the message send. An object without code evaluates to itself, and so the slot holding it acts like a variable. For example, when sending the **`x`** message to the cartesian point, the system locates the **`x`** slot in the point, extracts its contents (the

* The current version of SELF supports prioritized parents with differing numbers of asterisks for different parent priorities. Further details may be found in [CUCH91].

3 integer object), evaluates it (in this case just returning 3 again, since the 3 object contains no code and hence evaluates to itself), and returns the result (3) as the result of the original **`x`** message.

An object with code (a method) is treated as a prototype activation record. When evaluated, the system clones the method object, fills in the clone's **`self`** slot with the receiver of the message, fills in the clone's argument slots with the arguments of the message (if any), and executes its code. For example, if the **`print`** message were sent to the cartesian point, the system would locate the **`print`** slot in the point traits object, extract the **`print`** method object referenced by the slot, and evaluate the method object. Evaluating the method would involve cloning the method object to create a fresh activation record, filling in the contents of the **`self`** slot of the new activation record with the receiver cartesian point object, and then executing the messages specified by the code associated with the **`print`** method. The result of the last message in the **`print`** method would be returned as the result of the **`print`** message.



SELF supports assignments to data slots by associating an *assignment slot* with each assignable data slot. The assignment slot contains the *assignment primitive* method object, which takes one argument. When the assignment primitive is evaluated as the result of a message send, it stores its argument into the associated data slot. A data slot with no corresponding assignment slot is called a *constant* or *read-only slot* (as opposed to an *assignable data slot*), since a running program cannot change its value. For example, most parent slots are constant slots. However, SELF's object model allows a parent slot to be assignable just like any other slot, simply by defining its corresponding assignment slot. Such an assignable parent slot permits an object's inheritance to change on-the-fly at run-time, for instance as a result of a change in the object's state. We call such run-time changes in an object's inheritance *dynamic inheritance*, and we have found this facility to be of practical value in our SELF programming. Further information on the uses of dynamic inheritance may be found in [UCCH91].

4.2.1 Message Syntax

SELF message syntax is much like Smalltalk-80 message syntax. Both languages define three classes of message, distinguished syntactically:

- *Unary messages*. A unary message takes no arguments other than the receiver. Syntactically, a unary message name is written after its receiver expression (in postfix form), and is distinguished from other forms of message name by being an sequence of letters or digits that begins with a lower-case letter and does not end with a colon. Thus **`x`**, **`print`**, and **`isFirstQuadrant`** are all valid unary message names. Unary messages have highest precedence, and associate from left to right.

- *Binary messages*. A binary message takes a receiver and one argument, with the binary message name separating the two. A binary message is easily distinguished as any sequence of punctuation characters (excluding a few reserved sequences). Thus **`>`**, **`&&`**, **`=`**, and **`&^$#^`** are legal binary message names. Binary messages have medium precedence. No associativity is defined for binaries (programmers must explicitly add parenthesis to disambiguate sequences of binary messages), except that two binary messages left-associate if they are the same binary message. Therefore expressions like **`3 + 4 + 5`** are legal, with **`3 + 4`** being evaluated first, while expressions like **`3 + 4 * 5`** are illegal and must be explicit parenthesized. Arguments are always evaluated from left to right.

- *Keyword messages*. A keyword message takes a receiver and one or more arguments. Keyword message names are unusual in that the message name is written interspersed between the arguments to the message. Each piece of a keyword message name is a sequence of letters and digits, beginning with a letter, and ending with a colon (unlike unary messages which do not end with a colon). To aid in limiting the number of parentheses required for parsing, the first keyword piece must begin with a lower-case letter, while all subsequent keyword pieces must begin with an upper-case letter. The receiver is written before the keyword message, with an argument after each colon at the end of the keyword message pieces. The name of the message is the concatenation of the various name pieces. Therefore, **`x:`**, **`ifTrue:`**, and **`ifTrue:False:`** are all legal keyword message names, the first two taking one argument (and a receiver) and the third taking two arguments, while **`ifTrue:ifFalse:`** is not.

  Keyword messages have lowest precedence and associate from right to left. For example, the message **`x ifTrue: 5 False: 6`** sends the **`ifTrue:False:`** message to the result of the **`x`** message with 5 and 6 as arguments, while the message **`x ifTrue: 5 ifFalse: 6`** first sends the **`x`** message, then the **`ifFalse:`** message to 5 with 6 as an argument, and then the **`ifTrue:`** message to the result of the **`x`** message with the result of the **`ifFalse:`** message as an argument, as if the original message were parenthesized as **`x ifTrue: (5 ifFalse: 6)`**.

The code part of a method is simply a sequence of period-separated messages.

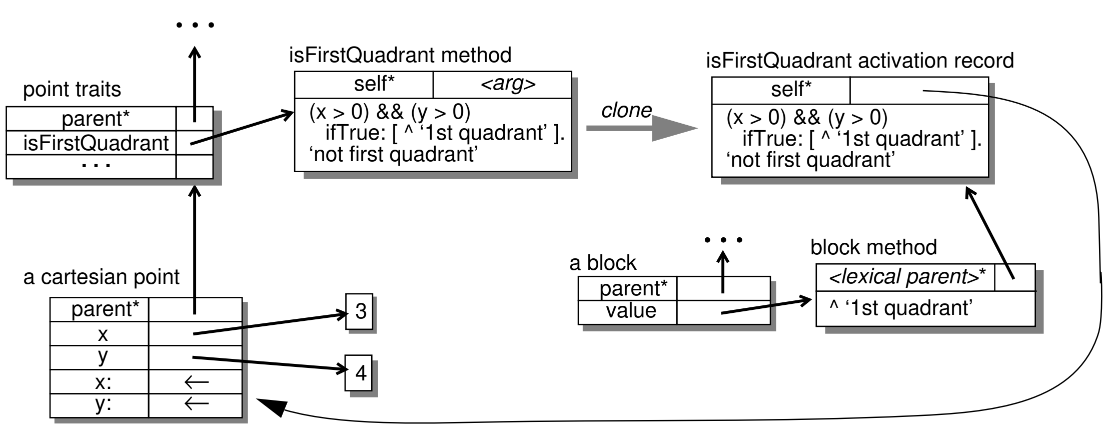
4.3 Blocks

SELF allows programmers to build their own control structures using *blocks*, SELF's version of closures. A block in SELF is an object with a slot named **`value`** that contains a special kind of method. When invoked (by sending **`value`** to the block object), this special block method runs as a child of its lexically-enclosing activation record (the activation record that was executing when the block object was created). A block method does not include a **`self`** parent slot, but instead has an anonymous parent slot that refers to the lexically-enclosing activation record object; the value of **`self`** is inherited from the enclosing method activation. These differences from "normal" methods enable blocks and block methods to act like lexically-scoped closures; SELF uses normal inheritance to implement lexical scoping.

Syntactically, blocks are identical to other method definitions, except that they are enclosed in square brackets instead of parentheses. In particular, variables local to a block activation record are declared as normal data slots in the slot list of the block literal.

For example, suppose an **`isFirstQuadrant`** method were added to the point traits object. This method tests whether both the **`x`** and **`y`** components of the receiver point are positive and if so returns the string literal **`'1st quadrant'`**. Otherwise the string literal **`'not first quadrant'`** is returned.

The following diagram shows the state of the system after invoking the **`isFirstQuadrant`** method and creating a new activation record.



The block object corresponds to the block literal enclosed in square brackets in the **`isFirstQuadrant`** method. The block's **`value`** slot refers to a block method object with an anonymous lexical parent slot, which refers to the block's lexically-enclosing activation record object.

A block method may terminate with a *non-local return* by prefixing the result expression with a ^ symbol (reminiscent of an up-arrow), causing the result to be returned not to the caller of the block method (the sender of **`value`**) but to the caller of the lexically-enclosing normal (non-block) method. Non-local returns thus have much the same effect as a **`return`** statement in C. For example, when executing the non-local return in the **`isFirstQuadrant`** example, the block would return not to the sender of **`value`** somewhere inside the **`ifTrue:`** user-defined control structure but instead to the caller of the lexically-enclosing method, in this case returning the **`'1st quadrant'`** string object to the sender of **`isFirstQuadrant`**.

4.4 Implicit Self Sends

Local variables and arguments are accessed in SELF using *implicit self* sends. These sends have **`self`** as the receiver of the message but begin the search for a matching slot with the current activation record rather than **`self`**. This search will follow the lexical chain of activation records (following the anonymous parent slots of nested block methods). Since arguments and local variables are simply normal slots in method prototype objects and their cloned activation records, implicit self message sends can support argument and local variable accesses using the same mechanisms used to access data slots and methods in "normal" objects. Since **`self`** is a parent slot of the outermost method activation record, implicit self sends can also be used to access slots in the receiver or its ancestors.

Implicit self sends are so termed because the **`self`** receiver is elided from the message send syntax; a message without an explicit receiver is implicitly a send to **`self`**. For example, the point **+** method contains the fragment **`x + arg x`**. This code first sends the **`x`** message to **`self`** implicitly. The lookup starts with the current activation record, and since the activation record does not contain an **`x`** slot, the system will scan the contents of the activation record's parent slots. The activation record's **`self`** slot is the only parent slot, and so the system will search the contents of the **`self`** slot, the receiver cartesian point, for an **`x`** slot. This search will be successful, and the system will evaluate the contents of the **`x`** slot to compute the result of the **`x`** message.

The example code fragment will next send the **`arg`** message to **`self`** implicitly. Again the lookup will begin with the current activation record, but this time the system will find a matching **`arg`** slot in the activation record. The contents of the **`arg`** local slot are accordingly evaluated, returning the argument to the original **+** message send.

Implicit self messages allow the SELF syntax for local slot accesses and slot accesses in the receiver to have the same concise syntactic expression as local, instance, and global variable accesses in Smalltalk, but with the more powerful semantics of full message sends. In particular, code which looks like it is accessing an instance variable, and which originally did access an instance variable, can be reused in situations in which the message actually invokes a method. This possibility enables the SELF system to solve such thorny reuse problems as the **`polygon`** and **`rectangle`** example from section 2.2.3 and the **`cartesian`** and **`polar`** point example to be described in section 4.6.

4.5 Primitives

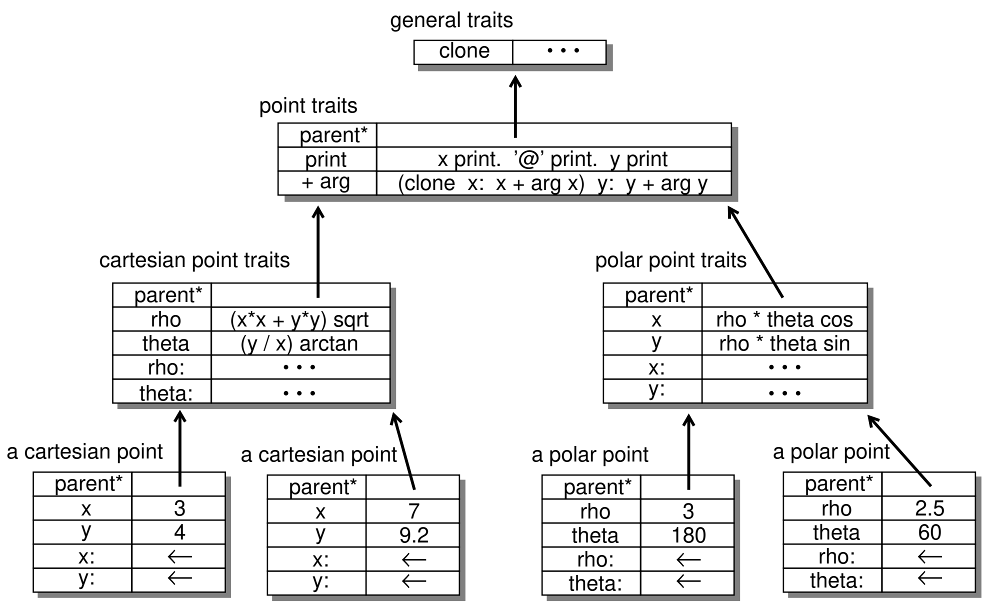
Much of the real work of a SELF program is performed by primitive operations provided by the virtual machine and implemented below the level of the language. Integer arithmetic, array accessing, and input/output are all provided via primitives to the SELF programmer. Primitive operations are invoked with the same syntax used to send a message, except that the message name begins with an underscore ("_"). For instance, **`_IntAdd:`** invokes the standard integer addition primitive. Every call of a primitive operation may optionally pass in a block to be invoked if the primitive fails by appending **`IfFail:`** to the message name and passing in the block as an additional argument. If invoked, the block is passed an error string identifying the nature of the failure (such overflow, divide by zero, or incorrect argument type). For example, **`3 _IntAdd: 'abc' IfFail: [ | :code | ...]`** passes a failure block in addition to the arguments to be added; this block will be invoked with the **`'badTypeError'`** object by the primitive since the arguments to the primitive are not both fixed-precision integers.

Loops are implemented in SELF via the **`_Restart`** primitive. A call to **`_Restart`** transfers control back to the beginning of the scope containing the **`_Restart`** call, creating a loop. The programmer uses a non-local return to break out of such a loop. Programmers can combine **`_Restart`**, non-local returns, and closures to build arbitrary user-defined looping control structures.

SELF uses **`_Restart`** to implement loops explicitly. Other languages such as Scheme instead perform *tail-recursion elimination* to automatically transform recursion into iteration, without introducing an extra language construct explicitly for iteration. Unfortunately, tail-recursion elimination, and more generally *tail-call elimination*, violates the user's execution and debugging model by eliminating activation records that the user expects to see. The Scheme language definition specifies that all tail-recursive calls must be transformed into iterations, which effectively introduces a special language mechanism for looping. In SELF, such looping code is explicit and easy to recognize, since only **`_Restart`** creates a loop. In Scheme, on the other hand, any procedure call that happens to be tail recursive will be transformed into an iterative loop, whether or not the programmer desired or expected it, and identifying when a procedure call is tail-recursive can be tricky.

4.6 An Example: Cartesian and Polar Points

The figure below presents an example collection of SELF objects. The bottom objects are two-dimensional point objects, the left ones represented using cartesian coordinates and the right ones using polar coordinates. The cartesian point traits object is the immediate parent object shared by all cartesian point objects, and defines four methods for interpreting cartesian points in terms of polar coordinates; the polar point traits object does the reverse for polar point objects. The point traits object is the shared ancestor of all point objects, defining general methods for printing and adding points, regardless of coordinate system. The point traits object inherits in turn from the topmost object in the diagram, which defines even more general behavior, such as how to copy objects.



Sending the **`x`** message to the leftmost cartesian point object finds the **`x`** slot immediately. The contents of the slot is the integer **`3`**, which evaluates to itself (it has no associated code), producing **`3`** as the result of the **`x`** message. Sending **`x`** to the rightmost polar point object, however, does not find a matching **`x`** slot immediately. Consequently, the object's parent is searched, finding the **`x`** slot defined in the polar point traits object. That **`x`** slot contains a method that computes a polar point's **`x`** coordinate from its **`rho`** and **`theta`** coordinates. The method gets cloned and executed, producing the floating point result **`1.25`**.

If the **`print`** message were sent to a point object, the **`print`** slot defined in the point traits object would be found. The method contained in the slot prints out the point object in cartesian coordinates. If the point were represented using cartesian coordinates, the **`x`** and **`y`** messages (implicitly sent to **`self`**) would access the corresponding data slots of the cartesian point object. But the **`print`** method works fine even for points represented using polar coordinates: the **`x`** and **`y`** messages would find the conversion methods defined in the polar point traits object to compute the correct **`x`** and **`y`** values.

This example illustrates conventional SELF programming practice. Most SELF code is structured into hierarchies of *traits objects*, abstract objects used to hold behavior to be inherited and refined by child objects. These traits objects play a role similar to one of the roles of classes in class-based languages. Concrete objects inherit from the traits objects, filling in any missing implementation, such as assignable data slots holding object-specific state information. The initial concrete objects are used as prototypical instances of the abstract data type and are cloned to create new instances.

The example also illustrates some of the challenges facing the SELF implementation. The frequency of message sends is very high; in this **`print`** example, nearly every source token corresponds to a message send. Even instance variables are accessed using message sends. Some other challenges facing the implementation that are not illustrated by this short example include user-defined control structures and generic arithmetic support.

Chapter 5 Overview of the Compiler

This chapter introduces the overall design of the SELF compiler. The next section describes the goals of our work. Section 5.2 describes the primary purpose of most of the new techniques developed as part of the SELF compiler: the extraction of representation-level type information. Section 5.3 relates the novel parts of the SELF compiler to the traditional front-end and back-end division of a compiler and outlines topics covered by the next several chapters.

5.1 Goals of Our Work

Our immediate goal is to build an efficient, usable implementation of SELF on stock hardware. However, we are not willing to compromise SELF's pure object model and other expressive features. We want to preserve the illusion of the system directly executing the program as the programmer wrote it, with no user-visible optimizations. This constraint has several consequences that distinguish our work from other optimizing language implementations:

- The programmer must be free to edit any procedure in the system, including the most basic ones such as the definition of `+` for integers and `ifTrue:` for booleans, and provide overriding definitions for other data types when desired.
- The programmer must be able to understand the execution of the program and any errors in the program solely in terms of the source code and the source language constructs. This requirement on the debugging and monitoring interface to the system disallows any internal optimizations that would shatter the illusion of the implementation directly executing the source program as written. Programmers should be unaware of how their programs get compiled or optimized.
- The programmer should be isolated even from the mere fact that the programs are getting compiled at all. No explicit commands to compile a method or program should ever be given, even after programming changes. The programmer just runs the program.

  This illusion of hiding the compiler would break down if the programmer were distracted by mysterious pauses due to compilation, analysis, or optimization. Ideally any pauses incurred by the implementation of the system would be imperceptible, such as on the order of a fraction of a second when in interactive use. Longer running batch programs can be interrupted by longer pauses, as long as the total time of the program is not slowed so much that the programmer becomes aware of the pauses.

Within these constraints on the user-visible semantics of the system, our main objective is excellent run-time performance. We wish to make SELF and other pure object-oriented languages with similar powerful features competitive in performance with traditional non-object-oriented languages such as C and Pascal. In particular, where the SELF program is not taking advantage of object-oriented features, such as in an inner loop that could have been written just as easily in C as in SELF, we want the performance of SELF to be close to the performance of optimized C. If these performance goals are met, many programmers may be able to switch from traditional languages to pure object-oriented languages and begin to reap the benefits afforded by pure object-oriented programming, user-defined control structures, generic arithmetic, and robust primitives.

Other goals are secondary to the constraint of a source-level execution model and the goal of rapid execution. In particular, run-time and compile-time space overheads are less of a concern than run-time speed. Modern computer platforms, especially workstations, are typically equipped with a large amount of physical main memory, and this amount is increasing at a rapid rate. We therefore are willing to use more space than would a straightforward implementation in order to meet our execution speed goals.

When we began this work, there were no techniques available to implement pure object-oriented languages like SELF efficiently without relying on special-purpose hardware support, cheating in the implementation, or diluting the object-oriented model by introducing non-object-oriented constructs into the language. Therefore the main part of our work involved developing and implementing new techniques for implementing object-oriented languages efficiently on stock hardware. These new techniques have enabled us to largely meet our goals both for faithfulness to the source code and run-time execution speed.

Of course, we do not only wish to implement SELF efficiently, but also a larger class of SELF-like languages. Fortunately, the new techniques are not specific to the SELF language. Most object-oriented languages, including C++, Eiffel, Trellis/Owl, Smalltalk, T, and CLOS, would benefit (to varying degrees) from the techniques we have developed. Also, languages with object-oriented subsystems would benefit, including languages supporting some form of generic arithmetic such as Lisp, APL [Ive62, GR84], PostScript [Ado85], and Icon [GG83], languages with logic variables such as Prolog, and languages with futures such as Multilisp [Hal85] and Mul-T [KHM89].

5.2 Overall Approach

This section describes the overall approach to achieving an efficient implementation of SELF and similar languages.

5.2.1 Representation-Level Type Information Is Key

Dynamically-typed object-oriented programming languages historically have run much slower than traditional statically-typed non-object-oriented programming languages. This performance gap is attributable largely to the lack of representation-level type information in the dynamically-typed object-oriented languages. This representation-level information about an object is embodied in the object's *class* in a class-based system. (Section 6.1.1 will describe *maps*, internal implementation structures that embody representation-level type information for prototype-based languages such as SELF.)

If the compiler could infer the classes (or maps) of objects at compile-time, it could eliminate much of the run-time overhead associated with dynamic typing and object orientation. In a dynamically-typed language, the compiler must insert extra run-time type-checking code around type-safe primitives and extra run-time type-casing code in support of generic arithmetic. If the compiler could infer the classes of the arguments to the type-checking primitive, then it could perform the type checks at compile-time rather than run-time. Similarly, if the compiler could infer the classes of the arguments to generic arithmetic primitives, then it could perform the type-casing at compile-time, generating code for a type-specific arithmetic operation instead of the slower generic operation.

In an object-oriented language, the compiler must insert extra run-time message dispatching code to implement dynamic binding of message names to target methods based on the run-time class of the receiver. If the compiler could infer the class of the receiver of a message, then it could perform message lookup at compile-time instead of run-time, replacing the dynamically-bound message with a statically-bound procedure call; the statically-bound call would subsequently be amenable to further optimizations such as inlining (described in Chapter 7) that can significantly boost performance.

5.2.2 Interface-Level Type Declarations Do Not Help

Clearly, the run-time performance of dynamically-typed object-oriented programs could be dramatically improved if the compiler could infer representation-level type information in the form of objects' classes or maps. On the surface, this would seem to imply that statically-typed object-oriented languages, with lots of type information available to the compiler, would have a huge advantage in performance over their dynamically-typed counterparts. Perhaps surprisingly, this advantage is in fact quite small.

In a non-object-oriented language, the type of a variable specifies the representation or implementation of the contents of the variable. This static information corresponds to knowing the exact class of the contents of the variable and hence supports the optimizations described above that reduce the gap between dynamically-typed object-oriented languages and statically-typed non-object-oriented languages. In an object-oriented language with interface-level type declarations, however, the type of a variable specifies only the set of operations that are guaranteed to be implemented by objects stored in the variable. The interface-level type deliberately does *not* specify anything about how the objects stored in the variable will *implement* the operations, in order to maximize the generality and reusability of the code. With only interface-level type information, the compiler cannot perform the optimizations that require representation-level type information. For example, knowing that an object *understands* the **+** message does not help the compiler generate more efficient code for the **+** message; only knowledge about how the object *implements* the **+** message (such as by executing the **+** method for integers) enables optimizations such as inlining that markedly improve performance.*

* Interface-level type information can be useful in special cases given system-wide knowledge. The compiler can examine all the possible implementations in the system that satisfy some interface and sometimes infer useful representation-level type information. For example, if only one object or class implements a particular interface, the static interface-level type information implies representation-level information. These kinds of optimizations are less likely to speed operations on basic data structures such as numbers and collections, however, where many implementations of the same interface are the norm.

5.2.3 Transforming Polymorphic into Monomorphic Code

The lack of static representation-level type information limits the run-time performance of object-oriented languages, whether dynamically-typed or statically-typed. Consequently, our new compilation techniques will strive to infer this missing representation-level type information, so that the compiler can perform optimizations to eliminate the overhead of dynamic typing and object orientation. Once these optimizations have been performed, the task of compiling a dynamically-typed object-oriented program reduces to the task of compiling a traditional statically-typed procedural program.

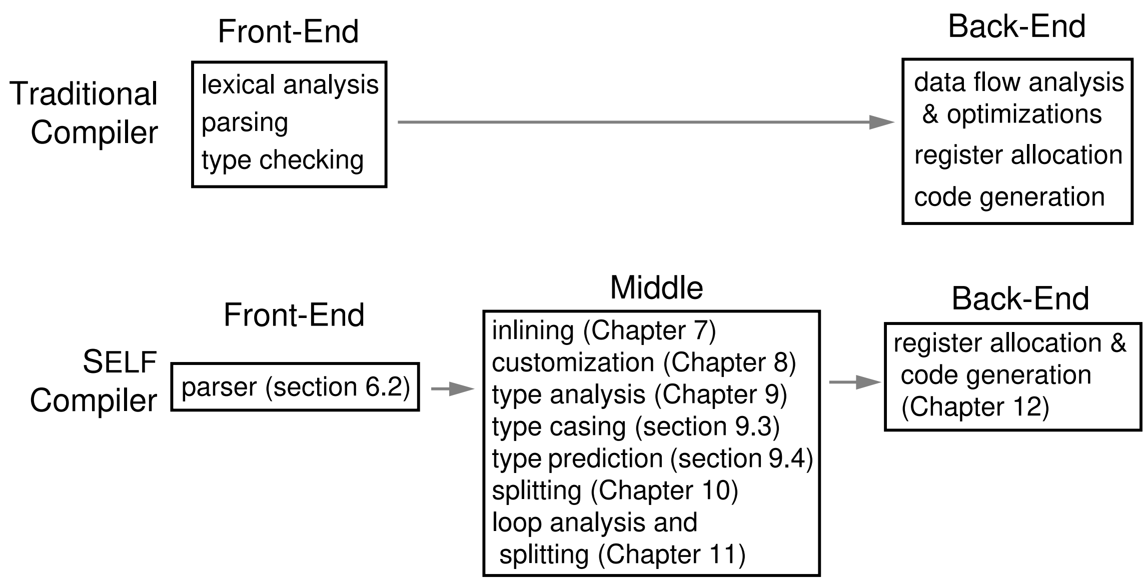
To perform inlining the SELF compiler must prove that the receiver of a message has a single representation, i.e., that the receiver expression is *monomorphic*. In general, however, SELF code is *polymorphic*: expressions may denote values of different representations at different times, and the same source code works fine for all these representations. Such polymorphism is central to the power of object-oriented programming. However, in some cases the SELF program does not exploit the full power of polymorphism. Sometimes a message receiver can be a member of only a single clone family. To exploit such cases, the compiler includes techniques such as *type analysis* (described in Chapter 9) to identify monomorphic expressions and subsequently optimize them.

In most cases, however, the compiler's task is not so easy: most SELF expressions really are potentially polymorphic. Nevertheless, the compiler can frequently optimize even polymorphic messages. The compiler includes several techniques such as *customization* (described in Chapter 8), *type casing* (described in section 9.3), *type prediction* (described in section 9.4), and *splitting* (described in Chapter 10) that can transform some kinds of polymorphic expressions into monomorphic expressions; these monomorphic expressions are then suitable for further optimization. All these techniques work by duplicating code, transforming a single polymorphic expression with N possible representations into N separate monomorphic expressions. Each monomorphic case can be optimized independently; without this separation no optimization would be possible. These techniques for trading away compiled-code space to gain run-time speed form the heart of the SELF compiler and are our key contribution to compilation technology for object-oriented languages.

Since identifying and creating monomorphic sections of code can be fairly time consuming, the SELF compiler seeks to conserve its efforts. In particular, the compiler attempts to compile only those parts of the SELF program that are actually executed. The compiler only performs customization on demand, exploiting SELF's dynamic compilation architecture as described in section 8.2. Additionally, many cases that could arise in principle but rarely arise in practice, such as integer overflows, array accesses out of bounds, or illegally-typed arguments to primitives, are never actually compiled by the SELF compiler, thus saving a lot of compile time and compiled code space and allowing better optimization of the parts of programs that *are* executed. This *lazy compilation of uncommon branches* is described in section 10.5.

5.3 Organization of the Compiler

Traditional compilers are typically divided into a *front-end*, which performs lexical analysis and parsing, and a *back-end*, which performs optimizations and generates code. The SELF parser, described in section 6.2, performs the functions of a traditional front-end by translating SELF source into a byte-coded representation. The SELF compiler performs many of the functions of a traditional back-end; Chapter 12 describes the SELF compiler's version of most of these traditional functions. The bulk of the SELF compiler effort, however, lies in between the two halves of a traditional compiler. This "middle half" of the SELF compiler performs the representation-level type analysis and inlining that bridges the semantic gap between the high-level polymorphic program input to the SELF compiler and the lower-level monomorphic version of the program suitable for the optimizations performed by a traditional compiler back-end.



The next chapter describes the supporting services provided by the rest of the SELF system architecture, including a description of the *map* data structures that convey representation-level type information of objects to the compiler. The "middle half" of the SELF compiler is described in the following several chapters. Chapter 7 describes inlining in more detail. Chapter 8 presents *customization*, one the SELF compiler's important new techniques. Chapter 9 describes *type analysis*, the technique used by the SELF compiler to infer and propagate the representation-level type information through the control flow graph. It also presents *type prediction*, a technique for guessing the types of some objects based on the names of messages and built-in profile information. Chapter 10 describes *splitting*, the primary technique used in the SELF compiler to turn polymorphic pieces of code into multiple monomorphic pieces of code. Chapter 10 also describes *lazy compilation*, a technique for only compiling those parts of methods that the compiler judges to be likely to be executed. Chapter 11 concludes the discussion of the middle end by describing type analysis and splitting in the presence of loops.

Chapter 6 Overall System Architecture

The SELF compiler does not operate in isolation. It is an integral part of the whole SELF system, depending on the facilities provided by the rest of the system and constrained to satisfy requirements imposed by the rest of the system. To place the compiler in context, this chapter describes the overall architecture of the SELF system, focusing on the impact these other parts of the system have on the design of the compiler.* The following several chapters describe the compiler itself.

6.1 Object Storage System

The *object storage system* (also called the *memory system*) represents SELF objects and their relationships. It provides facilities for creating new objects and automatically reclaims the resources consumed by inaccessible objects. It supports modifying objects via programming and for scanning objects to locate all occurrences of certain kinds of references.

Much of the memory system design exploits technology proven in existing high-performance Smalltalk and Lisp systems. For minimal overhead in the common case, the SELF system represents object references using direct tagged pointers, rather than indirectly through an object table as do some Smalltalk systems. An early version of the SELF memory system was documented by Elgin Lee [Lee88]; a more recent version was described in [CUL89].

The following two subsections describe techniques for efficient object storage systems pioneered by the SELF implementation. Subsection 6.1.3 describes constraints placed on the compiler by SELF's garbage collection algorithm. Appendix A describes object formats in detail.

6.1.1 Maps

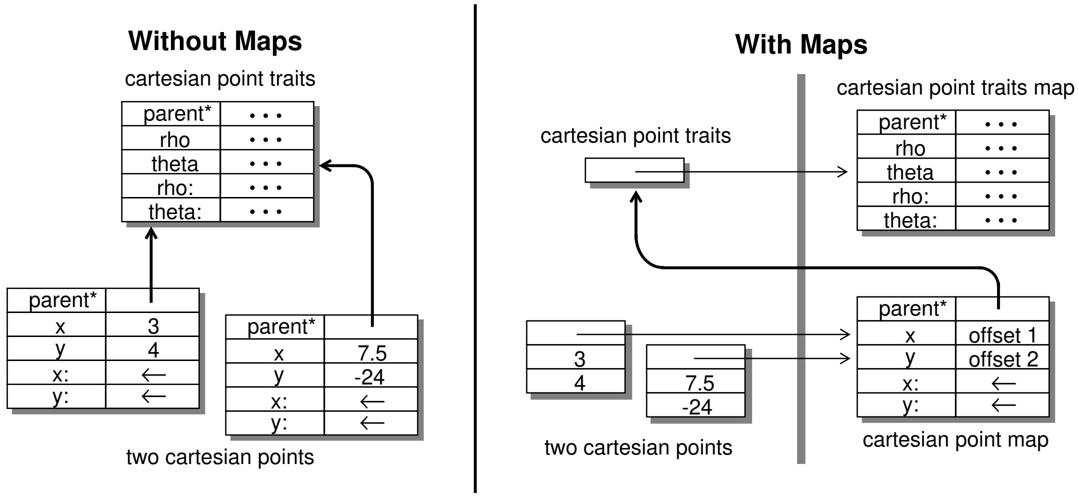
In traditional class-based languages, a class object contains the format (names and locations of the instance variables), methods, and superclass information for all its instances; the instances contain only the values of their instance variables and a pointer to the shared class object. Since SELF uses a prototype model, each object is self-sufficient, defining its own format, behavior, and inheritance. A straightforward implementation of SELF would therefore represent both the class-like format, method, and inheritance information and the instance-like state information in each SELF object. This representation would consume at least twice as much space as for a traditional class-based language.

Fortunately, the storage efficiency of classes can be regained even in SELF's prototype object model by observing that few SELF objects have totally unique format and behavior. Almost all objects are created by cloning some other object and then possibly modifying the values of the assignable slots. Wholesale changes in the format or inheritance of an object, such as those induced by the programmer, can be accomplished only by invoking special primitives. Therefore, a prototype and the objects cloned from it, identical in every way except for the values of their assignable slots, form what we call a *clone family.*

The SELF implementation uses *maps* to represent members of a clone family efficiently. In the SELF object storage system, objects are represented by the values of their assignable slots, if any, and a pointer to the object's map; the map is shared by all members of the object's clone family. For each slot in the object, the map contains the name of the slot, whether the slot is a parent slot, and either the offset within the object of the slot's contents (if it is an assignable slot) or the slot's contents itself (if it is a constant slot, such as a non-assignable parent slot). If the object has code (i.e., is

* Many of the techniques described in this chapter were designed by the SELF group as a whole and implemented by various members of the SELF group, including Elgin Lee, Urs Hölzle, David Ungar, and the author, and so should not be viewed as contributions solely attributable to the author.

a method), the map stores a pointer to a SELF byte code object representing the source code of the method (byte code objects are described further in section 6.2).



Maps are immutable so that they may be freely shared by objects in the same clone family. However, when the user changes the format of an object or the value of one of an object's constant slots, the map no longer applies to the object. In this case, a new map is created for the changed object, thus starting a new clone family. The old map still applies to any other members of the original clone family. If no other members exist (i.e., the modified object was the only member of its clone family), the old map will be garbage-collected later automatically.

From the implementation point of view, maps look much like classes, and achieve the same sorts of space savings for shared data. In addition, the map of an object conveys its static properties to the SELF compiler, much as does an instance's class in a class-based language. Nevertheless, maps are completely invisible to the SELF programmer. Programmers still operate in a world populated by self-sufficient objects that in principle could each be unique. The implementation simply is optimizing representation and execution based on existing usage patterns, i.e., the presence of clone families.

6.1.2 Segregation

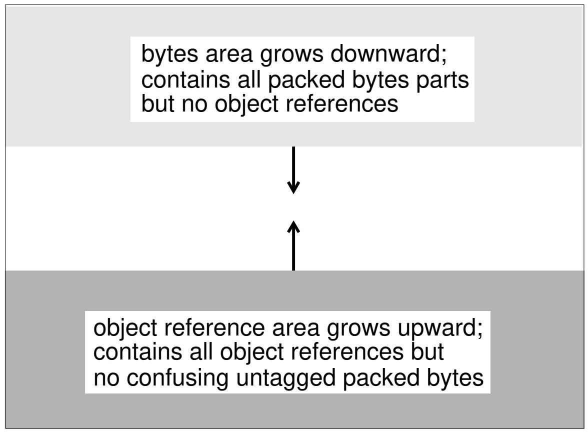
The memory system frequently scans all object references for those that meet some criterion:

- The scavenger scans all objects for references to objects in from-space as part of garbage collection.
- The programming primitives have to find and redirect all references to an object if its size changes and it has to be moved.
- The browser may need to search all objects for those that contain a reference to a particular object that interests the SELF user (i.e., following "backpointers").

To support these and other functions, the SELF implementation has been designed for rapid scanning of object references.

Ideally the system could just sweep through all of memory word-by-word to find object references matching the desired criterion. Unfortunately, since the elements of byte arrays are represented using packed bytes rather than tagged words (see Appendix A), byte array elements may *masquerade* as object references if byte arrays are scanned blindly. Most systems handle this problem by scanning the heap object-by-object rather than word-by-word. To scan an object, the system examines the header of the object to locate the part of the object containing object references and skip the part containing packed bytes (or other non-pointer data that could masquerade as a pointer). The pointer parts of the object are scanned, while the other parts are ignored. The scanner then proceeds to the next object. This procedure avoids the problems associated with scanning byte arrays, but slows down the scan with the overhead to parse object headers and compute object lengths.

The SELF system avoids the problems associated with scanning byte arrays without degrading the object reference scanning speed by *segregating* the packed untagged bytes from the other SELF objects. Each Generation Scavenging memory space (described more in the next section) is divided into two areas, one for the bytes parts of byte arrays and one for the rest of the data (including the object-reference part of byte arrays).* To scan all object references, only the object reference area of each space needs to be scanned (ignoring any scans for references to integers in the range [0..255], which require special support but almost never occur). This optimization speeds scans by eliminating the need to parse object headers.



A Segregated SELF Memory Space

To avoid slowing the tight scanning loop with an explicit end-of-space check, the word after the end of the space is temporarily replaced with a *sentinel* reference that matches the scanning criterion. This enables the scanner to check for the end of the space only on a matching reference, instead of on every word. Early measurements on a 68020-based Sun-3/50 showed that the SELF system scanned memory at the rate of approximately 3 megabytes per second. Measurements of the fastest Smalltalk-80 implementation on the same machine, ParcPlace Smalltalk-80, indicated a scanning speed for non-segregated memory spaces of only 1.6 megabytes per second. Current measurements indicate a scanning speed for SELF of 12 megabytes per second on a SPARC-based Sun-4/260 workstation.

For some kinds of scans, such as finding all objects that refer to a particular object (following backpointers), the scanner needs to find the objects that *contain* a matching reference, rather than the reference itself. In a system that scans object-by-object, this task is no more difficult than searching for references. However, in a system that scans word-by-word it may be difficult to locate the beginning of the object containing the matching reference. To support these kinds of scans without resorting to object-by-object scanning the SELF system specially tags the first header word of every object (called the *mark* word) to identify the beginning of the object. Thus, to find all objects containing a particular reference, the scanner proceeds normally, searching for matching references. Once a reference is found, the scanner locates the object containing the reference by scanning backwards to the object's mark word and then converting the mark word's address into an object reference by adding the right tag bits to the address.

6.1.3 Garbage Collection

The SELF implementation reclaims inaccessible objects using a version of *Generation Scavenging* [Ung84, Ung87] with *demographic feedback-mediated tenuring* [UJ88], augmented with a traditional mark/sweep collector to reclaim tenured garbage. The SELF heap is currently configured using a 200KB *eden* memory space for newly-allocated objects, a pair of 200KB *survivor* memory spaces for objects that have survived at least one scavenge but have not yet been tenured, and a 5MB *old* space for tenured objects.

The implementation of this algorithm imposes certain constraints on the compiler. The run-time system must be able to locate all object references embedded in compiled instructions whenever a scavenge or garbage collection occurs. Conversely, the garbage collector must be protected from examining any data values which could falsely masquerade as an object reference. In particular, the SELF compiler does not produce *derived pointers* to the interior of an object. This restriction allows the garbage collector to assume that all data tagged as an object reference really points to the

* This design is slightly different from segregation as described in [Lee88] and [CUL89], in which byte arrays were stored completely in the bytes area. The design was changed so that a byte array could have user-defined references to other objects in addition to its array of bytes.

beginning of an object in the heap, thus speeding the collector at some hopefully small cost in run-time efficiency for certain kinds of programs (notably those that iterate through arrays).

Generation Scavenging requires the compiler to generate *store checks*. Each store to a data slot in the heap needs to be checked to see whether it is creating a reference from an object in old-space to an object in new-space; all such references need to be recorded in a special table for use at scavenges. The current SELF implementation uses a *card marking* scheme similar to that used by some other systems [WM89].* Each card corresponds to a region of a SELF memory space (currently 128 bytes long), and records whether any of the words in the corresponding region contain pointers to new space. Whenever the compiler generates a store into an object that might be in old-space to an object that might be in new-space, the compiler also must generate code to mark the appropriate card for the modified data word.

Since stores into objects need to be fast, the SELF compiler attempts to generate code that is as fast as possible. If the compiler can prove that the target of the stored reference is an integer or a floating point immediate value (i.e., not a pointer), then no store check needs to be generated, since such a store does not create a reference from old- to new-space. Otherwise the compiler generates the following sequence (given in SPARC assembly syntax):

```
st   %dest, [%source + offset] ; do the store
add  %source, offset, %temp ; compute address of modified word
sra  %temp, log_base_2(card_size), %temp ; compute card index
stb  %g0, [%card_base + %temp] ; mark card by zeroing
```

The compiler generates code to shift the address of the modified data slot right by a number of bits equal to the $\log_2$ of the card size, adds it to the contents of a dedicated global register on the SPARC (a global variable on the Motorola 680x0), and zeros the byte at that address. Our system uses a whole byte per card, even though only a single bit is required to record whether the card is marked, since the store checking code would be slowed by the bit manipulation operations. Space for cards is allocated even for objects in new space so that the store checking code doesn't have to check to see whether or not the object being modified is in old space; the scavenger simply ignores the cards for objects in new space. With 128-byte cards, the space required to store the card mark bytes is less than 1% of the total space of the SELF heap, adding less than 45KB for our standard heap size of 5.6MB. The amount of space overhead can be varied against the cost of scanning a card at scavenging time by changing the size of the cards.

The dedicated global register named **`%card_base`** in the code above represents the base address of the array of bytes for the cards. It is initialized so that the address of the lowest memory word in the heap, shifted right the appropriate number of bits, and added to the global register contents yields the address of the first byte in the array of cards:

```
%card_base = &cards[0] - (&heap[0] >> log_base_2(card_size))
```

This store checking design imposes a relatively small overhead of 3 instructions for each store into memory to support generation scavenging; we are not aware of any other store checking designs that impose less overhead.

6.2 The Parser

To minimize parsing overhead, textual SELF programs are parsed once when entered into the system, generating SELF-level *byte code* objects, much like Smalltalk-80 **`CompiledMethod`** instances [GR83]. Each method object represents its source code by storing a reference to the pre-parsed byte code object in the method's map; all cloned invocations of the method thus share the same byte code object since they share the same map. A byte code object contains a byte array holding the byte codes for the source and an object array holding the message names and object literals used in the source; the byte code object also records the original unparsed source and the file name and line number where the method was defined for user-interface purposes.

* Earlier SELF implementations including that described in [Lee88] used a more traditional *remembered set* to record old objects containing pointers to new objects.

Each byte code in the byte array represents a single byte-sized virtual machine instruction and is divided into two parts: a 3-bit opcode and a 5-bit object array index. The opcodes used to represent SELF programs are the following:

0: INDEX-EXTENSION <index extension>
 extend the next index by prepending this index extension

1: SELF
 push **`self`** onto the execution stack

2: LITERAL <value index>
 push a literal value onto the execution stack

3: NON-LOCAL RETURN
 execute a non-local return from the lexically-enclosing method activation

4: DIRECTEE <parent name index>
 direct the next message send (which must be a resend) to the named parent

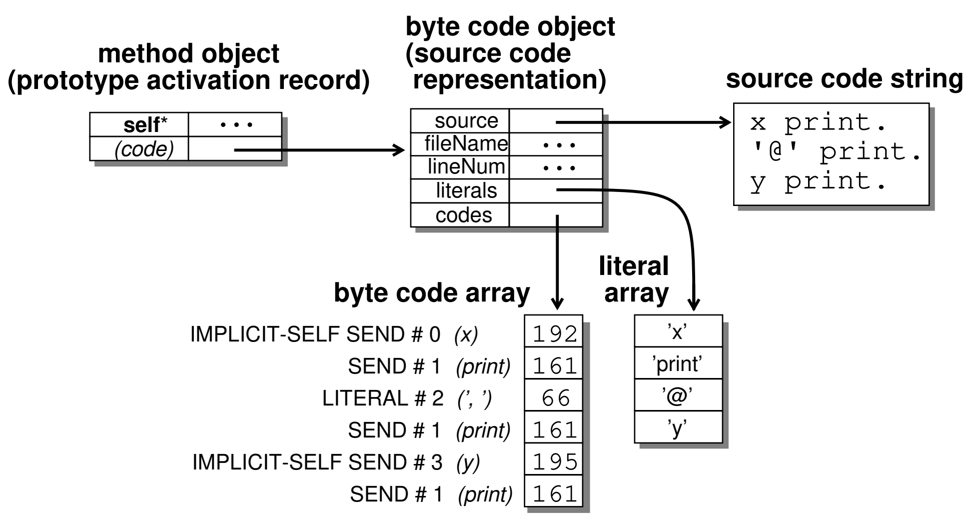
5: SEND <message name index>
 send a message, popping the receiver and arguments off the execution stack and pushing the result

6: IMPLICIT SELF SEND <message name index>
 send a message to (implicit) **`self`**, popping the arguments off the execution stack and pushing the result; begin the message lookup with the current activation record

7: RESEND <message name index>
 send a message to **`self`** but with the lookup beginning with the parents of the object containing the sending method, popping the arguments off the execution stack and pushing the result; like a **`super`** send in Smalltalk[*]

These opcodes are specified as if for direct evaluation by a stack-oriented interpreter; in reality, the SELF system dynamically compiles machine code that simulates such an interpreter. The index specified by several of the opcodes is an index into the byte code object's accompanying object array. The 5-bit index allows the first 32 message names and literals to be referenced directly; indices larger than 32 are constructed using extra INDEX-EXTENSION instructions. Since primitive operations are invoked just like normal messages, albeit with a leading underscore on the message name, the normal SEND byte codes can be used to represent all primitive operation invocations, simplifying the byte codes and facilitating extensions to the set of available primitive operations.

For example, the following diagram depicts the method object and associated byte code object for the point **`print`** method originally presented in section 4.1. The top-left object is the prototype activation record, containing placeholders for the local slots of the method (in this case, just the **`self`** slot) plus a reference to the byte code object representing the source code (actually stored in the method's map). The byte code object contains a byte array for the byte codes themselves, and a separate object array for the constants and message names used in the source code.



[*] Resends and directed resends are described in detail in [CUCH91] and [HCC+91].

6.3 The Run-Time System

6.3.1 Stacks

A running SELF program is a collection of lightweight processes, each process sharing the SELF heap address space but with its own set of activation records. As with most traditional language implementations, these activation records are implemented as a stack of frames, linked by stack pointers and frame pointers. The machine hardware provides support for efficiently managing stack frames. For example, the Motorola 680x0 architectures provide special instructions such as **`link`**, **`jsr`**, and **`movem`** for managing linked stacks of activation records and stack pointers [Mot85], and the Sun SPARC architecture provides hardware register windows to support fast procedure calls and returns with little register saving and restoring overhead [Sun91].

Garbage collection places some requirements on the design and implementation of the run-time system and the compiler. The garbage collector must be able to locate all object references stored in registers or on the stack. In the current SELF implementation, the compiler places a *saved locations mask* word at a fixed offset from each call instruction that could trigger a scavenge (i.e., all message sends and many primitives), identifying to the scavenger which registers and stack locations may contain tagged heap object references and should be scanned. On the SPARC, any of the 8 incoming and 8 local registers can contain valid object references (the 8 outgoing registers are handled by the next frame, which calls them its incoming registers, and none of the 8 global registers contain valid object references), thus requiring 16 bits of the saved locations mask word to mark which registers to scan. The remaining 16 bits of the 32-bit mask word indicate which of the first 16 stack temporary locations need to be scanned. Any additional stack temporaries are assumed to always need scanning; the compiler zeros these excess temporaries upon entering the method so that their contents will always be acceptable to the garbage collector.* This mask-word-based design allows the compiler more freedom in allocating data to registers than the alternative approach of fixing which registers can contain only object references and which can contain only non-pointer data. It does, however, slow scavenging with the overhead to extract and interpret the mask word for every stack frame; fortunately, we have not noticed this potential problem to be a performance bottleneck in practice.

In keeping with SELF's robust implementation, stack overflow is detected and reported via a signal to the process-controlling SELF code. In the current implementation, stack overflow is detected through an explicit check at the beginning of each compiled method that requires a new stack frame. On the SPARC, a dedicated global register maintains the current stack limit. On entrance to a compiled method, this global register is compared against the current stack pointer (also a register), and if the current stack pointer is past the stack limit, the stack overflow code is invoked. The 680x0 system is similar, except that the current stack limit is stored in a global variable in memory rather than a dedicated register. This stack overflow detection imposes a small run-time overhead to check for overflow on every method invocation.**

This polling for stack overflows at the beginning of methods actually is used for much more in the SELF system: it also is used for signal handling, keyboard interrupt handling, and memory scavenging requests. Whenever a running SELF process needs to be interrupted, either because a signal has arrived or a scavenge needs to be performed, the current stack limit is reset back to the base of the stack. This causes execution to be interrupted at the next message send point. When the stack overflow handler is invoked, it first checks to see what caused the "overflow": a pending signal, a scavenge request, or a real stack overflow, and branches to the appropriate handler.

Unfortunately, this polling approach to handling interrupts does not work for loops in which all message sends have been inlined away. To support interrupts and scavenges in these loops, the compiler generates extra code to check the stack limit value against the current stack pointer at the end of each loop body (at the **`_Restart`** primitive call, described in section 4.1). This ensures that interrupt handlers are always invoked relatively quickly after an interrupt is posted.

* Earlier memory system implementations only used the saved locations mask word for registers but not for stack locations. This forced methods with stack temporaries to execute a lengthy prologue to zero out all the stack locations. The new design avoids this overhead in nearly all cases arising in practice.

** An alternate implementation could avoid this run-time overhead by using hardware page protection to protect the memory page at the upper limit of the stack. If this page were accessed, the run-time system would interpret the subsequent memory access trap as a stack overflow error.

Interrupting SELF programs only at message send boundaries allows the compiler more freedom in generating code. The execution environment (the stack and registers) needs to be in a consistent state only when an interrupt could be caught (such as at message send boundaries) rather than at each instruction boundary. Debugging information to describe the state of execution, such as the saved locations mask word and the mapping from variable names to register assignments (described in section 13.1), need only be generated for interruption points such as message sends rather than for every machine instruction.

6.3.2 Blocks

In the current SELF implementation, blocks cannot outlive their lexically-enclosing scope. A block may only be passed down to the called routine, such as when the block is part of a user-defined control structure or is an exception handler.* This restriction is included so that activation records may be stack-allocated without additional special implementation techniques. Since blocks cannot outlive their lexically-enclosing scope, the contents of the implicit lexical parent slot of the block can be represented as a simple untagged pointer to the stack frame representing the lexically-enclosing activation record. Since all stack frames are aligned on a double-word (8-byte) boundary on the SPARC and at least a half-word (2-byte) boundary on the 680x0, the untagged address of the stack frame can be used in the representation of a block without fear of unfortunate interactions with the garbage collector.

The SELF implementation does not prevent a block object from being returned by its lexically-enclosing scope or stored in a long-lived heap data structure, but instead disallows the block's **`value`** method from being invoked after the lexically-enclosing scope has returned; such a "zombie" block is termed a *non-LIFO block*, since its lifetime does not follow the normal last-in-first-out (LIFO) stack discipline. To enforce this restriction, the compiler generates code to *zap* each block when its lexically-enclosing scope returns by zeroing out the block's frame pointer. Subsequent up-level accesses through a zapped block's null frame pointer cause segmentation faults which are caught by the SELF implementation, interpreted as non-LIFO block invocation errors, and signalled back to the SELF program.

6.4 Summary

The SELF memory system provides several important facilities for the compiler. Maps capture essential similarities among clone families, embodying the representation-level type information crucial to the compiler's optimizations. However, the garbage collector and the run-time system place constraints on the design and implementation of the compiler. Some of these constraints impose extra overhead on the run-time execution of programs (such as the current design of polling for interrupts) while others restrict the possible optimizations included in the compiler (such as disallowing derived pointers that could confuse the garbage collector). Fortunately, these restrictions are not too severe, and some parts of the overall system architecture ease the burden on the compiler, such as limiting interrupts to well-defined points in the compiled code and placing no restrictions on the allocation of pointer and non-pointer data to registers.

* Because they cannot be returned upwards, SELF blocks are not as powerful as blocks in Smalltalk or closures in Scheme. We do not miss this power in our SELF programming, since we can use heap-allocated objects created from in-line object literals to hold the long-lived state shared by the lexically-enclosing method and the nested blocks, and allow this in-line object to be returned upwards. Even so, this restriction on the lifetime of blocks in some ways reduces the elegance of the language, and some future SELF implementation may relax this restriction; we believe this can be accomplished without seriously degrading performance.

Chapter 7 Inlining

This chapter describes inlining of methods and primitive operations. Many people have researched the problem of improving the performance of procedure calls through inlining, as described in section 3.4.6. In this chapter we describe the approach taken in the SELF compiler and detail the heuristics used to guide inlining automatically.

7.1 Message Inlining

The SELF compiler reduces the overhead of pure object-orientation and user-defined control structures primarily through *message inlining*. If the compiler can infer the exact type (i.e., map) of the receiver of a message, then the compiler can perform the message lookup at compile-time instead of at run-time. If this lookup is successful (as it will be in the absence of dynamic inheritance and message lookup errors in the program), then the compiler can *statically bind* the message send to the invoked method, thereby reducing the message to a normal procedure call. Static binding improves performance, since a direct procedure call is faster than a dynamically-dispatched message send, but it does not help to reduce the high call frequency. Once a message send is statically bound, however, the compiler may elect to *inline* a copy of the target of the message into the caller, eliminating the call entirely.

To illustrate, on the SPARC the call and return sequence for a dynamically-bound message send takes a minimum of 11 cycles in our current SELF implementation (ignoring additional overhead such as LRU compiled method reclamation support described in section 8.2.3 and interrupt checking described in section 6.3.1). The call and return sequence for a statically-bound procedure call, on the other hand, takes just 4 cycles. But an inlined call takes *at most* 0 cycles, and usually takes even less, since the inlined body of the called method can be further optimized for the particular context of the call site. For example, any register moves added to get the arguments to the message in the right locations as dictated by the calling conventions can be avoided by inlining, and optimizations such as common subexpression elimination can be performed across what before inlining was a call boundary. Inlining is so good that in many situations the additional benefits derived from inlining are greater than the initial benefits derived from static binding. As reported in section 14.3, without inlining SELF would run between 4 and 160 times slower.

The effect of message inlining depends on the contents of the slot that is evaluated as a result of the statically-bound message:

- If the slot contains a method, the compiler can inline-expand the body of the method at the call site, if the method is short enough and not already inlined (i.e., is not a recursive call).
- If the slot contains a block **`value`** method, the compiler can inline-expand the body of the block **`value`** method at the call site, if it is short enough. If after inlining there are no remaining uses of the block object, the compiler can optimize away the code that would have created the block object at run-time.
- If the slot is a constant data slot (i.e., the slot contains a normal object without code and there is no corresponding assignment slot), the compiler can replace the message send with the value of the slot; the message then acts like a compile-time constant expression. These kinds of messages typically access what in other languages would be special constant identifiers or global variables, such as **`true`** and **`rectangle`**.
- If the slot is an assignable data slot (i.e., the slot contains a normal object without code and there is a corresponding assignment slot), the compiler can replace the message send with code that fetches the contents of the slot, such as a load instruction. These kinds of messages typically access what in other languages would be instance variables or class variables.
- If the slot is an assignment slot (i.e., the slot contains the assignment primitive method), the compiler can replace the message send with code that updates the contents of the corresponding data slot, such as a store instruction. These kinds of messages typically assign to what in other languages would be instance variables or class variables.

The next few subsections discuss interesting aspects of compile-time message lookup and inlining in the SELF compiler.

7.1.1 Assignable versus Constant Slots

As described above, the SELF compiler treats constant and assignable data slots differently. The compiler inlines the *contents* of a constant (non-assignable) data slot, but only inlines the *offset* or *access path* of an assignable data slot. This reflects the compiler's expectations about what will remain constant during the execution of programs and what may change frequently at run-time. The only object mutations that running programs are expected to perform are assignments to assignable data slots. Object formats and the contents of non-assignable data slots and method slots can only be changed using special programming primitives, which are not expected to be invoked frequently during the execution of programs. Accordingly, the compiler makes different compile-time/run-time trade-off decisions for rarely-changing information and for frequently-changing information:

- Rarely-changing information (i.e., the formats of objects, the definitions of methods, and the contents of non-assignable slots) is embedded in compiled code. This makes normal programs as fast as possible but incurs a significant recompilation cost if the information does change.
- Frequently-changing information (i.e., the contents of assignable slots) is not embedded in compiled code. This allows programs to change the information without cost but may sacrifice some opportunities for optimization.

Assignable parent slots are particularly vexing to the compiler. Most parent slots are non-assignable data slots. Consequently, the compiler feels free to assume their contents will not change at run-time. This assumption enables compile-time message lookup (the result of which depends on the contents of the parents searched as part of the lookup), which in turn enables static binding and inlining, the keys to good run-time performance.

In contrast, encountering an assignable parent slot blocks compile-time lookup even if the receiver type is known. Since the parent slot is assignable, the compiler assumes that the parent is likely to change at run-time, potentially invalidating any compile-time message lookup results. Consequently, the current SELF compiler does not statically bind or inline a message if an assignable parent is encountered during compile-time message lookup. This decision allows assignable parent slots to be changed relatively cheaply, but imposes a significant cost to the use of dynamic inheritance by slowing all messages looked-up through an assignable parent. We are currently exploring techniques that might reduce this cost.

In summary, the SELF compiler makes heavy use of the distinction between assignable and constant slots, and exploits the fact that whether or not a slot is assignable can be determined by examining only the object containing the data slot. If all slots in SELF were implicitly assignable, or if a data slot could be made assignable by adding a corresponding assignment slot in a child object (as was the case in an early design of SELF), then the compiler would no longer be able to treat most parent slots as unchanging. In the absence of appropriate new techniques, these alternative language designs would have serious performance problems.

These issues are specific to SELF's reductionist object model, however. The facilities supported in SELF using constant slots are supported in other languages using special language mechanisms. For example, Smalltalk uses different language mechanisms for instance variables, global variables, superclass links, and methods, while SELF uses slots for all of them. Therefore, the situations in which the SELF compiler takes advantage of a slot being constant correspond to the situations in which a Smalltalk compiler using similar techniques could assume a feature was constant because of the semantics of that language feature. Dynamic inheritance is also a SELF-specific problem, arising directly out of SELF's reductionist and orthogonal object model; we are aware of no other language which has a similar implementation issue.

7.1.2 Heuristics for Method Inlining

Whenever a message is statically-bound to a single target method, the compiler can inline the message to speed execution. However, unrestricted inlining can also drastically increase compiled code space requirements and slow compilation. Consequently, the compiler should not inline methods when the costs of inlining are too great. The compiler therefore must decide when presented with a statically-bound message whether or not to inline the message.

In the SELF system, the compiler is responsible for making inlining decisions; SELF programmers are not involved in or even aware of inlining. The compiler uses heuristics to balance the benefits of inlining against its costs in compiled code space and compilation time, electing to inline messages whose benefits significantly outweigh their costs. Of course, the compiler must avoid performing costly analysis to decide whether a method should be inlined, since that itself would significantly increase compilation time. The next two subsections describe the principle heuristics used by the SELF compiler in deciding whether to inline a message: method length checks and recursive call checks.

7.1.2.1 Length Checks

Ideally, to calculate the benefits and costs of inlining a method, the compiler would inline the method, optimize it in the context of the call, and then calculate the performance improvement attributable to inlining and the extra compile time and compiled code space costs of the inlined version. The compiler would then have enough accurate information upon which to base its inlining decision. If in the end the compiler decided not to inline the message, the compiler would back out of its earlier decision, reverting the control flow graph to its state before inlining. Unfortunately, compilation time could not be returned by backing out of an unwise inlining decision, and so this ideal method is too impractical to use directly.

The compiler approximates this ideal approach by calculating the *length* of the target method, and inlining the method only if the method is shorter than a built-in length threshold value; the compiler always inlines statically-bound messages that access constant data slots, assignable data slots, and assignment slots, since the inlined code (e.g., a load or store instruction) is frequently smaller than the original message send. This approach seeks to predict the compile time and compiled code space costs of inlining the method from the definition of the method at a fraction of the cost of the ideal method. If the length calculation is reasonably accurate at predicting which methods should be inlined and which should not, then this approach should achieve run-time performance results similar to those produced by the ideal approach with similar costs in compiled code space, but with little cost in compile time.

The formula for calculating a method's length plays a central role in deciding whether to inline a method, and so developing a good formula is extremely important to the effectiveness of the compiler. The length calculations in the SELF compiler have evolved over time, we hope becoming more and more effective at distinguishing "good" methods to inline from "bad" methods to inline.

To a first approximation the compiler calculates a method's length by counting its SEND, IMPLICIT SELF SEND, and RESEND byte codes. This length metric assumes that the number of sends is a good measure of the cost in terms of compiled code space and compile time of inlining the message, and that LITERAL and other administrative byte codes are relatively free. While sounding reasonable at first, this assumption has at least two glaring problems.

One problem is that it assumes that all sends are equally costly, since it assigns them all equal weight. This assumption is grossly inaccurate when considering sends that access local variables (recall that SELF uses implicit self messages to access local variables, as described in section 4.4) and sends that access data slots such as "instance variable" accesses. To improve the accuracy of the length calculation, the SELF compiler excludes from the length count IMPLICIT SELF SEND byte codes that access local variables or data slots in the receiver; the compiler does not "penalize" a method for accessing local variables or instance variables. The compiler might also reasonably exclude SEND byte codes that would access data slots in ancestors of the receiver (such as sends that would access the SELF equivalent of class variables or globals), but the cost of checking whether a SEND byte code accesses such a data slot would be relatively high (involving at least a call to the compile-time message lookup system), and so the current SELF compiler does not perform these additional checks. However, the increase in accuracy possible by performing these additional checks may someday be deemed important enough to outweigh the additional cost in compile time.

Excluding messages that only access local and instance variables greatly reduces the spread of cost for messages. Even so, there remains a fairly significant range of costs for the messages that are left. A future compiler might include additional heuristics that use the name of a message as an indicator of its cost. For example, a message like **`+`**, **`ifTrue:`**, or **`at:`** is probably cheaper to compile (and more important to optimize) than a message like **`initializeUserInterface`** which is unknown to the compiler. The compiler could reflect this expectation by incrementing the length count for an unknown message by more than for a "recognized" message such as **`+`** or **`ifTrue:`**. This distinction would mesh well with static type prediction, to be described in section 9.4.

The original assumption that LITERAL byte codes are free compared to SEND byte codes also frequently is mistaken when the literal in question is a block. If the block itself gets inlined (and in most cases the compiler will work hard to inline blocks so that it can optimize away the block creation code), then the cost of compiling the method will include the cost of compiling the block (plus the cost of compiling any inlined methods between the outer method and the block). To correct this deficiency, the SELF compiler adds the length of any nested blocks to the length of a method. This rule errs on the side of not inlining a method with nested blocks when inlining would have been reasonable, possibly reducing run-time performance, but is much better than erring in the other direction (inlining methods which should not be inlined) and possibly drastically lengthening compile times. One exception to the nested block rule is that failure blocks (blocks passed as the **`IfFail:`** argument to a primitive) are not added to a method's total length;

techniques such as lazy compilation of uncommon cases, described in section 10.5, ensure that failure blocks are almost never inlined.

Since optimizing away block creations is so important to good performance, the compiler uses more generous inlining length cut-offs for methods which either are block **`value`** methods themselves or are methods which take blocks as arguments (such as methods in a user-defined control structure).

- If the method is a block **`value`** method, the compiler uses a high length threshold (currently 75), intended to inline all reasonable block **`value`** methods.
- If the method is being passed a block as one of its arguments, the compiler uses a medium length threshold (currently 8). This seeks to preferentially inline methods that are part of user-defined control structures, in the hope that all uses of the block can be inlined away and the block creation code can then be removed.
- Otherwise, the compiler uses a low threshold (currently 5).

Additionally, along uncommon branches of the control flow graph, the compiler uses drastically-reduced length thresholds (currently 1) to prevent inlining of methods where the run-time pay-off is expected to be low. Since a message in an uncommon branch will probably never be sent at all, any effort expended to optimize it is likely to be wasted.

These heuristics do a reasonable job for most methods in inlining the right messages. For example, most common user-defined control structures such as **`for`** loops get inlined down to the primitive operations, enabling the SELF compiler to generate code similar to that generated by a traditional compiler. Unfortunately, these length calculation heuristics sometimes make serious mistakes, leading to overly long compile pauses when the compiler underestimates the cost of inlining several messages, and missed opportunities for optimization when the compiler errs in the opposite direction. Improved heuristics to support automatic inlining thus remain a promising area for future research.

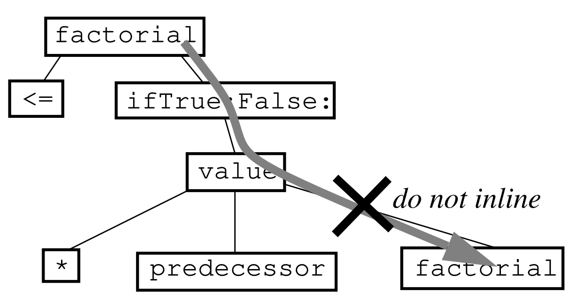
7.1.2.2 Recursion Checks

The compiler must not inline forever when analyzing a recursive routine. For example, when given the factorial function

```
factorial = ( <= 1 ifTrue: 1 False: [ * predecessor factorial ] )*
```

the compiler should not inline the recursive call to **`factorial`** even though it can infer that the type of the receiver to factorial is an integer. This requires the compiler to include some mechanism to prevent unbounded inlining of recursive methods.

One approach that would be sufficient to prevent unbounded inlining would have the compiler record the internal call graph of inlined methods, and not inline the same method twice in any particular path from the root to the leaves of the call graph; the recursive message send could be statically bound but not inlined. Since the internal call graph data structure is already maintained by the compiler to support source-level debugging (as will be described in section 13.1), this would be easy to include as part of recursion testing.



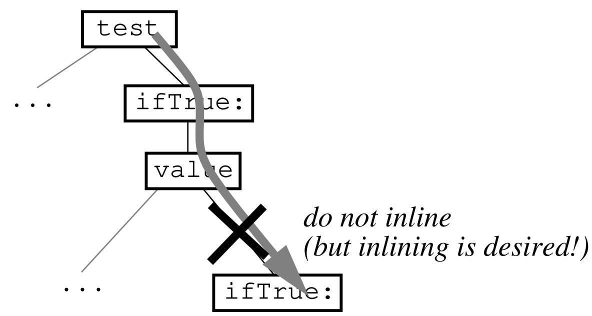
[*] This code may look strange to the SELF novice because all the **`self`** keywords that would be present in the Smalltalk-80 version have been elided using SELF's implicit self message syntax. Thus the receivers of the **`<=`**, **`*`**, and **`predecessor`** messages are all implicitly **`self`**.

A more precise approach would also check to see whether the type of the receiver was the same in both calls of the method, and allow the method to be inlined as long as the receiver maps were different. Since there are only a finite number of maps in the system, and new maps are not created during compilation, this receiver map check would be adequate to prevent unbounded inlining. In practice, only a few different maps ever are encountered in one run of the compiler, so this check would place quite a tight bound on the amount of "recursive" inlining allowed.

Unfortunately, even this more precise recursion check is too restrictive to handle user-defined control structures and blocks as desired. Consider the following simple code fragment:

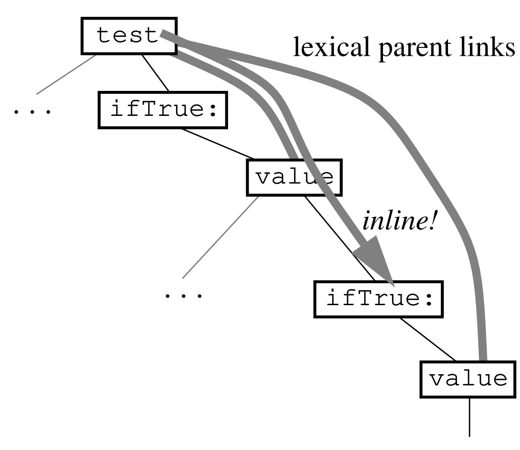
```
test = ( ... ifTrue: [ ... ifTrue: [ ... ] ] )
```

If this were C code, a C compiler would end up "inlining" both **`if`** statements. Similarly, we would like both calls of **`ifTrue:`** to be inlined, but the recursion check described above would disallow it, since the second call to **`ifTrue:`** occurs within an existing call to **`ifTrue:`**.



This example is typical of many similar situations in the implementations of common user-defined control structures such as **`to:Do:`** and **`whileTrue:`**, and some solution must be found in order to achieve good run-time performance.

The SELF compiler solves this problem by augmenting the approach described above with special treatment of lexically-scoped block **`value`** methods. When traversing the call stack, searching for pre-existing invocations of some method, the compiler follows the lexical parent link for block **`value`** methods instead of the dynamic link as with normal methods. This revised rule allows the example to be inlined as desired (since the outer **`ifTrue:`** method is skipped when following the lexical chain of the nested **`ifTrue:`** message), but still prevents unbounded recursive inlining since only a finite number of "recursive" invocations of a message can be made, one per lexical scope.

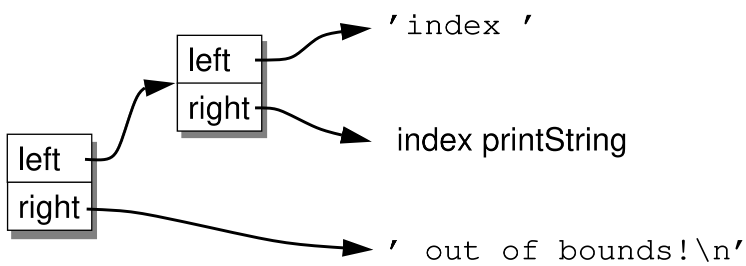


Even this recursion checking is conservative, however. The compiler might be able to inline a recursive call without getting into an infinite loop, if other information available to the compiler makes the processing of the two method invocations somehow different. To illustrate this possibility, consider the **print** method defined for **cons** cells that simply sends **print** to each of the receiver's subcomponents:

```
"Shared behavior for all cons cells"
traits cons = ( |
  ...
  print = ( left print. right print. ).
  ...
| ).
"Representation of an individual cons cell"
cons = ( |
  parent* = traits cons.
  left.
  right.
| ).
"Shared behavior for all collections"
collection = ( |
  ...
  "Concatenate two collections by creating a cons cell"
  , collection = ( (cons clone left: self) right: collection ).
  ...
| ).

"Test program"
outOfBoundsError: index = (
  ('index ', index printString, ' out of bounds!\n') print.
).
```

When compiling the **outOfBoundsError:** method, the compiler inlines the "**,**" concatenation messages down to the low-level **cons**ing code, retaining intimate knowledge about the contents of the **left** and **right** subcomponents of the **cons** cells.



The compiler inlines the initial **print** message sent to the top-most **cons** cell (since it knows the receiver's map) and inlines the nested **left** and **right** messages. At this point the compiler could inline the nested **print** messages, since it knows the types of the contents of the subcomponents of the **cons** cell it just constructed; in fact, the compiler has enough information to inline this whole example down to a series of three **_StringPrint** primitive calls, plus a call to **index printString**, thus optimizing away the **cons**'ing completely. Unfortunately, the recursion detection system prevents the compiler from inlining the second **print** message sent to the nested **cons** cell since the receiver maps are the same for the two **print** messages and one is invoked directly within the other, even though actually performing the inlining would not send the compiler into an infinite loop. With the current SELF compiler only the outermost level of **cons**'ing and the outermost level of **print**'ing is eliminated in this example.

Unfortunately, detecting when recursion would not lead to infinite looping is difficult. The SELF compiler remains conservative by never inlining a method with the same receiver map twice in the same mixed lexical and dynamic call chain. As an extension the compiler could inline a recursive method up to some small number of times. This change would speed both tightly recursive programs by "unrolling" the recursion a few times and would catch some cases like the **cons** cell example above where the recursion is in fact bounded. Of course, these benefits would need to be balanced against the extra compile time and space needed to unroll recursive calls.

7.1.3 Speeding Compile-Time Message Lookup

Compile-time message lookup turns out to be a bottleneck in the SELF compiler, consuming around 15% of the total compilation time. To speed compile-time message lookups, the compiler maintains a cache of lookup results, much like the run-time system includes a cache of message results to speed run-time message lookups. A compile-time message cache should be most important for certain classes of message sends, such as sends accessing global slots such as **`nil`**, **`true`**, and **`false`** which require a lot of searching of the inheritance hierarchy, and common messages such as **`+`** and **`ifTrue:`** which are sent frequently.

Unfortunately, because of the compiler's internal memory allocation scheme, the current compile-time lookup cache can only cache message lookup results during a single compilation; the cache must be flushed at the end of each compile.* This significantly reduces the hit rate of the cache, since the cache must be refilled with each new compilation. For example, one of our benchmark suites performed approximately 16,750 compile-time message lookups. Of these, 4,150 accessed slots other than local slots (12,600 were argument and local variable accesses, which did not go through the compile-time lookup cache). Of the 4,150 "real" messages, only 1,300 were found in the compile-time lookup cache; 2,850 caused misses, for a hit rate of only 30%. Consequently, the compile-time lookup cache reduces compile time by only 2% or 3%, as reported in section 14.3. A long-lived compile-time lookup cache would presumably have a much higher hit rate, since the cache-filling overhead associated with the per-compilation cache would be amortized over all compilations, and thus reduce the compile time costs of compile-time message lookup further. One possible implementation strategy for this cache that would interact well with change dependency links is outlined in section 13.2.

7.2 Inlining Primitives

In addition to user-defined methods, the compiler inlines the bodies of some primitive operations. Since primitive invocations are not strictly messages, but are more analogous to statically-bound procedure calls, the compiler can always inline calls of primitive operations. The implementations of certain commonly-used primitives, such as integer arithmetic and comparison primitives, the object equality primitive, and array accessing and sizing primitives, are built-in to the compiler. When any of these "known" primitives is called, the compiler generates code in-line to implement the primitive. Non-inlined primitives are implemented by a call to a function in the virtual machine that executes the primitive.

The compiler inlines commonly-used primitives to achieve good performance. The call and return overhead for small primitives is frequently larger than the cost of the primitive itself. Also, since primitives in SELF are robust, they always check the types and values of their arguments for legality (for instance, that the arguments to an integer addition primitive are integers or that the index to an array access primitive is within the bounds of the array). Many of these checks can be optimized away using the type information available at the primitive call site.

* The compiler allocates all internal data structures, including the compile-time lookup cache, in a special region of memory. When the compilation completes, the entire region is emptied. This approach relieves the compiler of the burden of manual garbage collection (the virtual machine, written in C++, has no support for automatic garbage collection of internal data structures) at the cost of not being able to easily allocate data structures that outlive a single compilation.

If the arguments to a side-effect-free, idempotent primitive are constants known at compile-time, the compiler can *constant-fold* the primitive, executing the primitive operation at compile-time instead of run-time and replacing the call to the primitive with its compile-time constant result. Constant folding is especially important for optimizing some user-defined control structures whose arguments may frequently be simple constants that control the behavior of the control structure. For example, the **`to:Do:`** control structure (SELF's form of a simple integer **`for`** loop) is defined in terms of the more general **`to:By:Do:`** control structure, with a step value of **`1`**:

```
to: end Do: block = ( to: end By: 1 Do: block ).
```

The body of the **`to:By:Do:`** routine tests the sign of the step value to see if the **`for`** loop is stepping up or down:

```
to: end By: step Do: block = (
  step compare: 0
        IfLess: [ to: end ByNegative: step Do: block ]
        Equal: [ error: 'step is zero in to:By:Do: loop' ]
        Greater: [ to: end ByPositive: step Do: block ] ).
to: end ByNegative: step Do: block = (
  "step down from self to end"
  | i |
  i: self.
  [ i >= end ] whileTrue: [
    block value: i.
    i: i + step. "step is negative, so i gets smaller"
  ] ).
to: end ByPositive: step Do: block = (
  "step up from self to end"
  | i |
  i: self.
  [ i <= end ] whileTrue: [
    block value: i.
    i: i + step.
  ] ).
```

The compiler can constant-fold the comparisons in the **`compare:IfLess:Equal:Greater:`** method, since it knows the receiver is **`1`** and the argument is **`0`**:

```
compare: arg IfLess: ltBlock Equal: eqBlock Greater: gtBlock = (
  < arg ifTrue: ltBlock
          False: [ = arg ifTrue: eqBlock False: gtBlock ] ).
```

Constant folding is critical to optimizing away the overhead of the general **`to:By:Do:`** loop to just the **`to:ByPositive:Do:`** loop.

7.3 Summary

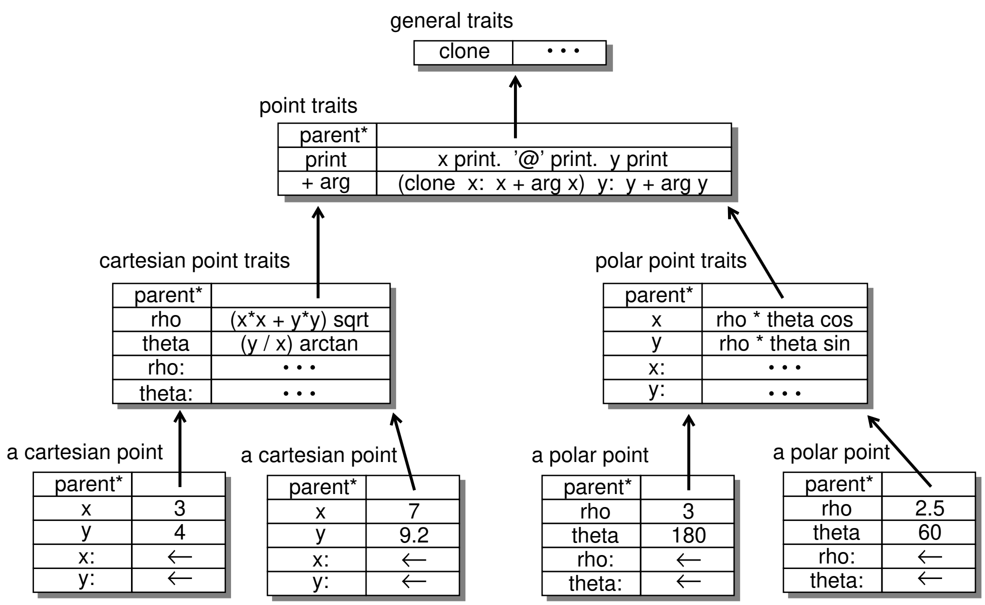
Inlining methods and primitives slashes the call frequency in pure object-oriented languages and languages based on user-defined control structures, opening the door for traditional optimizations such as global register allocation and common subexpression elimination that work well only for code with few procedure calls. However, inlining a message requires static knowledge of the map of the message receiver, and so requires sophisticated techniques for inferring the types of objects. These techniques are the subject of the next four chapters.

Chapter 8 Customization

Customization, one of the SELF compiler's main techniques, provides much of the type information enabling compile-time message lookup and inlining. This chapter describes customization and discusses important related issues.

8.1 Customization

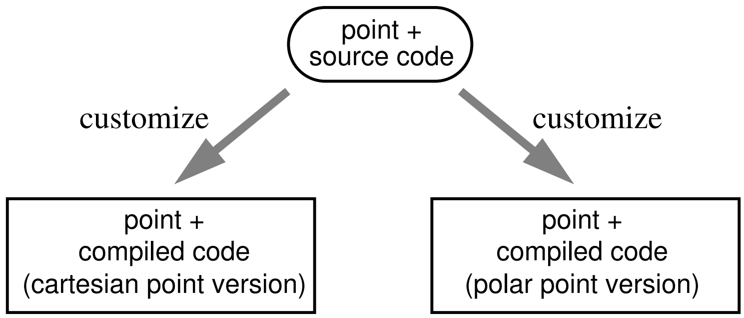
Programmers using object-oriented languages receive much of their expressive power by applying inheritance to organize code, factoring common fragments of code out into shared ancestors. In many cases, the factored code must be parameterized by specific information available to the inheriting objects or subclasses. The factored code can gain access to the specific information by sending a message to **`self`** and relying on the inheriting objects or subclasses to provide specific implementations of the message that take care of the more specific computation. For example, the point **`print`** example presented in section 4.6 uses sends to **`self`** to access behavior specific to either cartesian or polar points:



Sending the **`x`** and **`y`** messages to (implicit) **`self`** allows the single **`print`** method to work for both kinds of points, irrespective of how they actually implement the **`x`** and **`y`** messages.

Most object-oriented systems generate one compiled-code method for each source-code method. The single compiled method must be general enough to handle all possible receiver types that might inherit the single source method. In particular, a send to **`self`** must be implemented as full dynamically-bound messages, since different inheriting objects will provide different implementations of the message. This implementation architecture places well-factored object-oriented code at a performance disadvantage relative to less-well-factored code.

The SELF compiler avoids penalizing well-factored code by compiling a *separate version* of a source-code method for each receiver type (i.e., each receiver map) on which the method is invoked. Each version is invoked only for receivers with a particular map. Within the method, the compiler knows the precise type of **`self`** (the single receiver map for **`self`**) and therefore can perform compile-time message lookup and inlining for all sends to **`self`**. For example, in the point **`print`** example, the compiler generates one compiled version of **`print`** for cartesian point receivers and another compiled version for polar point receivers.



Within each version, the type of **`self`** is known statically, and the **`x`** and **`y`** messages can be statically bound to target methods and inlined.

Since in SELF many common messages are sent to **`self`**, including instance variable accesses, global variable accesses, and some control structures, this extra type information makes a huge difference in the performance of SELF programs; as shown in section 14.3, without customization SELF would run an average of 3 times slower. Customization completely overcomes the apparent performance disadvantage of accessing instance variables and global variables via messages as in SELF rather than special restrictive linguistic constructs as in Smalltalk and most other languages.

8.2 Customization and Dynamic Compilation

Customization potentially could lead to an explosion in compiled code space consumption. If a single source method were inherited by many different receiver types, it could be compiled and customized many different ways. Fortunately, this potential space explosion can be controlled in most cases by integrating customization with the *dynamic compilation* strategy used by the Deutsch-Schiffman Smalltalk-80 system, described in section 3.1.2.

As described in section 6.2, SELF source code is first parsed into byte code objects; no compilation takes place until run-time. When a method is first invoked, the compiler generates code for that method based on the byte-coded pre-parsed description of the source code. The compiler stores the resulting generated code in a cache (called the *compiled code cache*) and finally jumps to the generated code to execute the method.

The Deutsch-Schiffman Smalltalk-80 system generates a single compiled method for each executed source method. In the SELF compiler, dynamic compilation is integrated with customization: a method is custom-compiled only when first invoked with a particular receiver map. This approach usually limits the code explosion potentially created by customization, since instead of customizing for every inheriting receiver type, the system only customizes for those inheriting receiver types currently being manipulated as part of the user's "working set" of programs. Section 8.6 describes some pathological cases where dynamic customized compilation is still too wasteful of compiled code space, however, and suggests some approaches for handling these rare situations.

8.2.1 Impact of Dynamic Compilation

Dynamic compilation has a marked effect on the flavor of the system. Dynamic compilation is naturally incremental, enabling an effective programming environment. Turnaround time for programming changes can be short, since only code that needs to be executed must be compiled after a change, and only code affected by the change need be recompiled at all.

With dynamic compilation, compilation speed becomes much more important. Most programmers are unwilling to accept lengthy compilation pauses interleaved with the execution of their programs, even if the total compile time with the dynamic compilation system is less than with a traditional batch compilation system. Ideally, programmers should be unaware of compilation entirely, implying that each compilation or series of compilations should take only a second or two in a long-running non-interactive program or small fraction of a second in an interactive program or a program with real-time needs such as an animation play-back program. Traditional batch compilers, especially optimizers, normally do not labor under such compilation speed restrictions, probably because users do not expect compilation to be fast or unnoticeable. In a sense, dynamic compilation has created this problem by raising the level of expectation of users. The SELF compiler thus takes special pains to reduce compilation time, such as *lazy compilation of uncommon branches* as described later in section 10.5.

8.2.2 Compiled Code Cache

Dynamic compilation systems require that both the compiler and all the source code for the system be available at run-time (although possibly in a compact form, like the SELF byte code objects). These needs seem to imply that a dynamically compiled system will take up more space at run-time than a corresponding statically-compiled system. However, in both our SELF system and in the Deutsch-Schiffman Smalltalk-80 system, the dynamically-compiled code is *cached* in a fixed-sized region. If the code cache overflows, some methods are flushed from the cache to make room for new compiled code; the flushed methods will be recompiled when next needed. Caching has the advantage that only the working set of compiled code needs to exist in compiled form; all other methods exist only in the more compact byte code form. This organization can actually save space over similar statically-compiled systems, since in a statically-compiled system all compiled code must exist all the time, while in a dynamically-compiled system only the more compact byte codes need be kept all the time.

On the other hand, a dynamically-compiled system that caches the results of compilation may incur more compilation overhead than a dynamically-compiled system without caching (i.e., with a “cache” that only ever grows larger, never flushing code unnecessarily) or a statically-compiled system (assuming that all compiled code will eventually be needed). The size of the compiled code cache (and whether it is unbounded) is thus an important parameter to controlling the behavior of a system based on dynamic compilation: too small a compiled code cache can lead to excessive compilation overhead akin to thrashing, while too large a compiled code cache can lead to excessive paging on systems with virtual memory. In the current SELF system, the compiled code cache is sized at about 4MB for machine instructions (with additional space reserved for other information output by the compiler along with instructions), which is enough to hold all the commonly-used compiled code for the prototype SELF user interface, currently the largest SELF application.

8.2.3 LRU Cache Flushing Support

The current implementation of dynamic compilation and caching requires some support from the compiler to implement the replacement algorithm used in the compiled code cache. To select the compiled method(s) to flush to make room for new compiled methods, the code cache uses a *least-recently-used* (*LRU*) approximation strategy. Each compiled method is allocated a word of memory, used to record whether the method has been used recently. At the beginning of each compiled method, the compiled code must zero its word to mark the method as recently used. Partial sweeps over the compiled code cache check which methods have been used recently (i.e., which have their words zeroed), transferring this information to a separate, more compact data structure. After scanning, the examined words are reset to non-zero to begin the next time interval. This clock-like LRU detection strategy imposes a small run-time overhead to clear a word of memory at a fixed address on every method invocation.

8.3 Customization and Static Compilation

SELF can use customization without incurring a huge blow-up in compiled code space because the SELF compiler relies on dynamic compilation to limit customization to those receiver type/source method combinations that actually occur in practice. However, most object-oriented language implementations use traditional static compilation. In such environments, customization would appear to become much less practical, since compiled versions of source methods would have to be compiled “up front” for all possible receiver type/source method combinations, irrespective of whether the combinations occur in practice.

Nevertheless, the Trellis/Owl system (described in section 3.2.3) automatically compiles customized versions of methods for all inheriting subclasses statically. Trellis/Owl, like SELF, accesses instance variables via messages, and consequently Trellis/Owl's implementors developed a similar optimization to overcome the potential performance problems. Their system includes several techniques that together apparently keep down the costs of static customization. The Trellis/Owl compiler conserves space by generating a new compiled version of a method only when it differs from the compiled code of its superclass' version. This technique would solve the **`equalsString:`** problem by having all non-string classes share the same compiled code for the default definition of **`equalsString:`**. Trellis/Owl also keeps compiled code space costs and compile time costs down by performing little optimization and no inlining of methods or primitives; only messages to **`self`** accessing instance variables are inlined. Global variables and constants are accessed directly, not by messages, so objects such as **`false`** can be accessed directly without sending messages, unlike in SELF where **`false`** is accessed via a normal implicit-self message. Finally, Trellis/Owl includes a suite of built-in control structures and special declarations to make it easier to compile code for common types such as integers and booleans. We doubt that static customization would remain practical in an aggressively optimizing system like SELF with a pure language model, although further research to verify this belief would be useful.

8.4 Customization as Partial Evaluation

Customization can be viewed as a kind of *partial evaluation* (introduced in section 3.4.3): by customizing the compiler partially-evaluates a source method with respect to the type of its receiver to produce a residual function (the customized compiled code). Also like partial evaluation systems, the SELF compiler makes heavy use of type analysis and inlining to optimize routines.

There are several important distinctions between the SELF compiler and partial evaluators. The SELF compiler partially-evaluates (i.e., customizes) methods using type information extracted at run-time using dynamic compilation without any user type or data declarations, while partial evaluation systems typically are given an extensive static description of the input to the program over which the program is to be partially evaluated. Partial evaluators primarily propagate constant information, while the SELF compiler typically propagates more general information such as the representation-level types of expressions. Finally, partial evaluators typically unroll loops or inline recursive calls as long as they can be "constant folded" away, and therefore do not terminate on non-terminating input programs, such as programs containing errors that lead to infinite recursions. The SELF compiler must be more robust, compiling code in a reasonable amount of time even for programs that contain errors. Accordingly, the SELF compiler does not unroll loops arbitrarily (sacrificing some opportunities for optimization in the process), and its recursion detection rules (described in section 7.1.2.2) are more elaborate than those found in most partial evaluators.

8.5 In-Line Caching

8.5.1 In-Line Caching and Customization

With customized methods, the message lookup system has the additional job of locating the particular customized version of a source method that applies to the receiver type. Fortunately, this selection can be folded into *in-line caching* for no additional run-time cost. Like the Deutsch-Schiffman Smalltalk-80 system (described in section 3.1.2), the SELF system uses in-line caching to speed non-inlined message sends. In traditional in-line caching, the compiler verifies the cached method by checking whether the receiver's map is the same as it was when the method was cached in-line. This check extends naturally to handle customized methods by instead verifying that the receiver's map is the one for which the method was customized. This test has the same hit rate as for traditional in-line caching, since if the receiver's map is the same as before then its map will be the one for which the method was customized, and vice versa. The modified test also takes up less compiled-code space, since the cached receiver map no longer needs to be stored in-line at the call site. Finally, the new check is faster to perform, since the cached last receiver map no longer needs to be fetched from its in-line memory location (the required map value is now a compile-time constant embedded in the instructions of the method's prologue). The following SPARC instructions implement this check at the beginning of methods invoked through dynamically-dispatched message sends:

- If customized for integer receivers:

```
  andcc %receiver, #3 ; test low-order two bits for 00 integer tag
  bz,a _hit
  ; instruction beginning rest of method prologue (in delay slot)
_miss:
  sethi %hi(_InlineCacheMiss), %t ; call in-line cache miss handler
  jmp [%t + %lo(_InlineCacheMiss)]
_hit:
  ; rest of method prologue
```

- If customized for floating point receivers:

```
  andcc %receiver, #2 ; test second low-order bit for 10 float tag (cannot be mark (11))
  bnz,a _hit
  ; instruction beginning rest of method prologue (in delay slot)
_miss:
  sethi %hi(_InlineCacheMiss), %t ; call in-line cache miss handler
  jmp [%t + %lo(_InlineCacheMiss)]
_hit:
  ; rest of method prologue
```

- If customized for other receivers:

```
  andcc %receiver, #1 ; test low-order bit for 01 memory tag (cannot be a mark (11))
  bnz,a _map_test
  ld [%receiver + 3], %map ; load receiver's map (in delay slot)
_miss:
  sethi %hi(_InlineCacheMiss), %t ; call in-line cache miss handler
  jmp [%t + %lo(_InlineCacheMiss)]
_map_test:
  sethi %hi(<customized map constant>), %t ; load 32-bit map constant
  add %t, %lo(<customized map constant>), %t
  cmp %map, %t ; compare receiver's map to customized map constant
  beq,a _hit
  ; instruction beginning rest of method prologue (in delay slot)
  ba,a _miss ; branch back to call of in-line cache miss handler
_hit:
  ; rest of method prologue
```

For messages that miss in the modified in-line cache, the compiled code table is consulted to find the appropriate customized version of the target method. To support customization, the map of the receiver object is included in the key that indexes into the table. If a version of the method with the right receiver map has not yet been compiled, then the compiler is invoked to produce a new customized version, and the resulting version is added to the compiled code table for future uses of the same source method with the same receiver type. The in-line cache is then *backpatched* (i.e., overwritten) to call the newly-invoked method, so that future executions of the same message send will test the most recently invoked method first.[*]

[*] After the bulk of the research reported in this dissertation was completed, Urs Hölzle and other members of the SELF group designed and implemented an extension to normal in-line caching called *polymorphic inline caching* [HCU91]. Polymorphic inline caches roughly act like dynamically-growing chains of normal "monomorphic" in-line caches, eventually increasing the hit rate for a polymorphic inline cache to 100%. The performance data presented in Chapter 14 includes the improvements from polymorphic inline caches.

8.5.2 In-Line Caching and Dynamic Inheritance

In the presence of dynamic inheritance, the outcome of method lookup depends on more than just the map of the receiver: it also depends on the run-time contents of any assignable parent slots traversed by the lookup. Consequently, the simple in-line cache receiver map check is insufficient to guarantee that the cached method is correct for the receiver. One approach, used in an early SELF implementation, would simply disable in-line caching for messages affected by dynamic inheritance; a full lookup would be performed for every message involving assignable parents. Unfortunately, this approach places a severe run-time overhead on the use of dynamic inheritance.

The current SELF system extends in-line caching to also check the state of any assignable parents as part of checking for an in-line cache hit. The compiler generates extra code in the method prologue after the receiver type check that verifies the contents of any assignable parent slots. In most cases, the compiler only has to check the map of the assignable parent against a statically-known constant; in some cases the compiler must check the parent object's identity (these cases relate to certain aspects of SELF inheritance rules that depend on the relative identities of objects involved in the message lookup). If all assignable parents are correct for the cached method, then the in-line cache hits and the body of the method is executed. Otherwise, the in-line cache misses and additional processing is needed to resolve the miss, potentially involving a full message lookup.

This implementation of dynamic inheritance is much better than the simple approach of disabling in-line caching altogether, but it is still not as fast as desired, since the presence of dynamic inheritance currently blocks compile-time message lookup and message inlining. To make dynamic inheritance truly competitive in performance with messages involving only static inheritance, the system would need to include some means of statically-binding and inlining messages influenced by dynamic inheritance.

8.5.3 In-Line Caching and _Performs

In SELF, users may send a message whose name is a computed run-time value rather than a static compile-time string using a **_Perform** primitive.* For example, the following SELF code could be used to implement the **for** loop control structure more succinctly than the current way presented in section 7.2:

```
to: end By: step Do: block = (
  step compare: 0
        IfLess: [ to: end By: step Sending: '>=' Do: block ]
        Equal: [ error: 'step is zero in to:By:Do: loop' ]
        Greater: [ to: end By: step Sending: '<=' Do: block ] ).
to: end By: step Sending: name Do: block = (
  "step either up or down from self to end"
  | i |
  i: self.
  [ i _Perform: name With: end ] whileTrue: [
    block value: i.
    i: i + step.
  ] ).
```

This version of the **for** loop control structure passes in the name of the message to be used to test whether the loop is done.

To implement **_Perform**'ed messages efficiently, we generalize the notion of a message, and generalize the in-line cache prologue to handle this more general kind of message. A general message involves a number of parameters that control the message lookup, including the receiver's map and the name of the message. Each of these parameters may be either a compile-time constant or a run-time computed quantity. A normal message send is simply a special case of this generalized message, with the message name a compile-time constant. For **_Perform**'s the message name may be a run-time computed value. In addition, the receiver map, normally a run-time computed value, might be a compile-time constant, for instance if the message has been statically-bound but not inlined (such as for a recursive call). The generalized in-line cache is responsible for checking any run-time computed parameters to the message lookup that are not guaranteed to be compile-time constants (and so already checked at compile-time), such as the receiver's map or the message name.

* Other variants of **_Perform** allow other aspects of the message lookup, such as the object with which to start the search, to be computed and passed in as run-time values.

The compiler attempts to determine statically as many of the parameters to a message as possible, since the compiler can generate better code if it knows more about the message. For example, if the compiler can infer the value of the message name argument of the **_Perform** primitive statically, it replaces the **_Perform** primitive with a normal message send, which the compiler then attempts to optimize further. In this way the compiler integrates the treatment of normal messages and **_Perform**'ed messages, using the same kinds of techniques and run-time mechanisms to improve the performance of both.

8.6 Future Work

A logical extension to our current system would be to customize on the types of arguments in addition to the type of the receiver. There is no theoretical reason why customization should not apply to arguments in addition to the receiver, and the performance of some programs likely would improve with argument customization. From a practical standpoint, however, in a singly-dispatched language such as SELF the receiver is more important than the other arguments, since message lookup depends on the type of the receiver but not on the types of the arguments. Customizing on the receiver comes at no additional run-time cost, since in-line caching can handle customized methods at least as easily as non-customized methods. In contrast, any argument customization would require additional run-time checks in the method prologue. Whether argument customization pays off in practice depends on whether the benefits of knowing the types of arguments outweigh the run-time costs associated with checking the types of the customized arguments in method prologues and the costs in additional compiled code space. It seems likely that a successful system would only customize on those arguments, if any, that received heavy use in the body of a method, since customization on all arguments for all methods would almost surely lead to significant compiled-code space and compilation time overheads.

Even though customization usually improves performance significantly without greatly increasing compiled code space usage, customization may not be appropriate for all source methods. For some methods, the extra space cost associated with customization may not be worth the improvement in run-time performance, either because the method does not send messages to **self** often enough or because the method is called with many different receiver types. For example, in the current SELF system, testing two arbitrary objects for equality is implemented using *double dispatching* [Ing86]. The implementation of **=** for strings, for example, is the following:

```
traits string = ( |
  ...
  = anObject = ( anObject equalsString: self ).
  equalsString: s = (
    "both arguments are strings; now compare characters"
    ... ).
  ...
| ).*
```

If both arguments to **=** are strings, then the version of **equalsString:** for strings is called, which proceeds to compare individual characters within the strings. If, however, the argument to **=** is not a string, then the default version of **equalsString:** is called instead:

```
traits defaults = ( |
  ...
  equalsString: s = ( false ).
  ...
| ).
```

This version just returns **false**, since a string is never equal to something that is not also a string. The compiler generates a separate customized version of this method for all non-string receiver types compared against strings in practice. Normally this would be a small set, but one SELF program iterated through all the objects in the heap comparing them to a particular string; this program caused the compiler to generate a customized version of the default **equalsString:** message for *every non-string type in the system.* Clearly a single shared version of this method would be better. To prevent such pathological cases, we are investigating approaches in which the compiler can elect not to customize methods where the costs of customization outweigh the benefits.

* This syntax is not precisely SELF syntax; we using a more intuitive syntax in this dissertation for pedagogical reasons.

8.7 Summary

The SELF compiler performs customization as one of its main techniques to improve performance. Customization provides the compiler with precise static knowledge of the type of **`self`**, enabling it to statically-bind and inline messages sent to **`self`**. These kinds of messages are extremely common in SELF, since instance variables and global variables are accessed by sending messages to **`self`** rather than by special-purpose language mechanisms with limited expressive power. Customization directly overcomes the performance disadvantages of SELF's more expressive approach, clearing the way for future languages to rely on messages for variable accesses without adverse performance impact. By coupling customization with dynamic compilation, the space overhead for customization can be kept reasonable. By coupling customization with in-line caching, no extra run-time work is required to select the right customized version of an invoked source method.

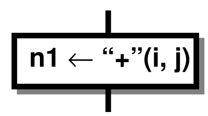
Chapter 9 Type Analysis

Type information plays a critical role in improving performance of object-oriented languages such as SELF. To obtain the maximum benefit from any type information the compiler can infer, the compiler uses sophisticated flow analysis to propagate type information through the control flow graph. This propagation is called *type analysis*, and is the subject of this chapter. To simplify the exposition, only type analysis for straight-line code without loops is discussed here; type analysis of loops will be described later in Chapter 11.

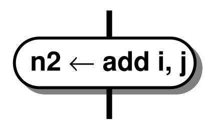
9.1 Internal Representation of Programs and Type Information

9.1.1 Control Flow Graph

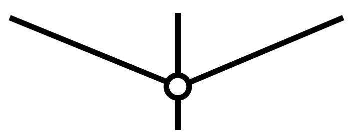
The compiler represents the method being compiled using a control flow graph data structure, with different kinds of nodes in the control flow graph for different kinds of operations. These nodes include high-level nodes such as message send nodes,



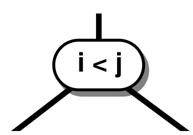
instruction-level nodes such as add instructions,



control flow nodes such as merge nodes



and conditional branch nodes,[*]



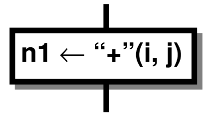
and bookkeeping nodes such as assignment nodes.

Most passes in the compiler involve some sort of traversal of this graph.

[*] In all diagrams in this dissertation, the "yes" or "success" arc of a branch node will exit on the left.

9.1.2 Names

The compiler uses *names* to capture data dependencies among the nodes. A name corresponds to either a source-level variable name (such as an argument or local variable) or a compiler-generated temporary name (such as a name referring to the result of a subexpression in the source code). The compiler treats both kinds of names in the same way. Names represent the flow of data through nodes in the graph by having some nodes bind names to computed results, with other nodes referring to bound names as arguments. For instance, the message send node

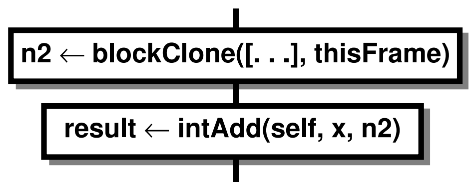


passes the data values bound to the names **`i`** and **`j`** as the receiver and argument to the **+** message, and binds the result data value to the (temporary) name **`n1`**.

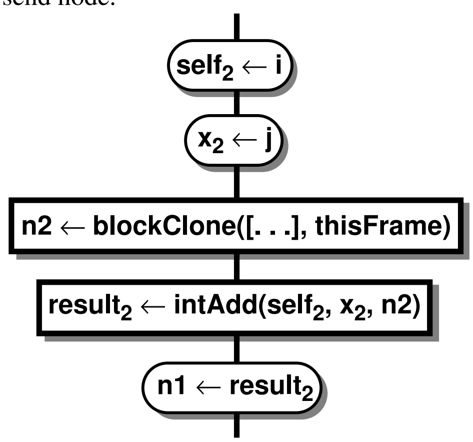
To illustrate control flow graphs, names, and inlining, consider the above message send node. If the compiler can infer that the type of the receiver **`i`** of the **+** message is, say, an integer, then it can lookup the **+** message for integers at compile-time and locate the following method:

```
+ x = ( _IntAdd: x IfFail: [ . . . ] ).
```

The compiler then can inline this method. Inlining a method entails constructing a new control flow graph for the inlined method

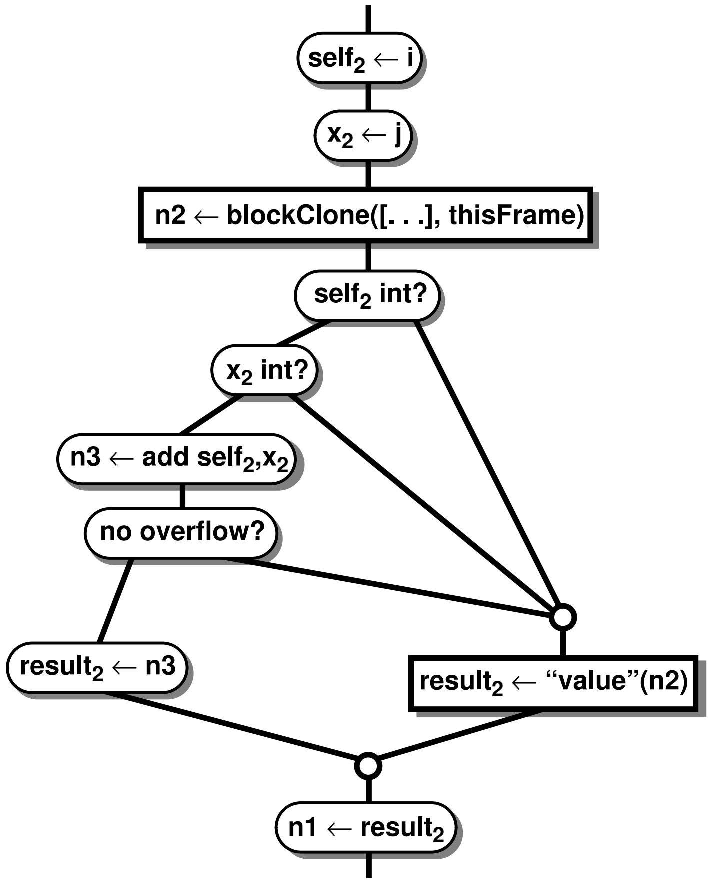


and splicing this control flow graph into the main graph in place of the message send node. Name assignment nodes are inserted to assign the names of the actual receiver and arguments to the names used as the formals in the inlined control flow graph, and likewise to assign the name of the result used in the inlined control flow graph to the name of the result of the eliminated message send node.*



* Names like **$\mathtt{self}_2$** and **$\mathtt{x}_2$** correspond to formals of inlined methods. New names are created for each inlined copy of a method.

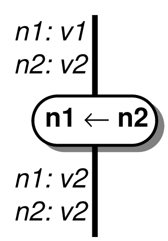
The compiler can also inline-expand the call to the **`intAdd`** primitive.



Most of the transformations of the control flow graph performed by the SELF compiler are of this flavor.

9.1.3 Values

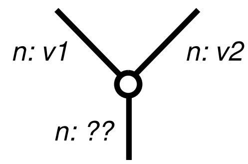
In our view, names are merely mechanisms for programmers and the compiler to refer to underlying data values and have no run-time existence. These underlying data values referred to by names are represented explicitly in the SELF compiler as *values*. Each value data structure in the compiler represents a particular run-time object. Many names may refer to the same value at a particular point in the program, and a single name may refer to different values at different points in the program. For example, after the assignment node



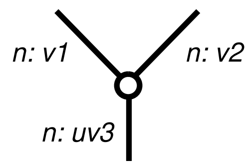
both **`n1`** and **`n2`** refer to the same value object (the value that **`n2`** referred to before the assignment node); **`n1`** may refer to a different value after the assignment than before the assignment. Since assignment nodes simply affect the compiler's internal mappings from names to values, they do not directly generate any machine code.

The primary invariant relating names and values is that if two names map to the same value at some point in the program, then both names are guaranteed to refer to the same object at run-time at that point. On the other hand, if two names map to different values, then the compiler cannot tell whether the names will refer to the same object at run-time or not. Values are immutable; a new value is created whenever the compiler needs a representation for a run-time object that is potentially different from any other object. For example, the receiver and argument names of the method being compiled are initialized to new unique values, as are the results of non-inlined message sends, primitives, and integer arithmetic. References to the contents of assignable data slots in the heap (such as instance variable accesses) also are bound to new unique values.

Merges in the control flow graph pose an interesting problem for maintaining the invariant relating names and values. If a name is bound to a value **`v1`** along one predecessor branch of a merge node and bound to a different value **`v2`** along another predecessor branch, to what value should the name be bound after the merge?

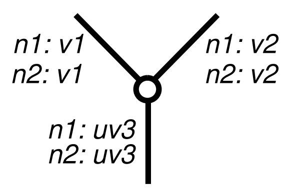


If either **`v1`** or **`v2`** is chosen, then for some paths through the program at run-time the compiler will have inferred incorrect information, potentially leading it to make incorrect optimizations that cause the optimized program to misbehave or even crash. Since neither incoming value is acceptable, a brand new value must be created to represent the run-time data value referred to by the name after the merge. The new values created by merge nodes are called *union values*.

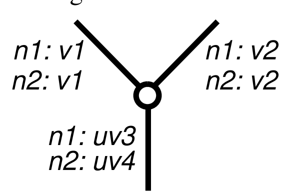


Union values are very similar to the ϕ-functions of SSA form, described in section 3.4.4.

Currently, each name that needs a new union value at a merge node is given a unique union value. A more accurate analysis would locate those names that for each predecessor branch are bound to the same value and assign them all the same new union value after the merge. For example, after the following merge node both **`n1`** and **`n2`** are guaranteed to refer to the same run-time object, and so should be given the same union value.

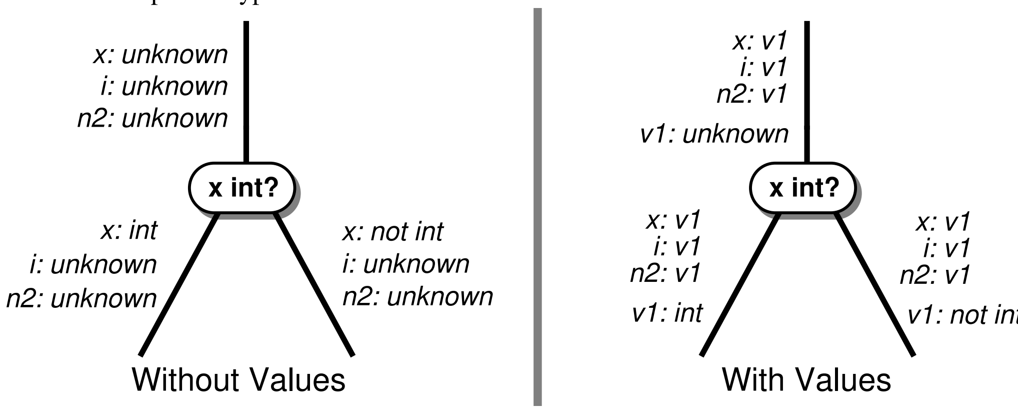


The SELF compiler's analysis currently is not sophisticated enough to recognize this situation, and so **`n1`** and **`n2`** each will be given its own new union value after the merge.



Values provide a better base than names for certain kinds of analysis and optimizations. As a consequence of aggressive inlining of user-defined control structures and operations, new names are created at a high rate, with many trivial assignment nodes introduced merely to assign one name to another (e.g., the name of an actual to the name of a formal). Values provide an explicit representation of the objects flowing through these names and as such are closer to the variables of a traditional language and compiler. Techniques and optimizations that are normally based on variable names in a traditional compiler, such as register allocation and common subexpression elimination, are more naturally based on values in the SELF compiler. (Common subexpression elimination with values will be described in section 9.6; global register allocation will be described in section 12.1.) In these respects, values serve purposes similar to those served by subscripted variables in SSA form (described in section 3.4.4). However, we feel that explicitly separating values from names, propagating values through the control flow graph independently from names and testing the values for equality, is simpler than first transforming the program into SSA form, with each name replaced with several subscripted names, then merging subscripted names into equivalence classes, and then testing the subscripted names for membership in the same equivalence class.

Values also improve the effectiveness of type analysis. Run-time type tests of a particular name, such as those implementing run-time type checking of the arguments of primitives or verifying guesses as part of type prediction (described in section 9.4), alter the type associated with the *value* that is bound to the tested name, instead of just the name itself as would happen in a simpler system that mapped names directly to types. This allows a single type test to alter the inferred type of several names, i.e., all those that are currently bound (aliased) to the same tested value. For example, without values, the type of only the tested value would be updated, while with values all aliased names will be updated with the improved type information.



Since many of these type tests occur deeply nested in inlined control structures and operations, the names that are used as part of the test are just local temporary names. Values therefore become critical for communicating this important type information beyond the local scope of the inlined control structure or operation. As shown in section 14.3, without values, SELF would run an average of 50% slower.

9.1.4 Types

Types are the primary data structures used by the compiler to represent type information and support various kinds of type-related optimizations such as compile-time message lookup and constant folding. A type describes a set of run-time objects usually sharing some common property. The particular kinds of sets the compiler is capable of describing concisely through types are motivated primarily by the optimizations currently performed by the compiler; new sorts of optimizations might require new kinds of type information to be represented and propagated through the control flow graph. The types currently used in the SELF compiler are described in the next few subsections.

9.1.4.1 Map Types

A *map type* specifies all objects that share a particular map, i.e., all objects in a single clone family.* This kind of type is perhaps the most important kind of type, since it is the most general type that still enables the compiler to perform message lookup at compile time and to perform type-checking of primitive arguments at compile time.

Map types are introduced by several sources:

- The type of **`self`** is a map type as a result of customization (described in Chapter 8).
- A run-time type test (such as testing whether an object is an integer) marks the tested object as being of a particular map type along the branch in which the test is successful.
- Some primitive operations are known to return objects of particular map types. For example, integer addition primitives are known to return integers if the primitive succeeds, and the **`_Clone`** primitive always returns an object with the same map as its receiver.

* For a class-based language this kind of type would specify all objects that are instances of a particular class and would be called a *class type*.

9.1.4.2 Constant Types

A *constant type* specifies a single object, i.e., a compile-time constant value. Constant types support the same sorts of optimizations as do map types, plus additional optimizations such as constant-folding of primitives.

Constant types are introduced by several sources:

- A literal in SELF source code is of constant type.
- The result of an inlined message that accesses a constant data slot (such as the **`true`** message) is of constant type.
- A run-time value test (such as testing for the **`true`** object at run-time) marks the tested object as being of a particular constant type along the success branch.

9.1.4.3 Integer Subrange Types

An *integer subrange type* specifies a range of integer values from a lower bound to an upper bound. Integer subrange types allow the compiler to perform some kinds of range analysis optimizations. For example, the compiler can eliminate an array bounds check when the range of the index is guaranteed to be within the bounds of the array. Similarly, the compiler can eliminate the overflow check from an integer arithmetic operation when the ranges of the two arguments prove that the result will not overflow the normal 30 bit integer representation. The compiler can even "constant-fold" an integer comparison when the ranges of the two arguments do not overlap. The integer map type and the integer constant types may be viewed as extreme cases of integer subrange types, although they are represented more concisely than other integer subrange types.

Integer subrange types are introduced by several sources:

- The result of a successful integer arithmetic primitive is an integer subrange (or an integer constant or the integer map type) computed from the ranges of the arguments to the operation.
- An integer comparison operation narrows the types of its arguments depending on the outcome of the comparison. For example, when comparing **`i < j`**, along the true branch the compiler can lower the upper bound of the type of **`i`** to be one less than the upper bound of the type of **`j`** (and similarly raise the lower bound of the type of **`j`**); along the false branch the analogous narrowing can occur. This narrowing can convert an integer map type into an integer subrange type.
- Some primitive operations are known to return integers within a particular range. For example, the array size primitive returns only non-negative integers less than some upper limit bounded by the maximum size of the heap.

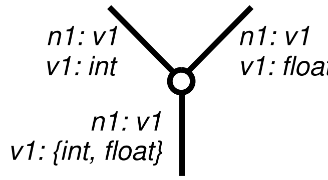
9.1.4.4 The Unknown Type

The *unknown type* specifies all possible objects and so conveys no information to the compiler. The compiler associates the unknown type with incoming arguments to the method being compiled (no customization is performed for arguments), the results of non-inlined messages, the contents of assignable data slots in the heap (e.g., instance variables), and the results of some primitive operations.

9.1.4.5 Union Types

A *union type* specifies the union of the objects specified by its component types. If a name is known to be of a particular union type, the contents of the variable at run-time could be any of the objects possible for any of the component types. If all the component types are covered by the same map type (i.e., all the component types specify objects in a single clone family), then the union type allows the same sorts of optimizations as a map type. It is more common, however, for the component types to be of different clone families, and so a union type typically provides less information than a map type but more information than the generic unknown type. Union types guide both run-time type casing (described in section 9.3) and splitting (described in Chapter 10).

Union types are created primarily as a result of merges in the control flow graph in which one value is associated with different types on different predecessor branches, analogously to union values. For example, if some value is associated with the integer map type along one predecessor branch of a merge, and associated with the floating point map type along the other predecessor branch, after the merge the value will be associated with the union type whose components are the integer map type and the floating point map type.



Union types are also created as the result of certain primitive operations. For example, the type of the result of a comparison primitive (along the success branch) is the union of the **`true`** constant type and the **`false`** constant type: *{true, false}*.

9.1.4.6 Exclude Types

An *exclude type* specifies a set of component types that the associated value is known *not* to be. Exclude types are introduced as a result of unsuccessful run-time type tests, such as recording that an object cannot be an integer along the failure branch of an integer type test. The compiler uses exclude types to avoid repeated tests for possibilities that are guaranteed not to occur.

Exclude types are not as expressive as would be full-fledged *difference types*. A difference type would specify those objects that were in one type but not in another, i.e., the set difference of the objects specified by two types. An exclude type is equivalent to the difference of the unknown type and the union of the excluded types:

<not t1, t2, t3> ⇔ *<unknown type> - <{t1, t2, t3}>*

Only differences from the unknown type can be expressed with an exclude type; an exclude type cannot express differences from some more precise type. For example, the current type system cannot record that some possibilities out of a map type are excluded, such as after a failed run-time value test of an object whose map type was known. A full-fledged difference type could describe this type as the difference between the known map type and the union of the excluded possibilities within the map type.

Unfortunately, generalizing exclude types to difference types would create new problems. With general difference types, a single type could be represented in multiple ways, significantly complicating type equality testing and other sorts of comparisons on types, such as whether one type covers another. This problem already exists in the current type system to some extent for integers, since an integer subrange type can represent the same type as a union of adjacent or overlapping integer constant types and/or integer subrange types. For example, the following types should all be considered equivalent:

{0, 1, 2, 3, 4, 5}	a union of integer constant types
{[0..2], 3, [4..5]}	a union of integer subrange and integer constant types
[0..5]	an integer subrange type

Adding general difference types exacerbates this problem, since now the following difference type also is equivalent to the above types:

```
int - {[minInt..-1], [6..maxInt]}
```

Even now the current SELF compiler does not always detect that types such as the first three types above are equivalent.[*] We chose not to worsen this situation, and consequently do not include fully general difference types in our type algebra. Luckily, exclude types seem to be sufficient in practice, since only rarely does the compiler perform value tests on objects whose map was already known. Nevertheless, the SELF compiler eventually should include some generalized approach to equality and other type comparisons in the presence of union types, difference types, and integer subrange types, perhaps by defining some canonical representation of sets of integers and translating into the canonical form prior to comparing types.

[*] This inaccuracy does not lead to incorrect compilation, since erring on the side of treating two things as of different types is safe, but it can lead to poorer generated code.

9.2 Type Analysis

To perform optimizations like compile-time message lookup and elimination of run-time type checks, the compiler needs to be able to determine the type associated with a name at a particular point in the program. To support this determination, the compiler maintains a mapping from names to values and a mapping from values to types. These mappings are propagated through the control flow graph as type analysis proceeds.

When performing type analysis, the compiler visits each node in the control flow graph in topological order. Each control flow graph node implements its own type analysis routine.* This routine typically examines the type mappings propagated from the node's predecessor(s), performs any optimizations of the node based on the type information, and finally produces new type mappings for the node's successor(s). The following subsections describe some node-specific type analysis operations in more detail. Discussion of type analysis in the presence of loops will be deferred until Chapter 11.

9.2.1 Assignment Nodes

An assignment node alters the name/value mapping of the assigned name. In the name/value mapping after the assignment node, the value associated with the assigned name (the "left-hand side") is the same as the value associated with the assigned-from name (the "right-hand side"). All other bindings are unaffected.

9.2.2 Merge Nodes

A merge node combines the name/value and value/type mappings from its predecessors to form a new pair of mappings after the merge. New union values and union types may be constructed.

9.2.3 Branch Nodes

A branch node makes two copies of the mappings, one for each successor branch, so that subsequent alterations to a mapping along one successor branch do not affect the mapping along the other successor branch.

9.2.4 Message Send Nodes

A message send node looks up the type bound to the receiver name. If this type is a map type (or more specific than a map type), then the compiler performs compile-time message lookup. If the lookup is successful and the compiler elects to inline the target method, then the message send node is replaced with a control flow graph representing the inlined target method; type analysis then begins to analyze the new inlined method. If the message is not inlined, then a new value is created to represent the result of the message send. The name of the message result is bound to this new value and the new value is bound to the unknown type. These altered mappings are then passed on to the message send's successor.

9.2.5 Primitive Operation Nodes

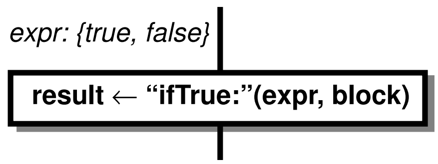
A primitive operation node first checks to see whether it can be inlined (whether the compiler has the implementation of the primitive built-in). If so, then the primitive operation node is replaced with nodes that implement the primitive in-line. The type information associated with the arguments to the primitive can be used to optimize inlined primitives, such as by eliminating unnecessary argument type checks, overflow checks, or array bounds checks. If the primitive is inlined, then type analysis continues with the first node of the primitive. Otherwise, a new value is created for the primitive's result and the primitive's result name is bound to this new value. The result value is in turn bound to either the unknown type or to some more precise type depending on the primitive and its argument types. The altered mappings are then passed on to the primitive's success to continue type analysis.

* The compiler is implemented in C++. Each control flow graph node is an instance of a class, with different classes for different kinds of control flow graph nodes. All control flow graph nodes define a virtual function which implements the node-specific type analysis.

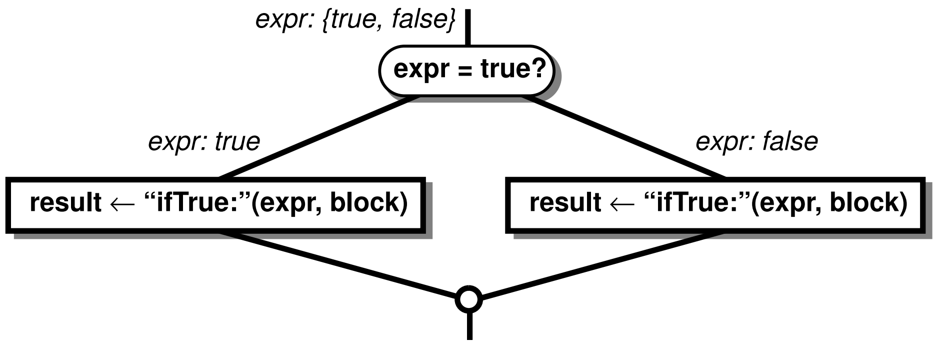
9.3 Type Casing

When the compiler can infer that the type of the receiver of a message is covered by a map type, the compiler can perform message lookup at compile-time, statically-binding and inlining the message. However, the type of a receiver frequently will not be a simple map type but instead will be a union of several different types. This can occur as a result of a primitive known to the compiler to return one of several different types (such as comparison primitives returning either **`true`** or **`false`**) or after a merge in the control flow graph where a single name was associated with several different types before the merge.

Type casing is a simple technique for optimizing message sends where the receiver is bound to a union type. For example, the **`ifTrue:`** message frequently is sent to an expression whose type is the union of the **`true`** constant type and the **`false`** constant type:



Optimizing such messages with type casing involves testing for each element type in the union and then branching to code specific to that type. Specifically, for each component of the union type that is covered by a map type, the compiler inserts a run-time type test before the message that checks for that type and in the successful case branches to a copy of the message. In the **`ifTrue:`** example, the compiler would perform type casing by inserting a run-time check for **`true`** that branched to a copy of the **`ifTrue:`** message:



Each copy of the type-cased message now can be statically-bound and inlined away, since the type of the receiver is known after a successful run-time type test. Furthermore, if all the components of the union type are covered by map types (such as in the **`ifTrue:`** example), then the last component of the union type does not need a run-time type check, since the other failed type checks will have excluded all other possible types. If on the other hand some components of the union type are not covered by a single map type (such as if the union type contains the unknown type as one of its components), or if the message cannot be inlined for some map types in the union, then a final dynamically-dispatched version of the message will be needed to handle the remaining case(s). Control flow re-merges after the copies of the message.

Type casing is a simple technique to take advantage of union type information. It has the effect of transforming a single polymorphic message into several monomorphic messages that can be further optimized. As such, it is an example of the general theme in the SELF compiler of trading away space for improved run-time performance. Type casing is also very similar to case analysis as performed in the TS compiler for Typed Smalltalk (see section 3.1.3). However, type casing incurs run-time overhead with the extra type tests for various component types of the union. While the resulting code is typically faster than the original message send (sometimes much faster, such as in the **`ifTrue:`** example above), it is not as fast as is frequently possible. Chapter 10 discusses *splitting*, a more sophisticated technique for exploiting union type information created by merges in the control flow graph without run-time overhead. Where the compiler cannot apply splitting, such as for unions created as the result of primitive operations, it falls back to this type casing technique.

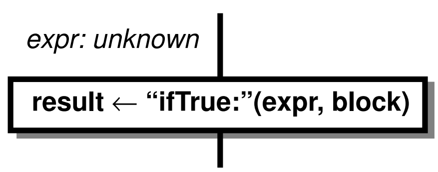
Type casing illustrates that a union type provides more information than just specifying the set union of the objects specified by its component types. The divisions of the various component types is also important, since it is these component types that drive type casing. For example, if prior to a merge node a name is bound to the integer map type along one predecessor and to the unknown type along the other predecessor, as part of type analysis the compiler will bind the name to the union of these two types after the merge: *{integer, unknown}*. A naive implementation would "simplify" this union type by noting that the unknown type covers the integer map type, and so the integer map type could be removed from the union without altering the set of objects specified by the type: *{unknown}* or simply *unknown*. However, such a simplification would lose a significant amount of important information. The compiler no longer would have the information that integers are a likely component of the type, and so the compiler no longer could separate out the integer case via type casing. If faced with a blank unknown type, the compiler would have no information upon which to decide that some subset of the possible types would be worth type casing for. Accordingly, union types in the SELF compiler are never "simplified" by eliminating component types covered by another component type, to preserve as much useful information as possible.

9.4 Type Prediction

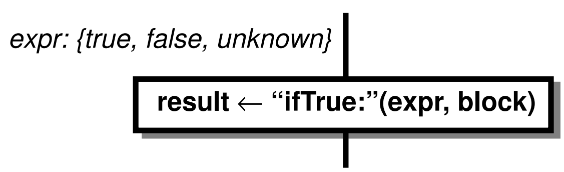
Customization and type analysis enable the compiler to infer the representation-level types of many expressions, which in turn enables the compiler to statically-bind and inline many messages and optimize away many run-time type checks. However, many messages remain whose receiver types cannot be inferred using only customization and type analysis. For example, the types of arguments are unknown (the compiler currently does not customize on the types of arguments), as are the types of the contents of assignable data slots in the heap (e.g., instance variables), so messages sent to these objects cannot be inlined using only customization, type analysis, and type casing.

To enable the compiler to infer even more type information, the SELF compiler performs *type prediction*. If the compiler cannot infer the type of the receiver of a message using customization or type analysis, then the compiler tries to use the *name* of the message as a hint about the likely type of the receiver. For example, the compiler predicts that the receiver of a **+**, **<**, or **to:Do:** message is likely to be an integer, the receiver of an **at:** message has a good chance of being an array, and the receiver of an **ifTrue:** message is almost certainly either **true** or **false**. These predictions are embedded in the compiler in the form of a small fixed table mapping message names to likely receiver types. A message whose name is not in the table is not type-predicted, and so must be implemented as a full message send in the absence of other optimizations.

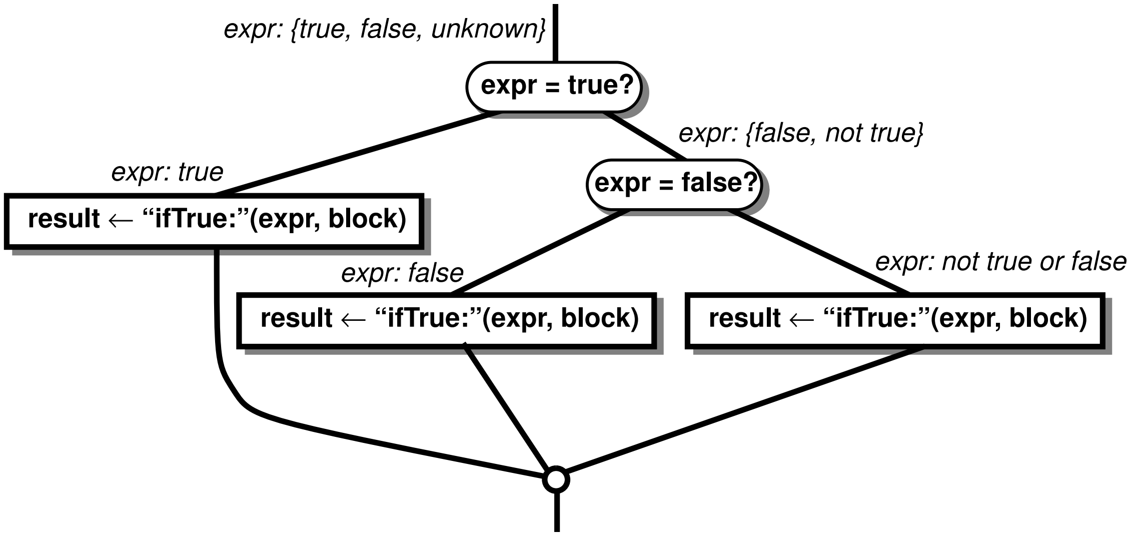
The compiler uses the predicted type(s) to transform the original receiver type into a union type, one of whose components is the original receiver type and whose other component(s) are the predicted type(s). For example, if the **ifTrue:** message is sent to an expression whose type is unknown:



then the compiler can perform type prediction and replace the receiver type with one that contains the **true** and **false** constant types:



Since the original type is still part of the union type, the receiver's type remains just as general as it was before type prediction. However, the form of the receiver's type is now suitable for further optimization via type casing: a run-time type test is inserted to check for each predicted type and then branch to a separate statically-bound copy of the message suitable for inlining; a fall-back dynamically-bound version of the message will remain in case none of the predictions are correct. In the **`ifTrue:`** example, after type prediction the compiler would apply type casing and insert run-time checks for **`true`** and **`false`** that each branched to a separate copy of the **`ifTrue:`** message:



Two copies of the **`ifTrue:`** message can be inlined away, since their receiver types are both compile-time constants; the third copy remains dynamically-bound since its receiver type is unknown. Control flow re-merges after the type prediction and type casing transformations.

If the rate of successful prediction is high, the performance of a predicted message can be much faster than the original unpredicted message. The cost of a type test is less than the cost of a dynamically-dispatched procedure call, and inlining predicted messages can lead to opportunities for additional time-saving optimizations. Of course, since the unsuccessful branch is slowed down by an extra run-time type test, using type prediction in cases with a low success rate can actually slow down the overall performance of the system.

In the current SELF system, type prediction has a high success rate. In the benchmarks used to measure the performance of SELF, almost all predictions are correct. This is because the benchmarks were originally written in traditional languages such as Pascal and BCPL, and the data types predicted using type prediction (integers, booleans, and arrays) are those that are normally the only ones used in traditional languages; the only mispredictions occur in benchmarks translated from Lisp which overload **`=`** to compare both integers and **`cons`** cells. However, even in large Smalltalk systems, the receiver of a message like **`+`** is an integer 90% to 95% of the time [Ung87], so type prediction is useful even for programs written in a heavily object-oriented style. As reported in section 14.3, type prediction speeds SELF programs by a factor of 3 on average, with object-oriented SELF programs benefitting almost as much as more traditional, numeric benchmarks.

Type prediction is similar to (and was inspired by) the technique in Smalltalk-80 systems that hard-wires the implementations of certain common messages into the compiler, as described in section 3.1.1. Both insert run-time type tests to verify static predictions embedded in the compiler. However, unlike Smalltalk's hard-wiring, type prediction does not embed the *definition* of predicted messages into the compiler (since inlining is used instead), nor does the compiler impose any *restrictions* on the use of predicted messages such as **`ifTrue:`** and **`==`**. Programmers are always free to change the definitions of predicted messages and to add new definitions that did not exist when the compiler was implemented. Type prediction coupled with inlining enables the SELF implementation to achieve the same run-time performance as hard-wired messages and still preserve SELF's pure message passing model.

Type prediction as currently implemented is a static technique: the message names and predicted receiver types is fixed in the compiler and cannot be changed by users. A better technique would be *dynamic type prediction*, where the message names and predicted receiver types would automatically adapt to the SELF source code currently in use. For example, dynamic profile data could be used to augment or replace the static table built into the compiler. We are actively investigating techniques that would make type prediction more adapting to changing usage patterns [HCU91].

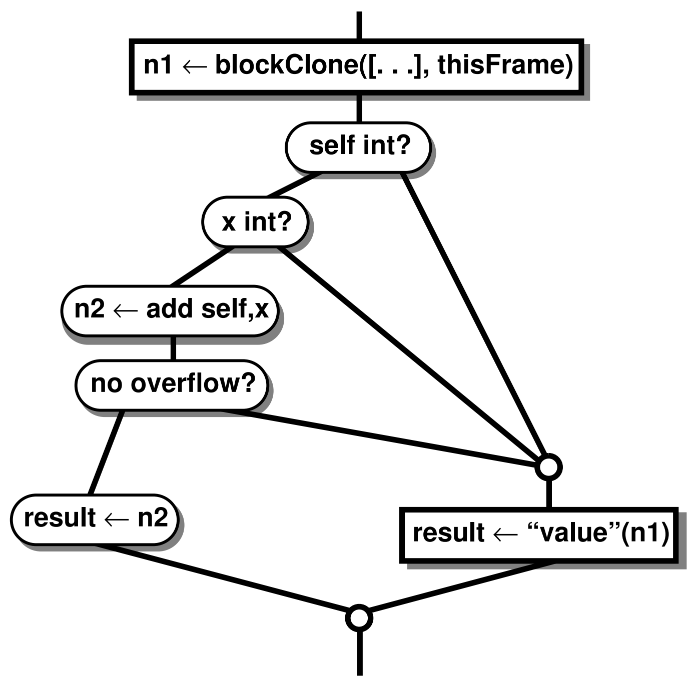
9.5 Block Analysis

Blocks are very common in SELF code, primarily because of their central role in the implementation of user-defined control structures. In a straightforward SELF implementation, each invocation of a control structure like **`to:Do:`** (SELF's version of a traditional **`for`** loop) would involve creating several blocks at run-time and invoking these blocks repeatedly during the execution of the loop. This approach to compiling user-defined control structures and blocks would never be competitive in run-time performance with traditional languages based on built-in control structures, where the compiler can generate only a few instructions to implement a control structure. Consequently, much of the SELF compiler's efforts are directed towards eliminating the overhead of user-defined control structures and blocks.

9.5.1 Deferred Computations

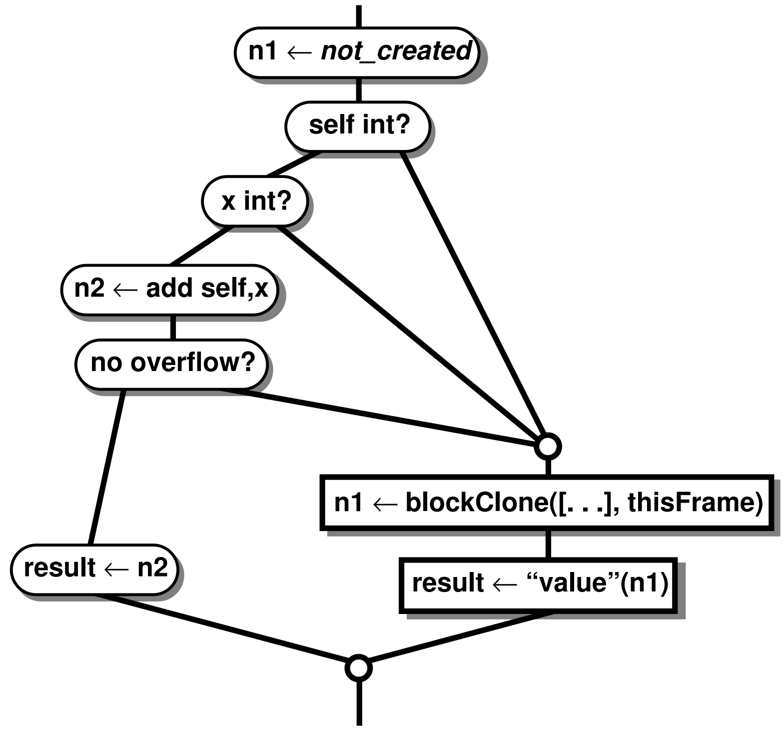
Several important block-related optimizations have already been mentioned in section 7.1: inlining statically-bound block **`value`** methods (and eliminating block creation operations if all uses of the block are eliminated) and preferentially inlining methods with block arguments to increase the likelihood of the block arguments getting inlined away. However, many blocks may have a few remaining uses that cannot be eliminated, thus requiring that the block be created, even if those uses are only on control flow paths that are executed rarely.

For example, consider an inlined integer addition primitive operation that is passed a failure block. If the primitive fails, then the block is sent the **`value`** message, and so if this message is not inlined (as it is not in the current SELF system) one use of the block remains and the block creation code cannot be completely eliminated.



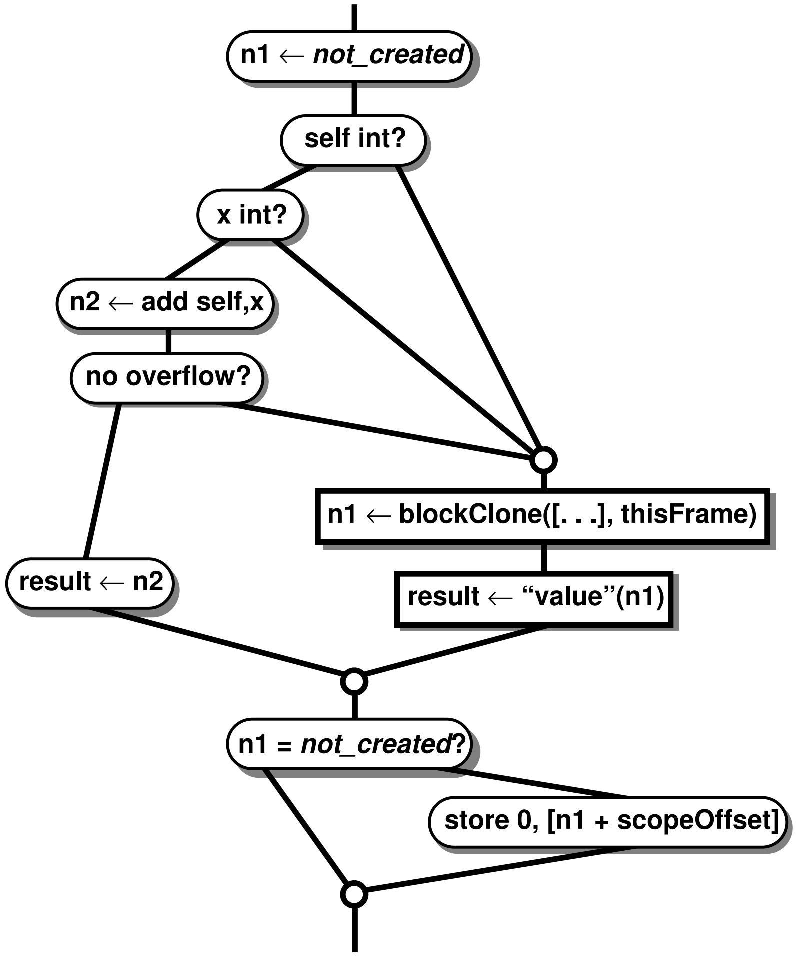
However, since most primitive invocations do not fail, the large majority of the invocations of the primitive do not use the created block. The overhead of creating a block for every primitive invocation, especially ones as simple as integer arithmetic, can quickly bring a system to its knees.

To limit the overhead of the block creation to just those parts of the program that need the block, the compiler *defers* the creation of blocks until they are actually needed as a run-time value. Then only those paths through the control flow graph that need the block will pay the expense of creating the block; other paths are not penalized.

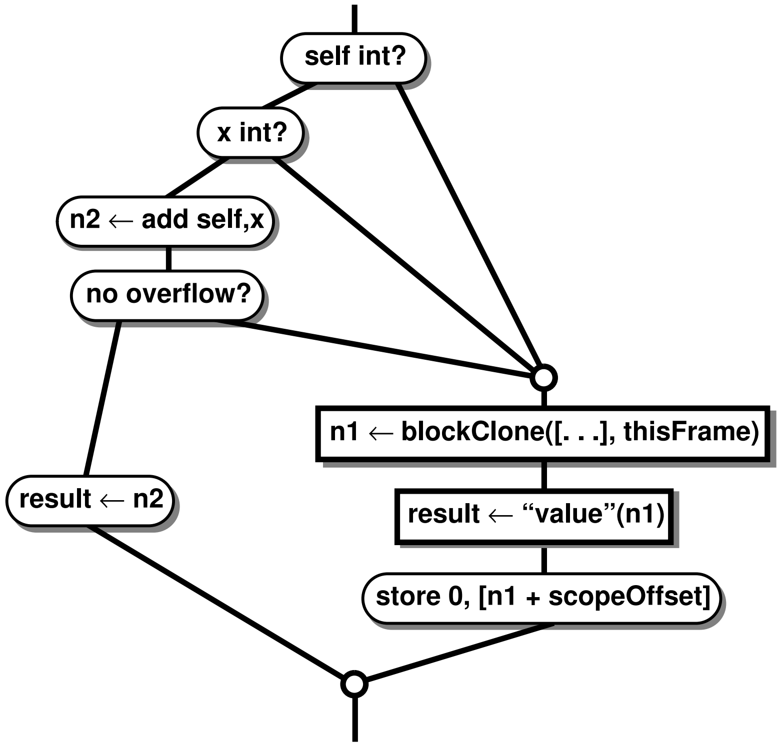


The compiler needs some way to tell whether a block has been created or not at some point in the program. This information is encoded in the type associated with the block value: a block's type is either a *deferred block type* or a *created block type*. In place of the original block creation code, the compiler generates an assignment node that binds a fresh value object to the deferred form of the block type. Any later uses that require the block to be a real run-time value, such as a non-inlined message send with the block as an argument, check whether the type of the block is deferred and if so insert additional nodes in the control flow graph before the use to create the block object at run-time. The block creation node rebinds the value object representing the block from the deferred block type to the corresponding created block type; the name/value binding of the created block remains unchanged so that all names aliased to the same block value simultaneously see that the block is now created, thus avoiding duplicate block creations.

Deferred computations such as block creation are handled at the same time other type analysis and optimizations are performed so that whether or not an expensive computation has been performed can influence the control flow graph that gets built. This is especially important for optimizing block zapping code (see section 6.3.2) out of those paths of the control flow graph where the compiler knows the block is never created. In an earlier version of the SELF compiler, deferred computations were handled in a later pass of the compiler after the bulk of the control flow graph transformations had been performed. This design led to poor performance, since some extra run-time tests had to be inserted to check whether the block creation had been performed before zapping the block. For example, in the following graph which includes the code to zap the failure block, even though the block creation is deferred to the primitive failure branch, the block zapping code remains even for paths where the primitive succeeds.



Handling deferred computations as part of type analysis allows other techniques that cooperate with type analysis, such as splitting (described in Chapter 10), to help optimize the treatment of blocks and block zapping so that only blocks that have been created need zapping code. This enables the block zapping code to be executed only along the branches where the block is created.



Optimizing away block creations when unnecessary, and deferring remaining block creations until just prior to their uses, is one of the most important optimizations performed by the SELF compiler. As reported in section 14.3, without deferred block creation SELF would run an average of 4 times slower, and more that 10 times slower for programs rife with user-defined control structures such as **`to:Do:`** loops.

Other computations such as arithmetic are also deferred until their first uses. All that is required of a deferred computation is that it have no externally-visible side-effects. Opportunities for deferring computations other than block creations occur far less often, however, and do not produce the same sorts of dramatic speed-ups as when deferring block creations.

9.5.2 Analysis of Exposed Blocks

Blocks support lexical scoping of local variables: a nested block can contain accesses and assignments to arguments and local variables in a lexically-enclosing scope. These accesses and assignments are called *up-level accesses* and *up-level assignments*. If a block's **`value`** method is inlined, up-level accesses and assignments in the **`value`** method can be treated as local accesses and assignments, and type analysis can be used to infer the types of these variables. An inlined block **`value`** method's non-local return can also be implemented as a direct branch to another part of the control flow graph if the block's outermost lexically-enclosing scope is also inlined within the same compiled method.

However, when a block cannot be optimized away and is instead passed as the receiver or argument to a non-inlined message, the compiler must become more conservative. Since the non-inlined message might invoke the block, the types inferred for any local variables potentially up-level assigned from within the block's **`value`** method must be weakened to include the unknown type; the current SELF compiler does no interprocedural analysis to infer the types of expressions up-level assigned from within non-inlined blocks. Also, since the non-inlined method might store the block in a long-lived global data structure, all subsequent non-inlined messages must be assumed to invoke the block, again requiring the types of up-level assigned variables to be weakened. We call blocks that might be invoked by other methods at any call site *exposed blocks*, since they have been exposed to the outside world and are no longer under tight control.

Since exposed blocks dilute the type information of potentially up-level assigned local variables, the compiler works hard to limit the number of blocks that must be treated as exposed. In addition to maintaining the name/value and value/type bindings during type analysis, the compiler maintains the set of blocks that have been exposed. A block is added to the current exposed blocks set if and when it is passed out at a non-inlined message send or stored into a data structure in the heap. In addition, all blocks up-level accessible from a newly exposed block must also be added to the exposed blocks set, since they might be accessed and invoked whenever the original block is invoked. A block is removed from the exposed blocks set once its lexically-enclosing scope returns (since the block will be zapped and unusable). At merge points, the compiler unions together the exposed blocks sets of the merge's predecessors to form the set of exposed blocks for the merge's successor, much as the compiler forms union types at merge nodes.

An exposed block set is used at a non-inlined message send node to alter the type bindings of all potentially up-level assigned variables of all exposed blocks in the set. When calculating the mappings for the message send's successor, each potentially up-level assigned name is rebound to its own new, unique value object, modelling an assignment to the name of an unknown object from within the exposed block.

The compiler must also somehow generalize the type associated with the new value object, since if the assignment does occur the compiler does not know what type of object will be assigned to the local variable. A simple strategy would simply bind the new value object to the unknown type. Unfortunately, such a naive approach would sacrifice any type information the compiler had inferred about the local variable prior to the message. The SELF compiler therefore tries to limit the damage to type information caused by a potential up-level assignment based on two heuristics. First, many potential up-level assignments will not actually be performed, so any information accumulated about the contents of the local variable would still be true after the message send. Second, of those assignments that are performed, the type of the variable after the assignment is likely to be similar to the type of the variable before the assignment, say in the same clone family; programs do not normally assign completely unrelated objects to the same local variable.

To exploit these trends, the SELF compiler assigns the type of a local variable after a potential up-level assignment as the union of the unknown type (in case the assignment actually occurs) and the original type of the local variable before the message send, generalized to the enclosing map type (in case the assignment does not occur or an assignment to a member of the same clone family occurs). For example, if the type of a potentially up-level assigned variable were *{[0..5], float}* before the message send, after the message send the type would be changed to *{unknown, integer, float}*. This union type information then can be exploited using type casing (see section 9.3). If the local variable is not assigned, or is assigned a member of the same clone family, this strategy incurs the overhead of only a type test upon subsequent accesses to the local variable, rather than requiring a full message send as would the naive strategy.

Exposed block sets improve the quality of type analysis by limiting the damage of up-level assignments to those parts of the control flow graph where blocks were actually created and exposed to the outside world. Since many blocks are only created and exposed along relatively uncommon branches, such as primitive operation failure blocks, the effect of these exposures is limited to those parts of the graph. This containment allows the common-case branches to execute without the conservative assumption that some local variables might have been assigned. In practice, exposed block analysis is very effective; as shown in section 14.3, with exposed block analysis SELF programs run almost 50% faster than they would if the compiler treated all blocks as exposed.

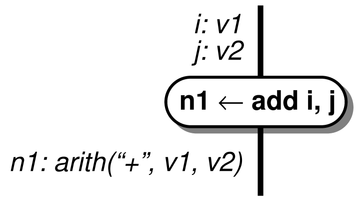
9.6 Common Subexpression Elimination

The compiler performs *common subexpression elimination* (*CSE*) as part of type analysis. CSE eliminates redundant computations by detecting when a computation has already been performed and its earlier result can be reused. The SELF compiler performs CSE on two kinds of computations: arithmetic calculations and memory references (loads and stores).

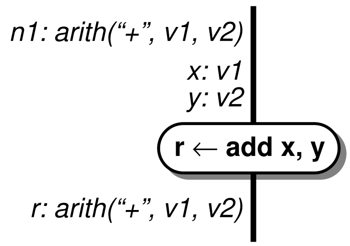
9.6.1 Eliminating Redundant Arithmetic Calculations

The compiler discovers redundant arithmetic calculations by searching the name/value mappings for an existing value that is the same as the result value of the potentially redundant calculation. If such a value already exists in the name/value mapping, then the compiler replaces the arithmetic calculation with a simple assignment of a name bound to the existing value to the name of the result of the eliminated calculation. To support equality comparisons of values, values representing the results of arithmetic calculations are structured, containing subcomponents for the receiver and argument values and the kind of arithmetic calculation. Two arithmetic values are guaranteed equal if and only if they have equal subcomponents.

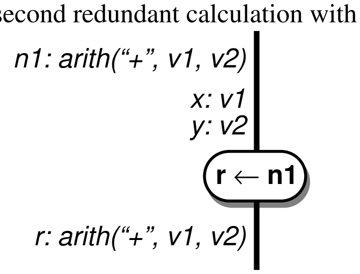
For example, the following **add** control flow graph node produces a structured result value:



If some later node in the graph calculates the same value:



then the compiler can replace the second redundant calculation with a simple assignment node:



The compiler typically can avoid generating code for these assignments by arranging that all names related by simple assignment are allocated to the same register (register allocation is described in section 12.1).

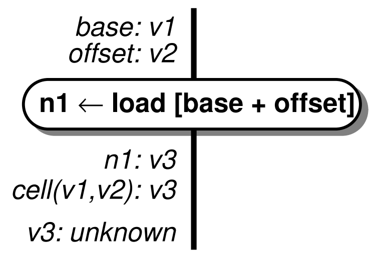
Currently the compiler detects equal calculations only if the operations are equal, the operands are equal, and the order of operands is the same, i.e., if the operation/operand trees are isomorphic. More sophisticated value equality systems would take into account arithmetic identities, such as the commutative property of addition and the relationships between addition and subtraction. This weakness in the current SELF compiler's rules actually makes a difference in the quality of generated code, although not a large difference. Additionally, values could be extended to enable equality testing even in the presence of conditional branches; the algorithms associated with subscripted names and SSA form, described in section 3.4.4, support such flow-sensitive equality testing. It remains an open question, however, how much practical benefit would be received from such additional analysis.

As reported in section 14.3, CSE of arithmetic expressions has a relatively minor effect on run-time performance, usually less than a 5% improvement in performance. Redundant arithmetic computations probably are less common in SELF programs than in traditional programs, in part because array accesses in SELF do not require multiplication of the array index by a scaling factor as in other languages, since SELF's tagged integer representation is already appropriately scaled for indexing into SELF's built-in one-dimensional object vectors. Also, other calculations that could be eliminated as redundant currently are not because of limitations in the garbage collector's treatment of derived pointers.

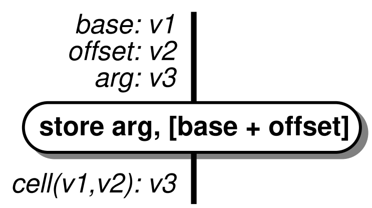
9.6.2 Eliminating Redundant Memory References

Redundant memory fetches and stores are detected in a fashion similar to detecting redundant arithmetic calculations. The compiler maintains a mapping from *cells* to values that is propagated through the control flow graph as part of type analysis. Cells are the compiler's internal representations of memory locations in the heap, such as assignable data slots (instance variables), array elements, and the lexically-enclosing frame pointers of blocks. The value object associated with the cell represents the current contents of the cell. A cell is "addressed" by two component values: a base value and an offset value. The base value is the value of the object being accessed by the memory reference; the offset value is either a constant (for fixed-offset memory references such as accesses to assignable data slots and frame pointers) or a normal computed value (for computed indexes into arrays). Two cells are guaranteed to refer to the same physical memory location if their corresponding base values and offset values are equal. Of course, two cells with different base and offset values *might* still refer to the same memory location, however, since two values that are not guaranteed to refer to the same run-time object might still do so (see section 9.1.3).

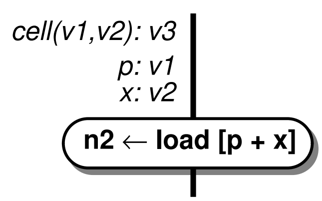
CSE of memory references works in much the same way as CSE of arithmetic calculations. Memory load instructions construct a new value object representing the contents of the cell and a new cell object addressed by the base and offset of the load instruction to the new contents value object. Three bindings are added to the mappings maintained by type analysis: a mapping from the name of the result of the memory load to the new value object is added to the name/value table, a mapping from the value to the unknown type is added to the value/type table, and a mapping from the new cell to the new value is added to the cell/value table.



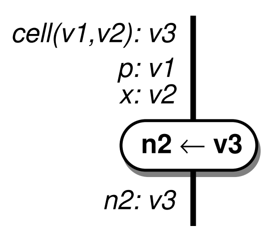
Memory store instructions similarly add a binding from a new cell object addressed by the base and offset of the store instruction to the value bound to the name of the object stored into the addressed memory location.



If at a later memory fetch node the same cell is already in the cell/value mapping table



then the compiler can replace the redundant memory fetch node with a simple assignment node, binding the result name of the redundant memory reference to the value associated with the existing cell.



In addition to eliminating an unnecessary memory reference, this optimization also preserves any type information the compiler has been able to infer about the contents of the memory cell. In some cases, the benefits from preserving this type information outweigh the benefits from merely eliminating an instruction. If CSE were performed at a later stage in the compilation process, as is done in other compilers for traditional languages, then this ability to preserve type information of memory cells would be lost. In effect, CSE of memory accesses supports type analysis of locations in the heap.

The compiler can eliminate a memory store node if the cell being assigned is already in the cell/value mapping and if the current contents value bound to the cell in the mapping is the same value as the one being stored. Otherwise, the store cannot be eliminated, and the compiler updates the cell/value mapping after the store node to reflect the compiler's knowledge about the cell's new contents. Additionally, all cells already in the mapping that *might* refer to the same memory location as the stored cell (i.e., any potentially aliasing cells in which the base and offset values *might* be the same) must be removed from the cell/value mapping, since their contents are now ambiguous. Similarly, non-inlined message sends must be assumed conservatively to assign to all global heap cells that could be assigned. Therefore, at each non-inlined message send, all cell/value bindings for cells that could be changed by a SELF program (including instance variables and array elements but excluding frame pointers and array sizes) must be removed from the cell/value mapping; such cells could be called "exposed" cells in analogy with exposed blocks.

To avoid losing precious type information, such potentially modified cells are treated in much the same way as potentially up-level assigned local variables (described in section 9.5.2). Each potentially-modified cell is given a new unique value object and added back into the cell/value mapping; the new value object is bound to the union type constructed by generalizing its previous inferred type to the enclosing map type and combining this type with the unknown type. In this way, the damage to type information from assignments to potential alias cells and potential assignments to "exposed" cells can be limited.

CSE of memory cells is used for another purpose in the SELF compiler: eliminating unnecessary array bounds checks. At each array access the compiler checks to see whether the cell corresponding to the accessed array element is already in the cell/value binding. If so, then the compiler omits the code that would have checked whether the array access was in bounds. This optimization is legal because the program must have been able to access the array cell before without error, since the cell is available; the bounds check must already have been performed as part of the previous access. Theoretically, this optimization should be unnecessary, because the compiler should be able to eliminate the bounds check more directly by remembering that the bounds check already had been performed for the same array index. However, the current implementation of the SELF compiler does not record this information. Checking for CSE of memory cells is therefore a cheap way of eliminating some redundant array bounds checks without much additional mechanism. In the future, however, the SELF compiler should include enough information to be able to determine when array bounds checks have already been performed (or, even better, when no array bounds checks are required at all at run-time), and this technique will be removed as redundant.

As reported in section 14.3, common subexpression elimination of redundant memory references is more effective for SELF than common subexpression elimination of redundant arithmetic. While the average performance improvement from CSE of memory references is around 5%, some benchmarks speed up by more than 30% from this technique. The effect of being able to track type information through assignments and subsequent fetches from memory cells accounts for a sizable fraction of the total contribution of CSE of memory references; eliminating array bounds checking using available cell information accounts for a smaller fraction.

9.6.3 Future Work: Eliminating Unnecessary Object Creations and Initializations

Ideally, the compiler could eliminate some object creations and stores if all uses of the object (such as memory fetches out of the created object) were eliminated. As an example of a situation where such throw-away objects get created, a quadratic formula function might return multiple roots by creating an object with a pair of assignable data slots, store the result roots into the object, and then return the object:

```
quadraticFormulaA: a B: b C: c = ( | result. temp. |
  temp: (b squared - (4 * a * c)) sqrt.
  result: ( | r1. r2. | ) _Clone. "create an object to hold the roots"
  result r1: (b negate + temp) / (2 * a).
  result r2: (b negate - temp) / (2 * a).
  result ).
```

The caller of the quadratic formula routine would extract the roots out of the result object's data slots and then throw the result object away:

```
printRootsA: a B: b C: c = ( | result |
  result: quadraticFormulaA: a B: b C: c.
  (result r1 printString, ' & ', result r2 printString) printLine.
).
```

If the called routine were inlined into the calling routine, then the memory fetches that extract the data slots of the returned object would be optimized away as redundant, since the compiler would have recorded the earlier stores to the same memory cells. Thus, the intermediate object no longer would be useful to the execution of the program, and we would like the compiler to eliminate the object creation and the memory stores as unnecessary.

As another example, consider the standard **`to:Do:`** looping control structure. This control structure takes two integers and a block as arguments and iterates through all the integers between the two integer arguments, invoking the block for each integer. Perhaps a better way of defining a **`for`** loop would apply the standard **`do:`** control structure (which iterates over an arbitrary collection) to an interval object (which represents the collection of all integers between its lower and upper bounds). In SELF, an interval can be created by sending the **`to:`** message to an integer with an integer argument, so **`for`** loops could be written as follows:

```
(lowBound to: upperBound) do: [ "body of the loop" ].
```

Intervals are implemented as follows:

```
traits interval = ( |
  parent* = traits collection.
  ...
  do: aBlock = ( | i |
    i: lowerBound.
    [ i <= upperBound ] whileTrue: [
      aBlock value: i.
      i: i successor.
    ].
    self ).
  ...
| ).
interval = ( |
  parent* = traits interval.
  lowerBound.
  upperBound.
| ).
```

with the interval creation code defined for integers:

```
traits integer = ( |
  ...
  to: upperBound = (
    (interval clone lowerBound: self) upperBound: upperBound ).
  ...
| ).
```

This design would economize on the number of concepts needed for normal SELF programming; **`do:`** is a well-known operation in SELF, and intervals are a useful data structure in their own right. Programmers would not need special iterator methods just for integer loops.

In typical usage, this style of **`for`** loop would create a new interval object for each invocation of the loop. If performance comparable to a traditional language's **`for`** loop built-in control structure is desired, the overhead of creating this interval object must be reduced. Fortunately, the interval object is created and then almost immediately thrown away after the **`do:`** method for intervals extracts the lower and upper bounds of the iteration, with no remaining run-time uses, and so we would hope that the compiler could optimize away the object creation overhead entirely.

Unfortunately, the SELF compiler currently cannot optimize away object creations and stores. Eliminating object creates is complicated by source-level debugging. If the debugger is invoked when the object is in scope and therefore visible on a stack dump, the compiler must provide enough information for the debugger to at least create the illusion at debug-time that a real object was created and initialized. This problem could be avoided in some cases if the compiler was able to determine whether the object could ever be visible at debug-time; this analysis would be much like determining whether the created object was ever "exposed" to the outside world. The compiler could eliminate any stores into an un-exposed object (since those stores could never be seen by other routines or the SELF programmer) and subsequently eliminate the object creation code itself (since all uses of the created object would be gone).

With the current SELF coding style, this optimization is not crucial for good performance, although it would be helpful. However, if the new style of **`for`** loops using intervals and **`do:`** were to be implemented efficiently, or if other aspects of SELF programming style were to change, then this optimization would be needed to maintain the current high level of run-time performance.

9.7 Summary

The SELF compiler uses type analysis to propagate information about the types of variables and expressions through the control flow graph, to maximize the benefits received from the relatively scarce type information that the compiler can infer. The compiler maintains several data structures as part of type analysis. The mappings from names to values and from values to types are central, being used to determine the type of the receiver of a message (in support of compile-time message lookup and inlining) and the type of arguments to a primitive (in support of eliminating run-time type-checking overhead). Type testing code alters the value/type mapping, implicitly altering the induced name/type mapping for all names aliased to the tested value, as is necessary in a system relying on aggressive inlining.

Type casing exploits the information contained in union types by inserting run-time type tests that branch to monomorphic versions of code that are amenable to compile-time message lookup and inlining. Type prediction uses context information in the form of the names of the message sent to an expression to make a prediction about the likely type of the expression. This prediction is exploited by replacing the original type of the predicted expression with a union type that contains both the predicted type(s) and the original type. Type casing then is employed to read the new union type information and insert the appropriate run-time type tests that verify the prediction and branch to optimized code in the case that the prediction is correct.

The compiler optimizes blocks as part of type analysis, primarily by deferring the creation of a block until its first run-time use, if any. The compiler also calculates which blocks have been “exposed” to the outside world, and weakens the types of only those variables that are potentially up-level assigned by exposed blocks.

The name/value mapping is also used to support common subexpression elimination of redundant arithmetic calculations, since the result values of arithmetic calculations are structured and so can be compared for equivalence. An additional mapping from cells to values supports common subexpression elimination of redundant memory fetches and stores, in turn allowing type analysis to track the types of values stored into and later fetched out of assignable data slots in the heap.

Chapter 10 Splitting

This chapter describes *splitting*, a technique for transforming polymorphic code into multiple copies of monomorphic code amenable to further optimization such as inlining. Splitting thus resembles type casing (described in section 9.3) but introduces no additional run-time overhead for type tests. This chapter discusses splitting of straight-line code; splitting of loops will be discussed in Chapter 11.

10.1 A Motivating Example

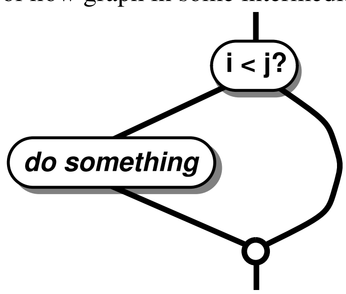
Splitting was originally motivated by attempting to generate good code for the following expression:

```
i < j ifTrue: [ do something ]
```

Expressions like this occur in virtually all programs, and so it is imperative to generate good code for this example. A reasonable C compiler, when faced with the similar C code:

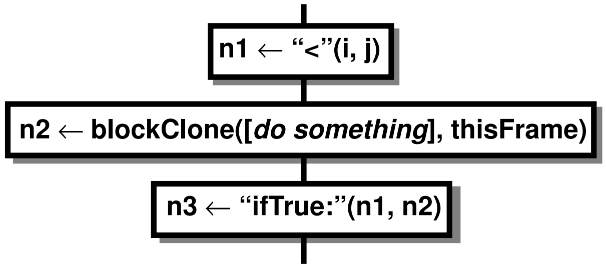
```
int i, j;
...
if (i < j) { do something; }
```

would produce the following control flow graph in some intermediate step:



The techniques presented so far can come close to this graph, but not quite.

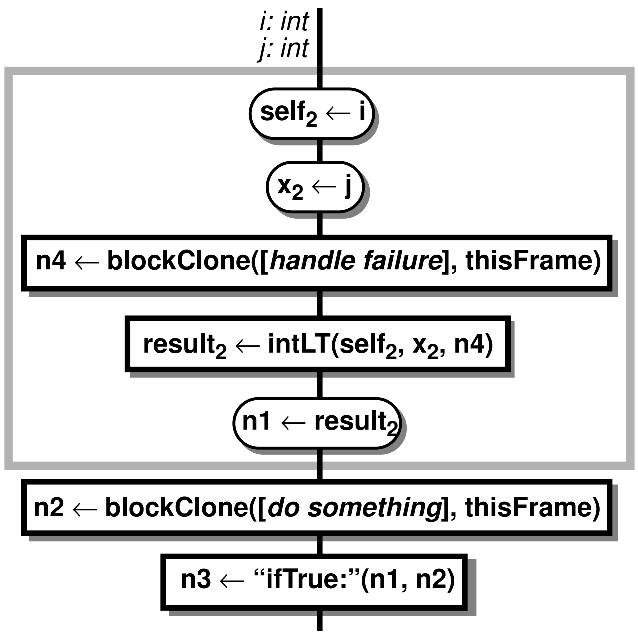
The SELF compiler would begin with the following control flow graph representation of the original conditional expression:



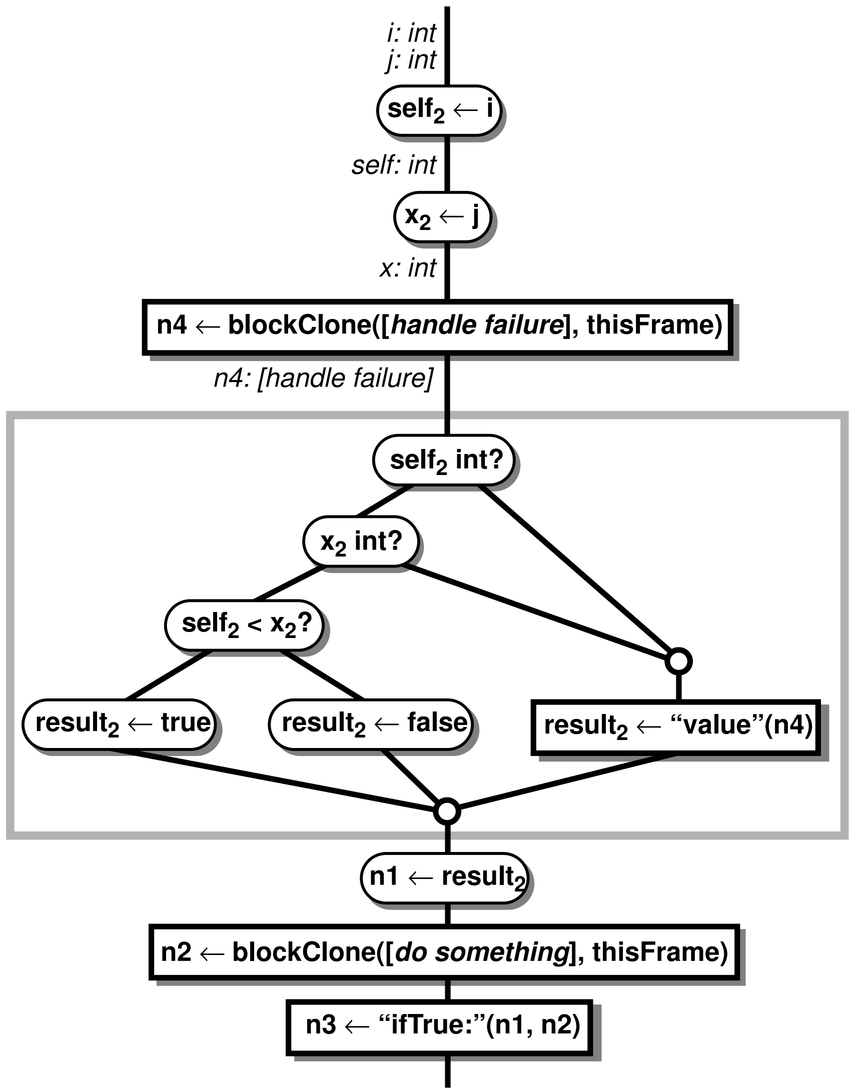
To simplify the example, assume that the SELF compiler is able to infer via type analysis that `i` and `j` are integers. Then the compiler can lookup the definition of `<` for an integer receiver to find the following method:

```
< x = ( _IntLT: x IfFail: [ handle failure ] ).
```

The compiler can inline expand this method in place of the `<` message to produce the following control flow graph:

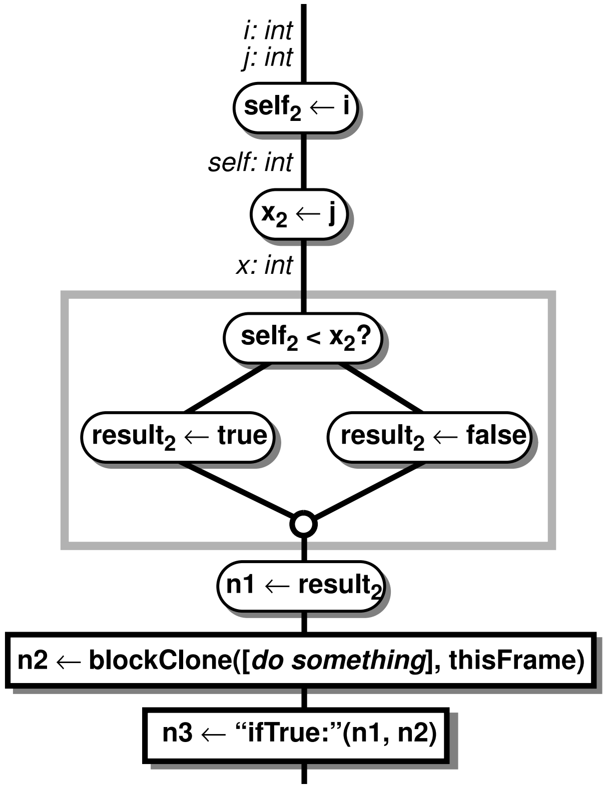


Type analysis proceeds, eventually reaching the call to the **`intLT`** primitive and expanding it in-line.

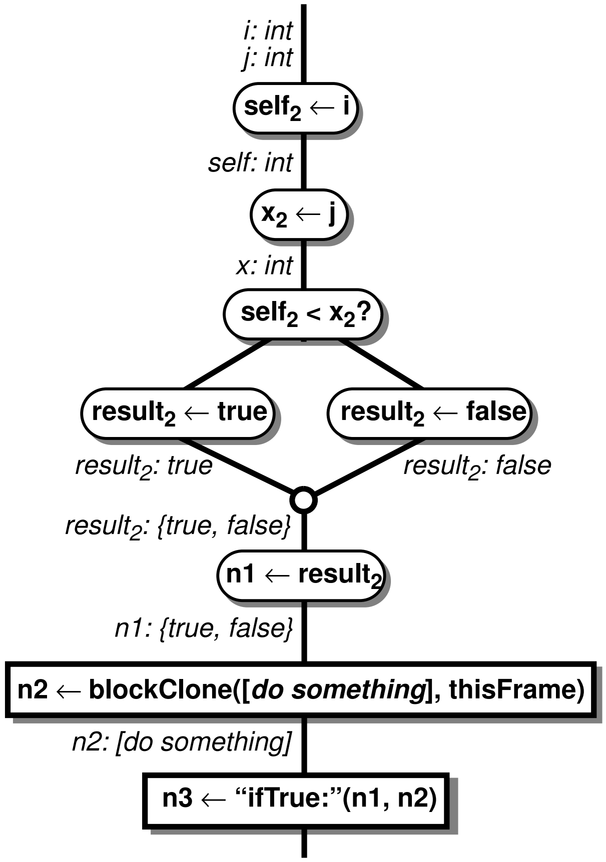


The compiler can eliminate the type checks on the arguments to the **`intLT`** primitive, since the compiler knows through type analysis that both **`self`** and **`x`** are integers. Subsequently, the compiler can eliminate the creation of the

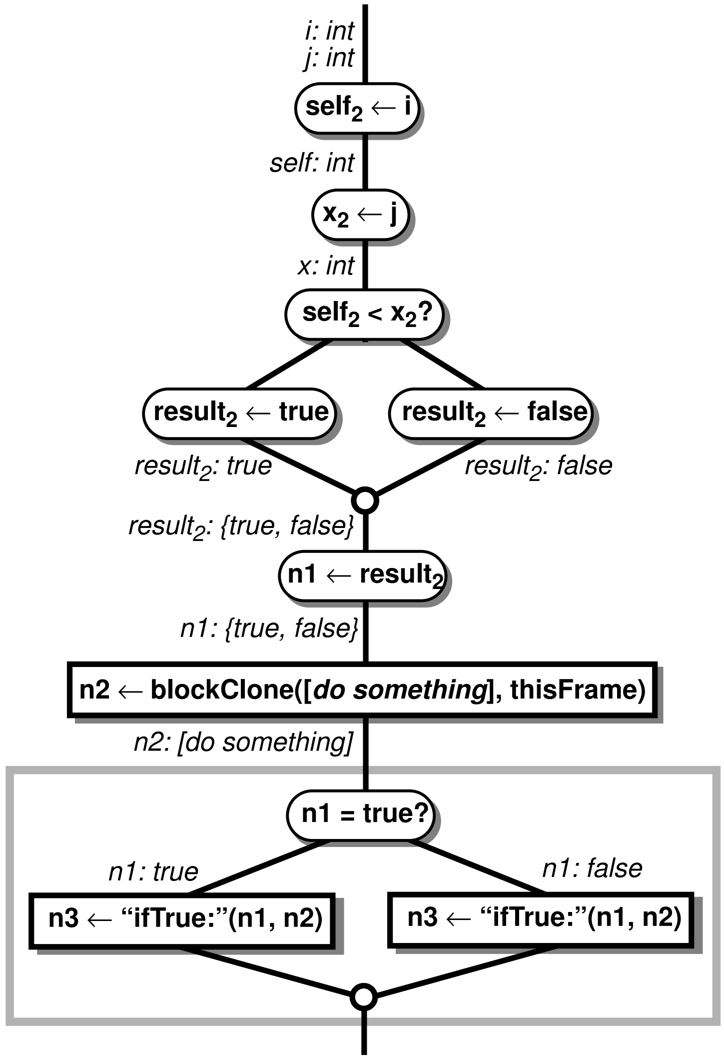
[*handle failure*] block, since it is no longer needed as a run-time value. These optimizations produce the following control flow graph:



Type analysis starts again, analyzing the body of the **`intLT`** primitive, and eventually reaching the **`ifTrue:`** message with the knowledge that the type of the receiver of **`ifTrue:`** is *{true, false}*:



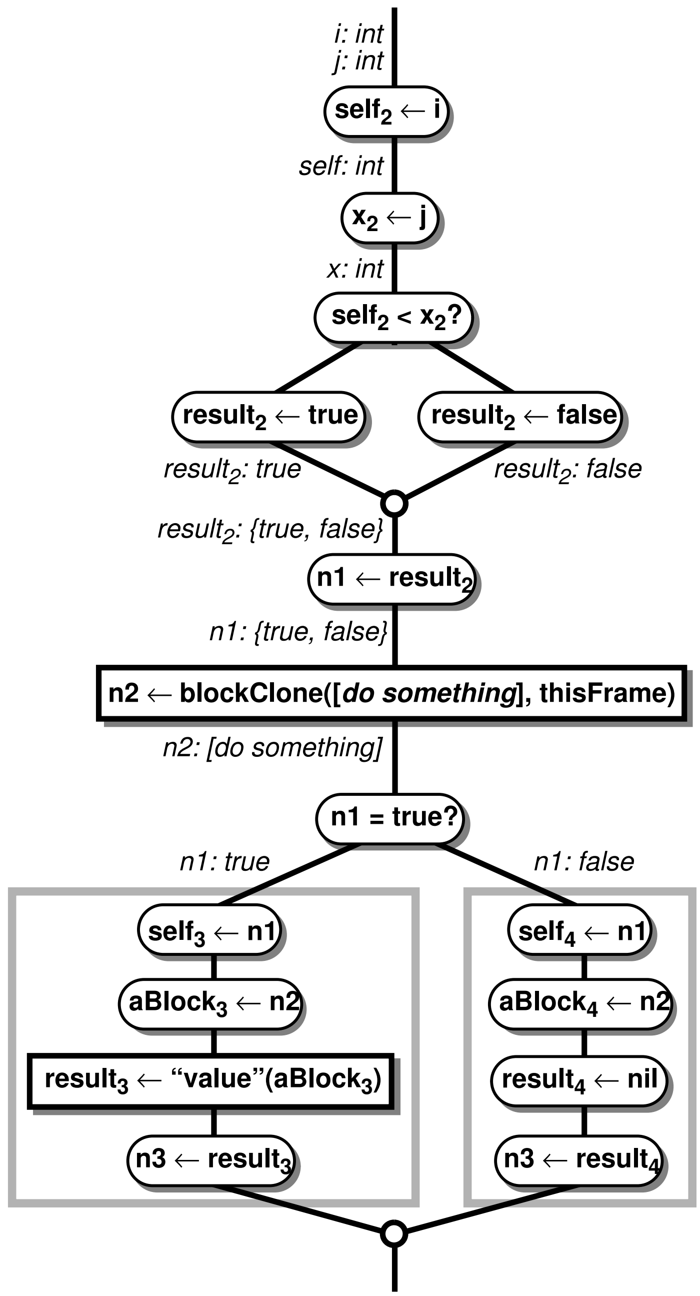
At this point, the compiler could use type-casing to insert a run-time type test to separate the **`true`** and **`false`** cases:



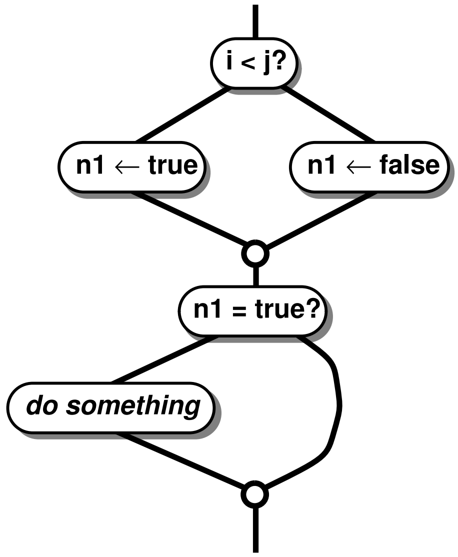
The compiler then can perform message lookup at compile-time for the two copies of the **`ifTrue:`** message, locating the following methods:

```
true = ( |
  ...
  ifTrue: aBlock = ( block value ).
  ...
| ).
false = ( |
  ...
  ifTrue: aBlock = ( nil ).
  ...
| ).
```

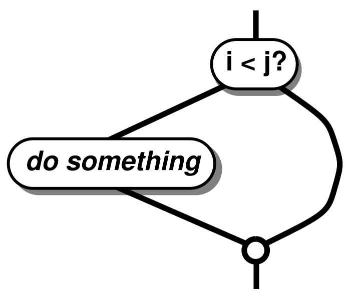
The compiler would inline these methods into the control flow graph:



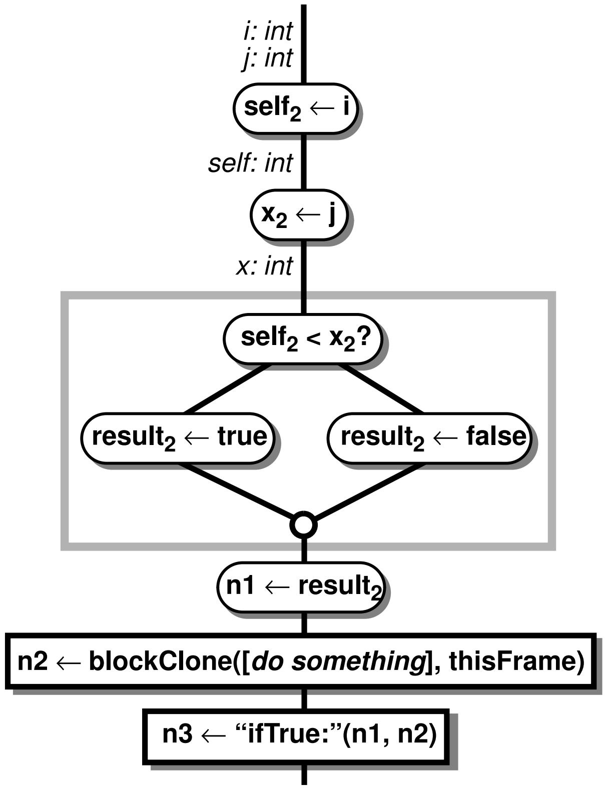
The compiler restarts type analysis, determines that the type of **`aBlock`**$_3$ is a particular cloned block literal, and inlines the block's **`value`** message down to just the body of the block; the block creation code can then be eliminated since there are no remaining uses of the block. This produces the following final control flow graph (after removing the assignment nodes which do not correspond to generated instructions):



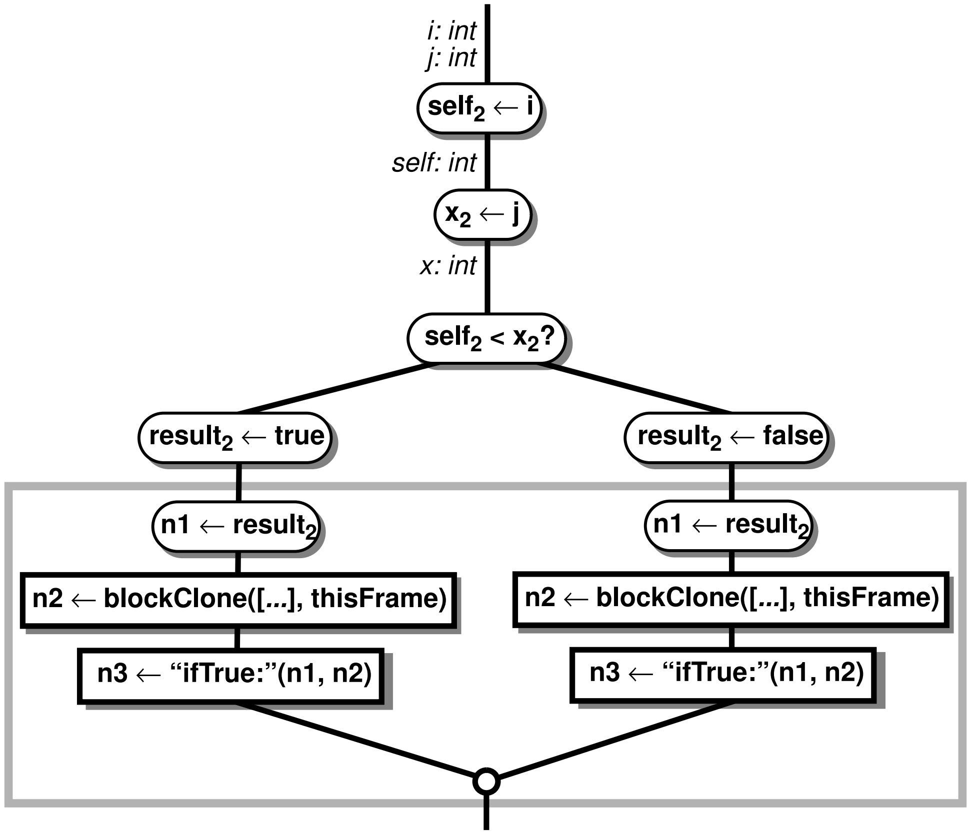
Unfortunately, this type casing approach produces code that is less efficient than the single compare-and-branch sequence generated by the C compiler:



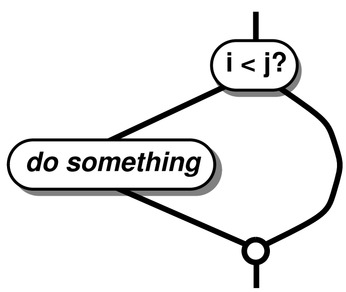
These inefficiencies stem from the control flow merging together after the **`intLT`** primitive only to be split apart again as part of type casing before the **`ifTrue:`** message. If the merge after the **`intLT`** primitive were simply postponed until after the two versions of the **`ifTrue:`** message, the inefficiencies would disappear, and the SELF compiler would generate the same code as the C compiler. For example, if after inlining the **`intLT`** primitive to reach the following graph:



the compiler had delayed the merge until after the **`ifTrue:`** message, copying all control flow graph nodes between the primitive result and the **`ifTrue:`** message to get the following graph:



then the compiler could directly inline the two **`ifTrue:`** messages without inserting any run-time type tests, since type analysis will infer that the type of the left-hand **`ifTrue:`** message is *true* and the type of the right-hand message is *false*. After inlining, the assignments to $\texttt{result}_2$ can be optimized away, since they are no longer needed at run-time. This leads directly to the following control flow graph, after eliminating assignment nodes which do not generate machine code:



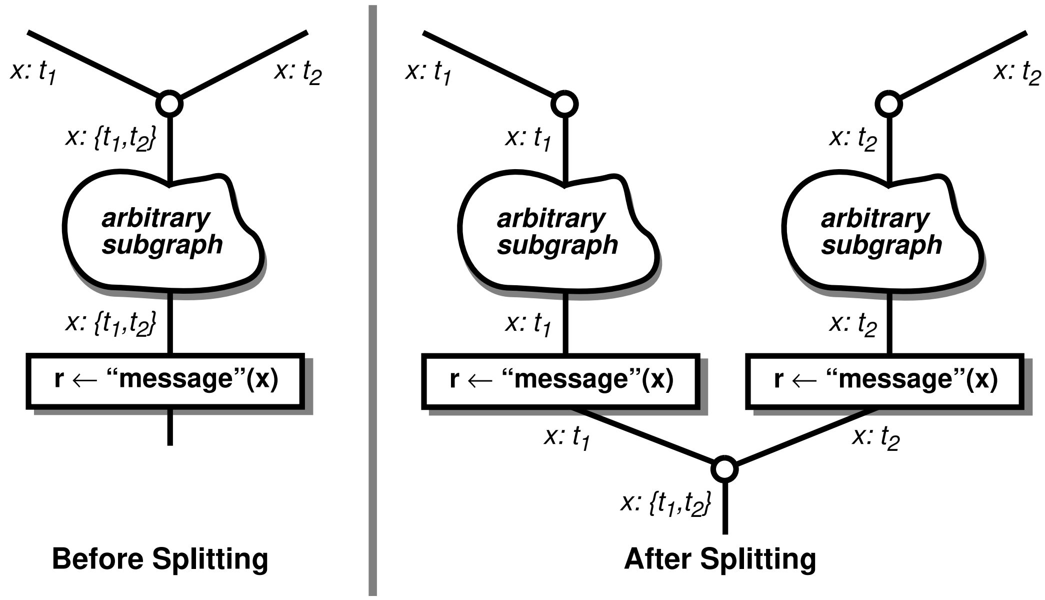
This graph is the same as that produced by the C compiler. To achieve this level of performance and be competitive with traditional languages, the SELF compiler needs some mechanism to postpone merges selectively, to avoid falling back on run-time type casing code. In other words, the SELF compiler needs some *splitting* mechanism.

We use “splitting” as a general term for techniques which lead to multiple versions being compiled of parts of the control flow graph, each version optimized for different situations such as for different type bindings. Several kinds of splitting have been implemented in the SELF compiler, each with a different trade-off between compilation speed and execution speed. The main discriminating characteristics of the various splitting strategies is when they decide to postpone a merge (and thus split nodes downstream of the postponed merge) and how they decide to stop postponing the merge. The next few sections describe these different approaches.

10.2 Reluctant Splitting

Reluctant splitting has been used in one form or another in all the implementations of the SELF compiler. In this splitting variant, the compiler initially merges control flow together but can later reverse this decision if desired, undoing the merge by copying parts of the control flow graph. This kind of splitting is called "reluctant" because the compiler splits merges only on demand; *lazy splitting* or *demand-driven splitting* would have been equally appropriate names.

For example, consider the following generalized example:



Under reluctant splitting, when reaching a potential merge point, the compiler merges control together. The compiler remembers the merge by forming union types for any names bound to different types before the merge. In this example, the **x** name is bound to the union type $\{t_1, t_2\}$. If later on some control flow node could be optimized if a name bound to a union type were instead bound to a component of the union type, the compiler will reverse its earlier decision to merge control together by duplicating all the control flow graph nodes between the premature merge and the node that demands the split. This duplication approximates the control flow graph that would have been generated had the original merge never taken place. In this example, the message send node could be optimized if the receiver **x** were bound to either component type rather than the joint union type, and so the message send node demands that the merge be postponed at least until after the message send. This demand is satisfied by duplicating all the nodes between the message send back to the original merge point. Subsequent splits may postpone the merge even farther. By copying parts of the control flow graph for each different type, the compiler has transformed what used to be a single polymorphic (and hence unoptimizable) message send into several independent monomorphic (and hence optimizable) message sends, without inserting any extra run-time overhead for type tests.

If the arbitrary subgraph copied as part of reluctant splitting is restricted to be empty (has no nodes that generate instructions), then we call the splitting algorithm *local reluctant splitting*, since the splitting takes place locally in the control flow graph. If arbitrary subgraphs are allowed, then we call the splitting algorithm *global reluctant splitting*.*

Splitting turns out to be a crucial technique for achieving good performance. As detailed in section 14.3, without any form of splitting SELF programs would run only half as fast.

* These two cases were termed *local message splitting* and *extended message splitting* in [CU90].

10.2.1 Splittable versus Unsplittable Union Types

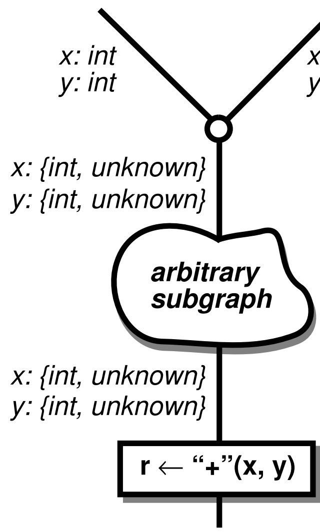
The compiler detects when reluctant splitting is possible by checking whether the type of some expression is a union type, since merges create union types of their component types and these component types may be split apart later. However, not all union types are created by merges: some are the results of primitives known to return one of a set of possible types. For example, the type of the result of a floating-point comparison primitive is the union of the **`true`** constant type and the **`false`** constant type. However, the compiler does not inline this primitive, instead invoking it by calling an external C++ function built into the SELF implementation, and so there is no merge node that can be split apart to separate the **`true`** result from the **`false`** result. Splitting is not applicable in this case, and the compiler should fall back on type casing (described in section 9.3) to optimize messages sent to the result of this primitive.

The compiler distinguishes between union types created as a result of merges in the control flow graph (and thus amenable to splitting) and those created externally (and hence not amenable to splitting). The former kind are called *splittable union types* while the latter are called *unsplittable union types*. Merge nodes create splittable union types; a floating-point comparison primitive returns an unsplittable union type. The compiler only attempts splitting for splittable union types.

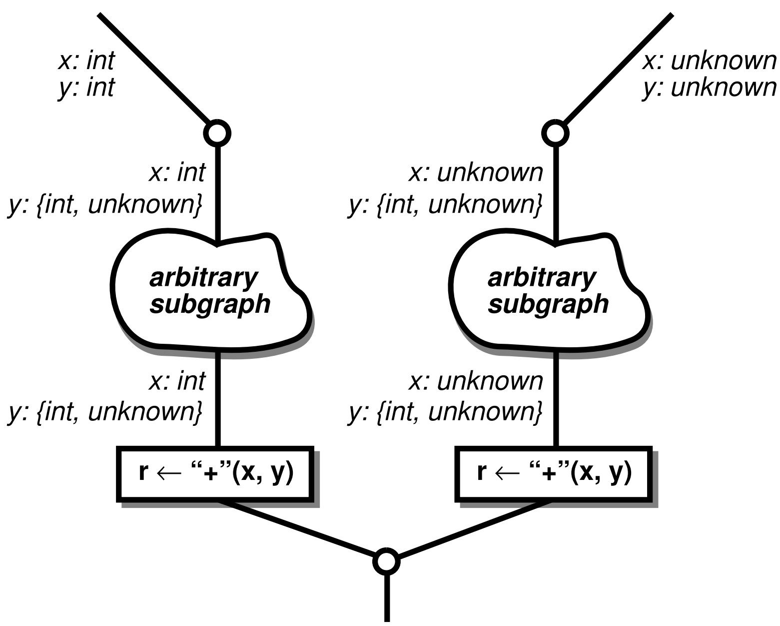
10.2.2 Splitting Multiple Union Types Simultaneously

The compiler uses splitting to transform a single piece of polymorphic code into several independent pieces of monomorphic code. Each splitting step typically operates on a single name and a single union type, such as the name and type of the receiver of some message. After the splitting operation, the compiler can update the type of the name along each of the split branches to the appropriate component type. These more precise types can then enable inlining along the split branches. Thus, an important function of splitting is to break apart union types into their component types which enables further optimizations.

This dividing of union types works fine for the single name being split upon, but does not improve the types of any other names that might also be more precise after the split. For example, consider the following control flow graph fragment:



After splitting the **+** message for the **`integer`** map type case of **`x`**, the compiler can alter its type bindings for **`x`** along the two split branches to get the following split control flow graph:

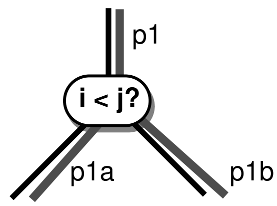


Unfortunately, using the techniques described so far the compiler would not notice that the type of **`y`** could also be narrowed along the two split branches. This would cause the compiler to insert an unnecessary integer type check for the **`y`** argument to the **`intAdd`** primitive that will eventually get inlined as part of the implementation of **+** for integers.

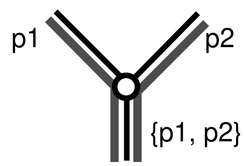
10.2.2.1 Paths

The SELF compiler solves this problem by augmenting type information with information about which possible paths through the control flow graph lead to various name/value and value/type bindings. These possible paths through the control flow graph are represented internally in the compiler by *path objects*. Path objects serve to link together pieces of type information that are guaranteed to occur together, such as components of different union types.

A path object represents a possible flow of control through the control flow graph. During type analysis the compiler keeps track of the set of path objects that represent the possible paths through the control flow graph that lead to the node being analyzed. The initial node in the control flow graph is associated with the initial path object. For straight-line code (i.e., basic blocks), the set of paths is propagated through from predecessor node to successor node unchanged. At branch nodes, for each incoming path the compiler creates a new path object for each outgoing branch, capturing the fact that two paths through the control flow graph are possible for each incoming path:

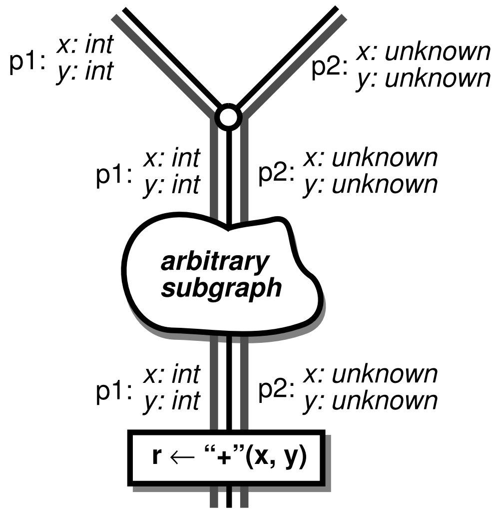


At merge nodes, the compiler forms the union of the set of paths along each incoming predecessor of the merge:

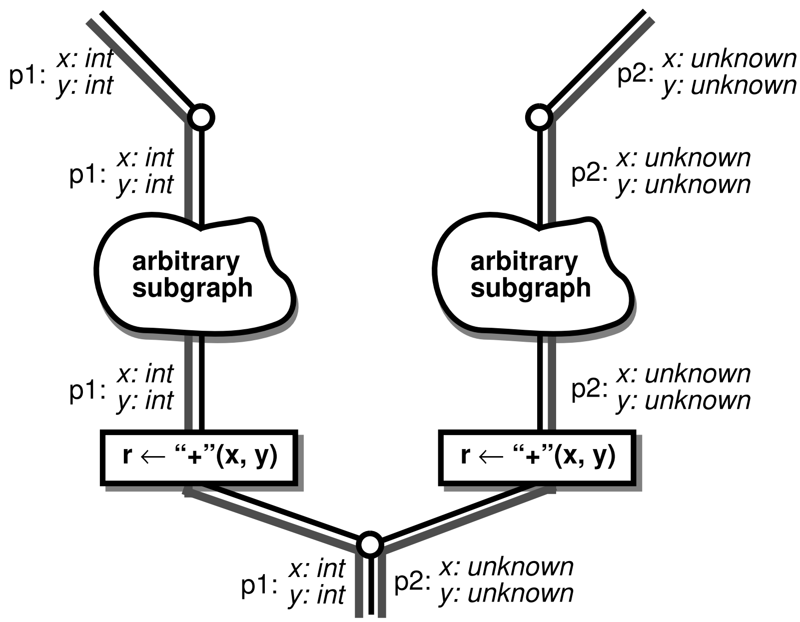


The compiler associates type information not with just a node in the control flow graph but with each path through the node. Conceptually, the compiler records name/value and value/type mappings (plus any other kinds of type information propagated such as cell/value mappings) for each path through the control flow graph. To compute the type of an expression at some point in the control flow graph, the compiler forms the union of the information about the expression associated with each path that reaches the node.

To illustrate, the following graph fragment shows how the compiler associates type information with paths rather than just nodes for the earlier example:



When the compiler reaches the **+** message, it elects to split off the integer case. To do this it must separate those paths where **x** is bound to *int*, namely p1, from the other paths, namely p2. This leads to the following graph:



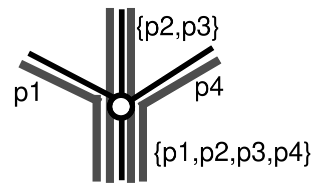
Paths elegantly solve the problem of narrowing the type of **y** appropriately after the split. When the compiler splits off one set of paths from another, in this case splitting off path p1 from path p2, all types associated with the split paths get narrowed implicitly. The type of **y** automatically is narrowed to *int* when p1 is split off from p2, since the type of **y** is associated directly with individual paths, not just with control flow graph nodes.

Splitting is couched in terms of separating one subset of paths from the remaining paths, instead of splitting some union type. The splitting subsystem of the compiler operates solely in terms of splitting paths apart and knows nothing about why the split is being performed or what affect the split should have on various kinds of type information. Other parts of the compiler are responsible for deciding when a split is in order, either to break apart a splittable union type, or to make some expression available for common sub-expression elimination, or even some combination of these. Once the compiler has decided to split, it calculates what path subset satisfies the desired criteria and invokes the splitting subsystem to actually perform the split.

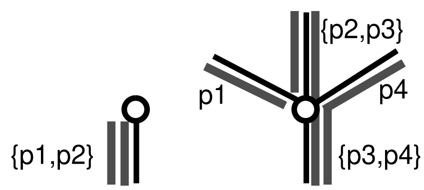
This separation of concerns simplifies the organization of the compiler by narrowing the interface between the splitting system and the rest of the compiler. It also enables certain kinds of splitting operations that otherwise would be costly or awkward. For example, sometimes the compiler needs to split off branches in which several expressions have particular types, instead of the normal case of splitting based on a single expression's type. This can occur when splitting off branches to a primitive in which all the primitive's arguments have the right types (i.e., where the primitive will not fail with a type error), or when connecting loop tails to loop heads as described later in section 11.4. The compiler can perform such splits simply by calculating which paths satisfy the right requirements and then calling the splitting subsystem with that path subset.

10.2.2.2 Splitting Merge Nodes

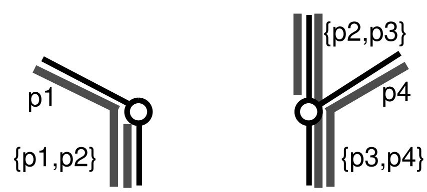
The splitting subsystem of the compiler performs the actual splitting operation by walking backwards through the control flow graph, copying each control flow graph node as it is traversed. At a merge node, the compiler examines each of the merge node's predecessors to decide how to split them, based on the paths that come in along that predecessor. For example, consider the following control flow graph fragment:



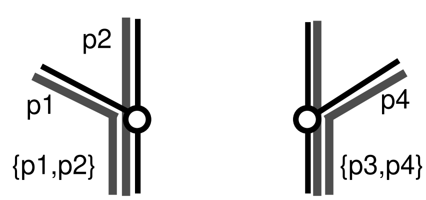
Say the compiler wants to split apart paths p1 and p2 from the other paths, and that the compiler has split nodes up to this merge node (the copied nodes are on the left):



The compiler first examines at the left-most predecessor. Since all of its paths (namely p1) are in the set of paths being split off (namely {p1, p2}), this predecessor is simply redirected to the copied merge node and not processed further:



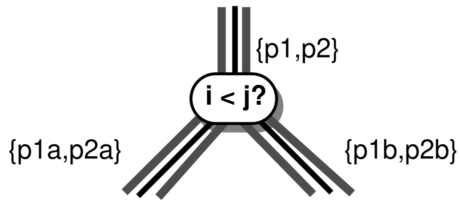
A subset of the middle predecessor's paths (i.e., p2) should be split off, while the rest (i.e., p3) should remain behind, unsplit. The compiler therefore continues to split the middle predecessor branch, eventually producing the following graph:



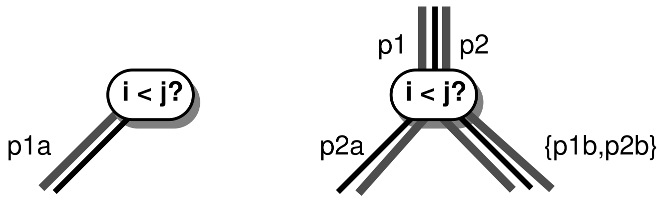
The compiler finally examines the right-most predecessor. Its paths (namely p4) are not in the split half, so this predecessor simply remains connected to the unsplit merge node.

10.2.2.3 Splitting Branch Nodes

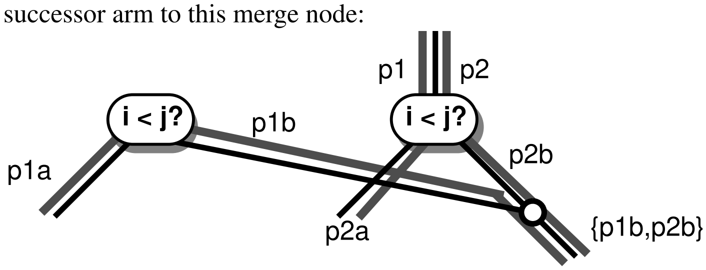
Branch nodes require special treatment to ensure that a branch node and its predecessor are split at most once, independently of whether one or both successors are split. Consider the following control flow graph fragment:



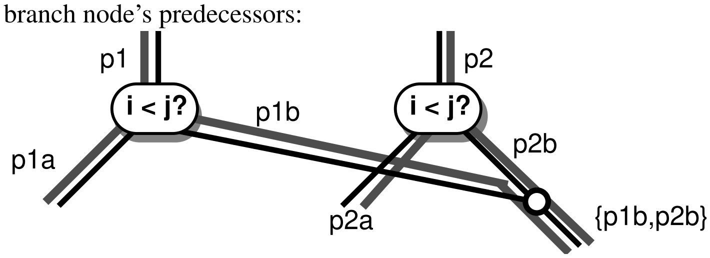
When the branch node is first split, say by splitting off the p1a path from the p2a path along the left-hand successor, the compiler first copies the branch node.



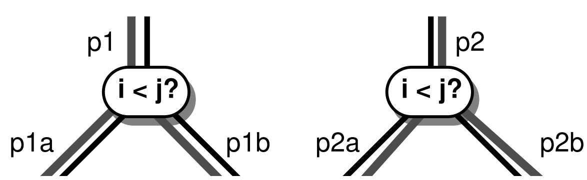
The compiler then inserts a new merge node after the other successor branch (the one that was not split) and connects the copied branch node's other successor arm to this merge node:



Finally, the compiler splits the branch node's predecessors:



If later on the branch's other successor is split, say by splitting off the p1b path from the p2b path, then normal splitting rules for merge nodes will break apart the freshly-inserted merge node, completely severing the link between the two branch nodes:



The current implementation of branch splitting actually optimizes this strategy by lazily creating and inserting the extra merge nodes. When a branch node is first split, instead of creating a merge node to join together the two unsplit successor branches, the compiler simply marks the branch node as "partially split" and links the two branch node copies together. If the other branch successor is split, then the appropriate control flow graph links are made, simulating the step of breaking apart the merge node that would have been inserted. To handle the case where the other successor to a branch node is not split, the compiler visits all "partially split" branches after the whole splitting operation is complete and then creates and inserts the necessary merge nodes. This optimization should speed splitting, especially if most branch nodes get split from both sides, but does introduce some complexity.

10.2.2.4 Implementation of Paths

A direct implementation of type information and paths as described could be quite inefficient if many paths have similar type information. Each path would have its own complete copy of the type information, with lots of duplication among paths. To avoid this potential problem, the current SELF compiler reintroduces splittable union types and associates each component of a splittable union type with the set of paths that generate that type. For example, the compiler represents the type binding where a variable **x** is bound to a type *t1* on paths p1 and p2 and a type *t2* on paths p3 and p4 as

```
x: {t1=[p1,p2], t2=[p3,p4]}
```

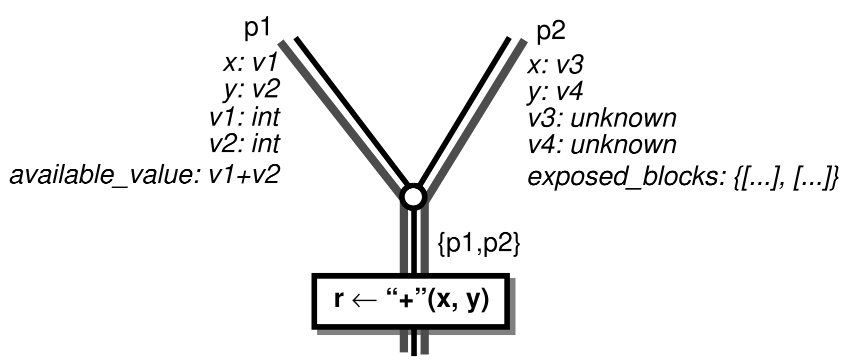
instead of

```
p1: x: t1; p2: x:t1; p3: x:t2; p4: x:t2.
```

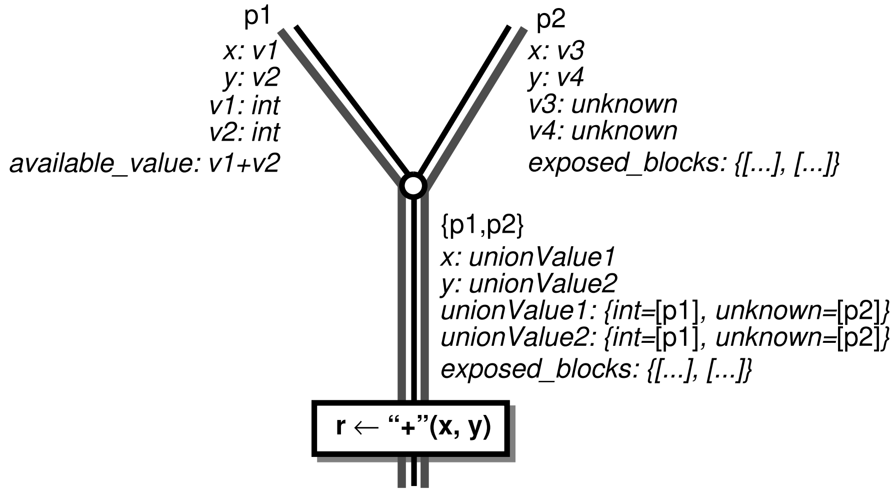
Values bound to types other than splittable union types are interpreted as being bound to the same type for all paths. Thus, information that does not vary from path to path is stored just as concisely as it was before paths were introduced.

This representation of type information is more compact than the straightforward representation using a complete copy of the type information for each path. It supports the same style of splitting including narrowing of all appropriate type bindings after a split. It also supports very efficient detection of when two paths lead to different types that might be split apart (by checking for values bound to splittable union types) and efficient calculation of which paths lead to desired types (by examining the set of paths associated with the desired component types of splittable union types).

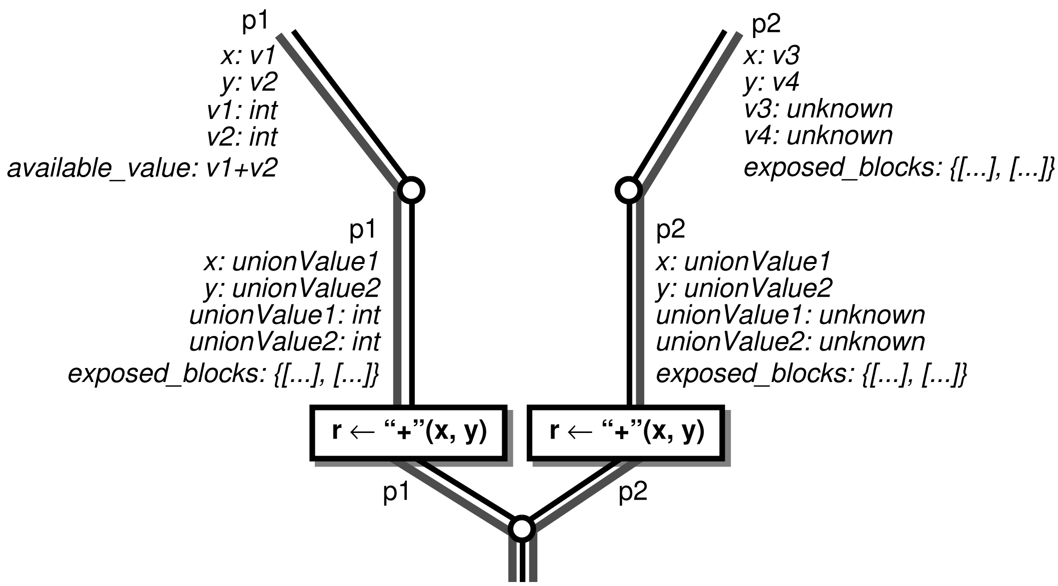
Unfortunately, this representation has some drawbacks over the straightforward representation. The chief drawback is that currently not all type information is connected with paths. Only types bound to values in the value/type mapping can depend on path information and be automatically narrowed, since only splittable union types relate to path information; all other information, such as name/value mappings, cell/value mappings, and exposed blocks lists, are assumed to apply to all paths. This lack of precision means that information other than the types associated with values can be lost if paths merge together and are later split apart. For example, in the following graph, the two paths merging together have different information:



When combining the information after the merge, the compiler creates new union values to represent the bindings for **x** and **y**. Paths are associated with the components of these splittable union types. The compiler must treat the *v1+v2* value as no longer available (since it is not available along path p2 and the compiler cannot associate available value information with particular paths) and must consider the exposed blocks from path p2 as exposed after the merge (since the compiler cannot associated exposed block information with individual paths):



When reaching the **+** message send, the compiler splits apart the type of **x** to optimize the integer receiver case, producing the following graph:



The compiler is able to narrow the type of **y** using the path information in the splittable union types of **x** and **y**. Unfortunately, the compiler cannot reclaim the lost information that *v1 + v2* is an available value along path p1 (which if the compiler could also restore the original value bindings of **x** and **y** would allow the compiler to eliminate as redundant the **x _IntAdd: y** calculation), nor can the compiler exclude the unnecessary blocks from the exposed blocks list along path p1. These differences do not lead to incorrect code: it is always legal to have fewer available values or more exposed blocks than is required. They do, however, sacrifice some information that could lead to better code.

A better implementation of type information and paths would enable the compiler to reclaim *all* type information after a split, as if no merge had ever occurred. The current SELF compiler only handles reclaiming type bindings; value bindings, available expressions, and exposed blocks lists are not reclaimed. The type binding case is by far the most important, since inlining a message is a much more important optimization in most cases than, for instance, common subexpression eliminating a single instruction. Therefore, we hope that the opportunities for optimization lost by the current imperfect implementation are relatively minor.

10.2.3 Heuristics for Reluctant Splitting

Since splitting can lead to an increase in compiled code space and compilation time, the compiler includes heuristics to determine when splitting should be performed and thus balance the expected improvement in run-time execution speed against the expected costs in increased code size and compile times. In the current SELF compiler, two factors influence the decision on whether to split part of the control flow graph: the *cost* of the split in terms of the number of copied control flow graph nodes and machine instructions, and the *weight* of the split paths in terms of their relative likelihood of execution. The compiler avoids splitting if the cost of the split is high or if the weight of the split paths is low, with stricter limits for splits that enable only less important optimizations.

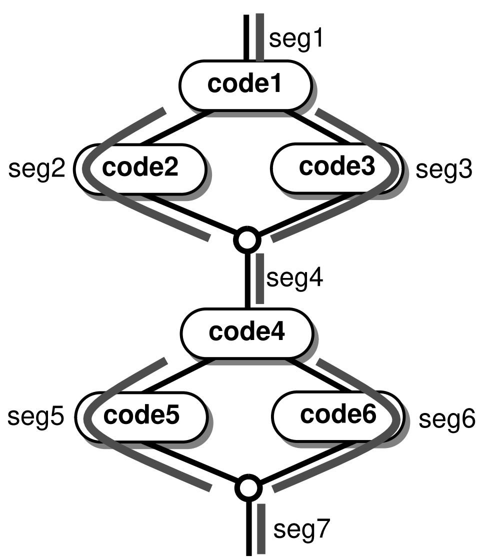
10.2.3.1 Costs

The cost of splitting one set of paths off from the remaining paths is calculated as the sum of the costs of the control flow graph nodes that would be copied as part of the split. Each node in the control flow graph has an associated cost, determined as follows:

- Many kinds of nodes have zero cost, since they do not generate machine instructions and can therefore be copied for free. For instance, name binding (assignment) nodes typically will generate no instructions, since the register allocator will arrange that the left- and right-hand-sides of the assignment end up in the same register.
- Other kinds of nodes will generate one or two machine instructions (such as arithmetic nodes and compare-and-branch nodes), and are given a cost of 1 or 2 as appropriate.
- Non-inlined message send nodes are given a cost of 5 to account for the extra space required for the send's in-line cache and for the extra instructions that are frequently needed to move the send's arguments into the locations defined by the calling convention.

It is difficult to determine efficiently which control flow graph nodes would be copied as part of a split. The compiler could determine this by simulating the split, traversing the graph in the same manner as the splitting operation would, but this would be expensive. The current SELF compiler represents paths as connected *path segments* to enable a more efficient computation of where paths overlap. Each straight-line sequence of code in the control flow graph (each basic block) is associated with a single path segment object. Path objects are represented as a list of the path segments that are traversed by the path. All paths that pass through a segment's corresponding basic block share the single path segment object.

For example, the following control flow graph sports 7 path segments which are linked together in 4 different paths (path segments listed from last to first, as path segments are "consed" onto the head of a path at each branch and merge point):



p1: (seg7 seg5 seg4 seg2 seg1)
p2: (seg7 seg6 seg4 seg2 seg1)
p3: (seg7 seg5 seg4 seg3 seg1)
p4: (seg7 seg6 seg4 seg3 seg1)

Path segments are concise abstractions of straight-line chunks of the control flow graph. Since straight-line chunks of code are split as a unit, the cost of splitting the nodes in the chunk can be computed once as the nodes are type analyzed and stored in the path segment object. Then to determine the cost of a total split, the compiler can first determine which path segments will get copied as part of a split, and then sum the costs of those path segments.

The compiler can compute which control flow graph segments will get copied as part of a split by finding those path segment objects that are shared between the set of paths being split off and the set of paths being left behind, up to the point at which the split paths and the unsplit paths diverge. In the above example, if paths p1 and p2 are to be split off from paths p3 and p4, the compiler calculates that segments seg7, seg6, seg5, and seg4 are shared between the split and unsplit paths. Segments seg3 and seg2 are not shared, and so will not be copied by the split, and even though seg1 is shared, it lies beyond the point at which the split and unsplit paths diverge, and so it will not be copied as part of the split.

This mechanism using path segments is more efficient than the straightforward approach of just simulating the splitting operation because it effectively caches the results of many of the operations that the simulation would perform, such as the costs of the nodes in each basic block. Also, the path graph is much smaller than the control flow graph, so traversing the path graph is faster than traversing the main control flow graph. Of course, such an optimization increases the complexity of the compiler.

10.2.3.2 Weights

The weight of a node represents the compiler's estimate of the likelihood of that particular node being executed. A weight is composed of two measures: a loop nesting depth and an "uncommonness" amount. The loop nesting depth component records the number of loops that have been entered but not exited since the beginning of the method. The initial loop nesting depth is zero.

The uncommonness component records how unlikely the compiler considers reaching that point in the control flow graph. For example, the compiler considers having a non-integer receiver for a message like **+**, in the absence of other information to the contrary, to be unlikely, and consequently increases the uncommonness component of the failure branch downstream of a run-time type test inserted as part of type-predicting the **+** message. (Type prediction was described in section 9.4.) The compiler also considers primitive failure to be unlikely, so if, for instance, the index argument to a **`_ByteAt:`** primitive either is not an integer or lies out of bounds, the compiler again increases the uncommonness component of the failure branch. The initial uncommonness value is zero, with higher uncommonnesses indicating less likely branches. A weight with an uncommonness value of zero is called *common case*, while weights with positive uncommonness values are called *uncommon cases*.

Weights do not attempt to record expected execution frequencies at a finer grain. For example, for branch nodes that correspond to normal comparisons in the source code, both successor branches are given equal weight, and this weight is the same as the weight of the predecessor of the branch. A more precise weighting system could mark the downstream branches each with half the weight of the branch predecessor. Unfortunately, this approach rapidly leads to trouble. Consider a series of conditional branches, corresponding to a series of **`if/elseif`** tests. By the time the leaves of this decision tree are reached, the weight of any individual leaf will be only $1/2^{\mathrm{depthOfTree}}$ of the original weight before entering the decision tree. Since the compiler uses weight information to decide when splitting, inlining, and other optimizations are worthwhile, with these exponentially-reducing weight calculation rules the compiler might decide that none of the leaves of a decision tree are likely enough to merit optimization, even though for each execution of the decision tree some leaf is always executed. Clearly this behavior of weights is undesirable.

Another problem with fined-grain weights is how to combine weights together at merge nodes. A basic principle that the computation of weights should obey is *conservation of weight*: the sum of the weights leaving some arbitrary part of a control flow graph should equal the sum of the weights entering that part of the graph; otherwise, some execution frequency is being lost or gained. This implies that if weights are halved at branches, then weights should be summed at merges. Unfortunately, this fine-grained approach to computing weights is difficult to implement in the presence of loops. A loop head node merges together the loop entrance branch and any loop tail branches created by **`_Restart`** calls.* The weight of the loop head is therefore defined in terms of itself, since the loop tail weight depends on the loop head weight, and so a recurrence equation must be solved to calculate the weight of the loop. A simple approach would just increment a loop depth counter and ignore the weight of the loop tail branch. However, this approach violates

* The **`_Restart`** looping primitive was described in section 4.1.

conservation of weight, since the weight of the path through the loop that eventually connects back up to the loop head is unaccounted for and lost when summing the weights of the branches that exit the loop.

Since fine-grained weight propagation rules are difficult to support without weight loss, and since their extra precision is frequently unneeded and sometimes even counter-productive, the current SELF compiler uses coarser-grained weights that measure only loop depth and uncommonness. Successors to branch nodes are given equal weight with the predecessor to a branch node, and merge nodes take the maximum weight of their predecessors rather than the sum of their weights. These rules satisfy conservation of weight, since the weight of all loop exit branches is the same as the weight of the loop entrance branch (ignoring the effect of failed type predictions and primitives along loop exit branches). Also, since all leaves of a decision tree are the same weight as the entrance to the decision tree, all leaves will be optimized as if they were executed whenever the decision tree is executed, which is usually the desired effect.

Weights are associated with both paths and with nodes. The weight of individual paths is maintained as type analysis progresses and adjusted whenever a loop is entered, a loop is exited, or an uncommon branch is taken. Weights of paths remain unchanged when paths flow together at merge nodes or split in two at branch nodes. The weight of a node is the maximum of the weights of the paths reaching that node.

Since weights are associated with paths in addition to nodes, the splitting operation can easily compute the weight of a node that might be split off as the maximum of the weights of the split paths. After splitting, a new weight can be calculated in the same way for the branch left behind.

10.2.3.3 Cost and Weight Thresholds

Costs and weights are used to control which paths, if any, should be split when the compiler detects an opportunity for splitting. The compiler includes three threshold values, **`MaxSplitCost`**, **`MaxLowSplitCost`**, and **`MaxSplitUncommonAmount`**. **`MaxSplitCost`** and **`MaxLowSplitCost`** specify the largest cost of a splitting operation, above which the compiler will decide not to perform the split. **`MaxSplitUncommonAmount`** defines a threshold value of uncommonness that selects between **`MaxSplitCost`** and **`MaxLowSplitCost`**:

```
if splitWeight.uncommonAmount <= MaxSplitUncommonAmount then
  -- this is a relatively common path; use the more generous threshold
  if splitCost <= MaxSplitCost then
    -- this split is relatively inexpensive; go ahead!
    SPLIT
  endif
else
  -- this is an uncommon path; use the more stingy threshold
  if splitCost <= MaxLowSplitCost then
    -- this split is relatively inexpensive; go ahead!
    SPLIT
  endif
endif
```

Currently, for global reluctant splitting **`MaxSplitCost`** is set to 50, **`MaxLowSplitCost`** is 0, and **`MaxSplitUncommonAmount`** is also 0. This means that for common-case nodes (nodes with a zero uncommonness weight), the compiler is willing to duplicate up to 50 instructions as part of a split (this limit is rarely reached in practice). For uncommon nodes, however, the compiler is not willing to duplicate any instructions; only splits that do not increase the size of the compiled code are allowed along uncommon paths. Local reluctant splitting mode is enabled simply by changing **`MaxSplitCost`** to 0, thus preventing the compiler from ever duplicating instructions as part of a split.

10.2.4 Future Work

The current SELF compiler's heuristics for deciding when to perform reluctant splitting are fairly crude. They only weigh the expected cost of a split in terms of additional machine instructions generated and the likelihood of the split branch being executed and so benefitting from the split. Many other pieces of information could be used to improve the results of splitting. For example, the current heuristics take into account the expected increase in compilation time from a split only indirectly, through the dependence on the expected number of duplicated machine instructions. Also, these heuristics do not include any measures of the expected improvement in run-time performance caused by the split, other than the weight of the split paths. A better set of heuristics would differentiate various splitting opportunities with some characterization of the expected pay-off, for example by giving a high pay-off value for splitting that enables inlining of a message and a relatively low pay-off value for splitting that allows only constant folding or common subexpression elimination to be performed. Profile information gathered from previous executions of the program might also be employed to direct the compiler's attention to the most important parts of the program.

A more serious problem with the current heuristics is that they only examine local information. For example, when deciding whether or not to split a message, the compiler only looks at the costs and benefits for that single message. A better approach would take into account any future messages to or operations on the receiver. If many operations are going to be performed on some expression, then the compiler should be more willing to split the first operation, since the cost of the split can be amortized over all the subsequent uses of the split. Similarly, if the compiler has already duplicated many nodes as part of earlier splits, it should become more reluctant to split future messages, to avoid spending too much compile time and compiled code space on splitting. The current localized view influences other aspects of the compiler as well, such as deciding whether or not to type-predict or type-case a message. A more global perspective throughout the compiler could lead to significantly better trade-offs between compiled code space, compilation time, and run-time performance.

10.2.5 Related Work

Reluctant splitting is similar to redirecting predecessors in the TS compiler for Typed Smalltalk (described in section 3.1.3). Both approaches can duplicate a node after a merge, postponing the merge until after the duplicated node, to take advantage of extra information available prior to the merge. There are several differences, however. Reluctant splitting is performed using type information as part of type analysis, while redirecting predecessors is performed later in the compilation process using lower-level conditional branch information. By being performed as part of type analysis and inlining, splitting considerations can influence how the control flow graph gets constructed, and information other than conditional expressions can be split upon. Also, global reluctant splitting in the SELF compiler can split arbitrary amounts of code, while redirecting predecessors only splits nodes immediately after a merge.

Reluctant splitting has many similarities to Wegman's node distinction (described in section 3.4.5). Both approaches copy control flow graph nodes when they lead to different properties that some later code wants to optimize independently. However, node distinction is primarily a theoretical framework for explaining a variety of code motion and other optimizations, both traditional ones and novel ones, but not a practical mechanism for type-based splitting. The main disadvantage of node distinction as currently formulated is that the distinguishing criteria along which to duplicate nodes must be known in advance, before any duplication takes place. In other words, the compiler must know if it will be splitting before it merges any information together. In practice, however, the compiler does not know which merges will be split apart later and so should be postponed, and which merges should merge normally. This information only becomes available to the compiler when it reaches a message send node that wants the merge to be split apart, well after it has already made the decision of whether or not to split the merge. Reluctant splitting has the practical advantage that it operates on demand, first merging branches together and later splitting only if the information needs to be split apart and it is cost-effective to do so. One potential avenue for future work would combine the two approaches to produce a more theoretical framework for demand-driven splitting algorithms.

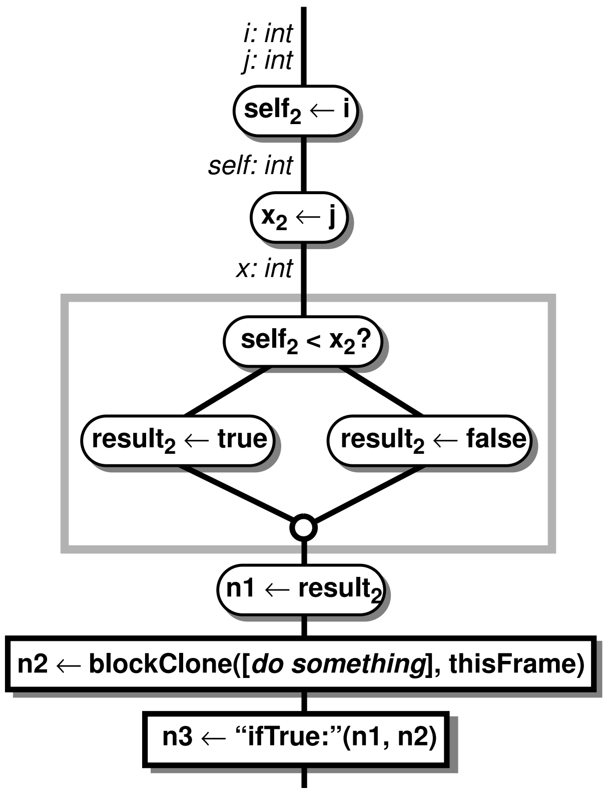
10.3 Eager Splitting

The SELF compiler supports an alternate splitting strategy that is almost diametrically opposed in philosophy to reluctant splitting. Where reluctant splitting initially merges branches together until they demand to be separated, *eager splitting* initially separates branches that would merge together in the source code. Each possible path through the control flow graph leads to its own sequence of control flow graph nodes; the control flow graph becomes a tree with no merge points.

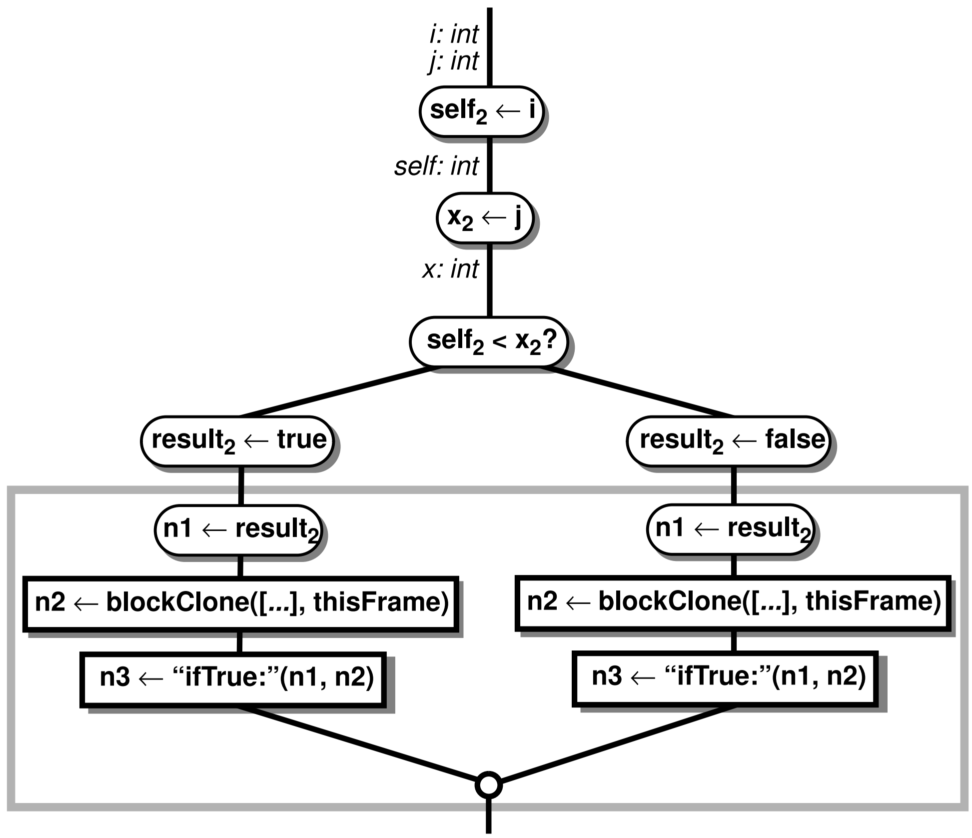
For example, when compiling the

```
i < j ifTrue: [ "do something" ]
```

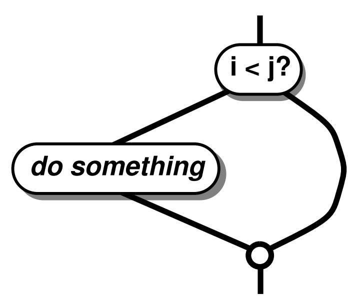
example, the compiler reaches the following control flow graph after inlining the **`intLT`** primitive:



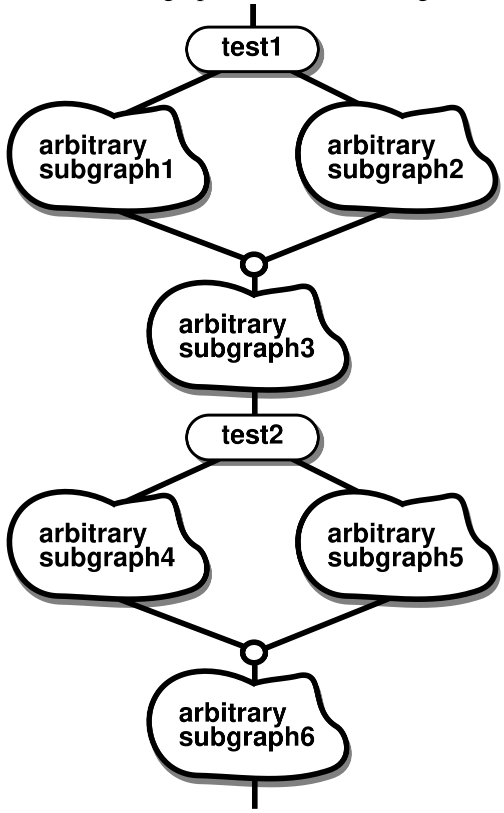
When the compiler reaches the merge node, under eager splitting the compiler does not merge control together but instead pursues each branch independently by duplicating the rest of the control flow graph for each predecessor of the merge:



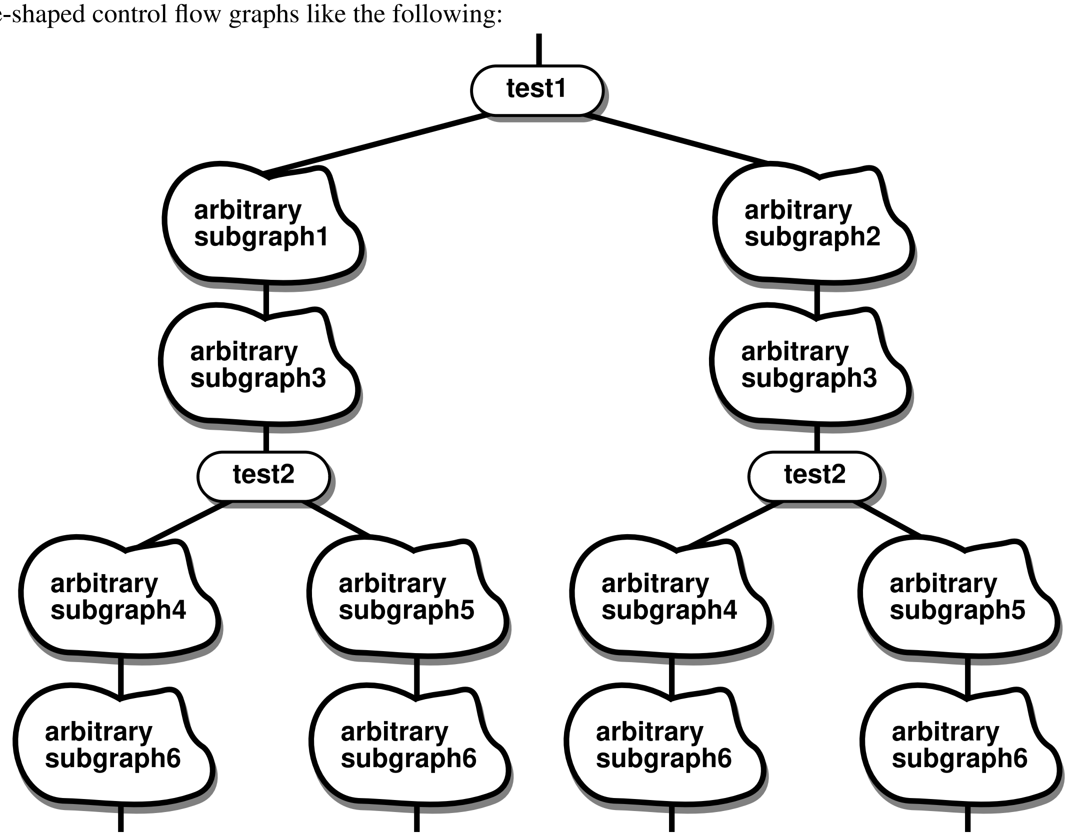
After processing these two branches independently, the compiler reaches the following final control flow graph:



In general, eager splitting transforms control flow graphs like the following:



into tree-shaped control flow graphs like the following:



Eager splitting has a number of advantages. Since there is no backtracking in the compilation and type analysis process (the compiler never changes its mind about whether two paths should be split apart), the compiler can generate the best possible code with a relatively simple type analysis system. No path data structures are required, since the control flow graph itself becomes isomorphic to the path data structure. Since the graph never merges, paths and splittable union types are not needed. Consequently, no inaccuracies or approximations induced by the representation of path-based type information can occur.

Also, the parts of the compiler that make use of the type information, such as the message inliner and the primitive operation inliner, need not constantly concern themselves with path-dependent type information and whether or not to split to get more specific information. This makes the rest of the compiler much simpler. In the reluctant splitting world, the complexity of testing for splittable union types and electing to perform a split is spread out through the parts of the compiler that exploit the type information. In eager splitting, the rest of the compiler is isolated from splitting issues, since a centralized system is making the splitting decisions (namely, to split always).

However, eager splitting has a number of serious drawbacks. The most obvious is the potential exponential code explosion as each path of control flow in the source code in turned into a separate physical branch through the control flow graph. As a first cut at reducing this space problem, the SELF compiler only performs eager splitting among common-case control flow graph branches; uncommon branches use an alternate splitting technique such as reluctant splitting. This heuristic attempts to balance better the benefits and costs of splitting by paying a high code space price only where it is likely to pay off (the common-case paths), and paying a much-reduced price where it is less likely to pay off (the uncommon case paths).

A problem still remains for control flow graphs containing loops: if the graph can never merge, then loop tails cannot merge with their loop heads, and programs containing loops suddenly have infinite tree-shaped expanded control flow graphs after eager splitting. The solution in the current SELF compiler's implementation of eager splitting is to treat loop heads differently from other merge nodes and allow loop tails to be connected to loop heads to form merges.

Even with these two modifications to the pure eager splitting model, however, the compiler still is likely to duplicate far too much code. For example, if two common-case branches merge together in the original unsplit graph, eager splitting will compile two completely independent copies of the rest of the control flow graph for those two branches. If those two branches have the same type information before the merge, then the two copies of the rest of the control flow graph will be *completely identical*! Performance measurements reported in section B.4.4 of Appendix B show that the SELF compiler takes about an order of magnitude more compiled code space and compile time with this version of eager splitting than it does using reluctant splitting.

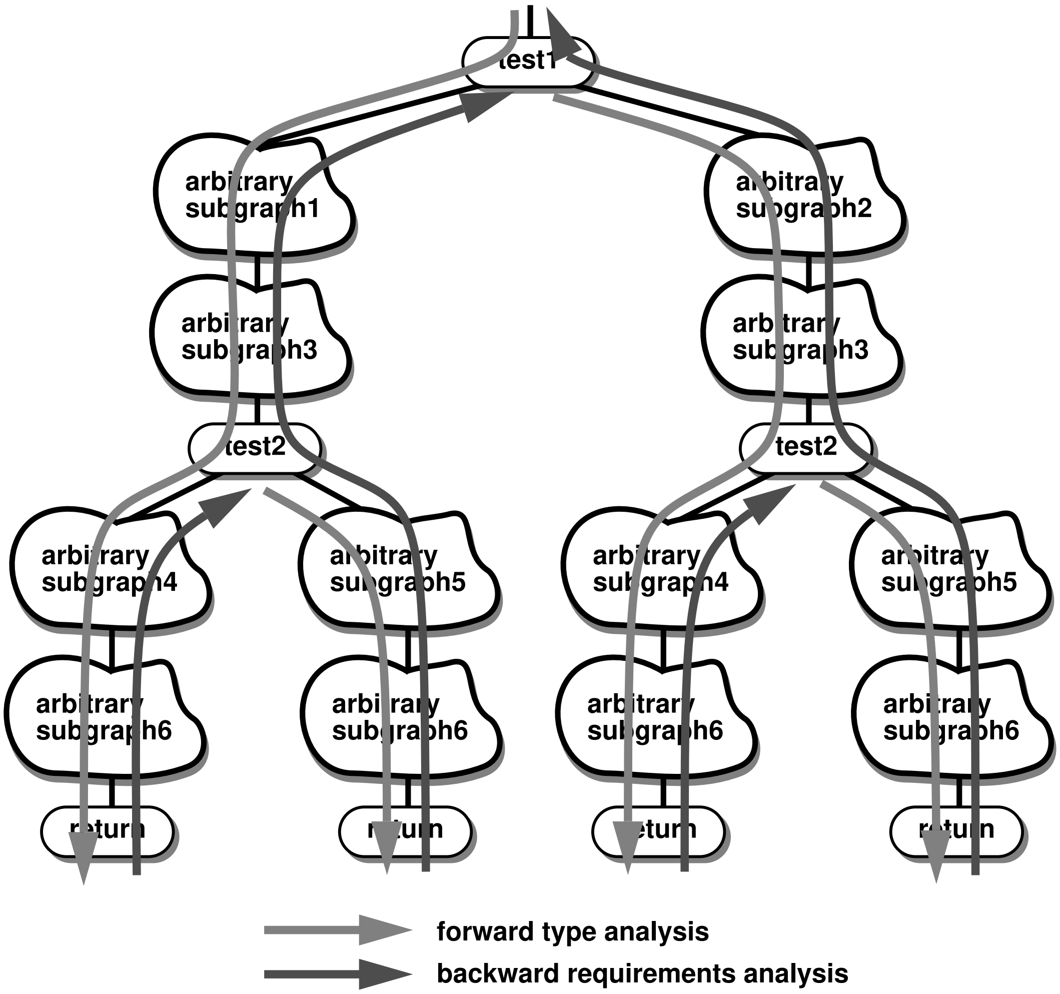
This observation suggests an approach for reducing the code explosion further: *tail merging*. In general, if two terminal branches in the control flow graph are the same, then they can be combined to save code by introducing a merge to connect the two branches together. Of course, to save compilation time as well the compiler would like to be able to detect when two terminal branches are going to be identical without actually compiling both branches and then comparing them. This approach is called tail merging after the similar traditional optimization that combines the ends of the two arms of a conditional statement if they are the same.

An easy way of detecting that two branches will generate the same code is to check whether the type information is the same at the beginning of the two branches. If they are the same, then the two branches must generate the same code and can be merged together. This prevents unnecessary duplication for "diamonds" in the control flow graph (i.e., conditional branches following by corresponding merges) where neither of the two arms of the conditional ("sides of the diamond") affect the type information entering the diamond, and so the type information exiting the diamond at the merge is the same along both branches.

Unfortunately, this simple approach is overly conservative and consequently does not save as much compiled code space or compilation time as would be desired: compilation speed and compiled code density improve by a factor of two over eager splitting without tail merging, according to measurements reported in section B.4.4 of Appendix B, but still fall well short of the performance of reluctant splitting. This form of tail merging misses cases in which two branches start out with different type information but still end up producing identical control flow graphs because the rest of the control flow graph does not depend on all of the available type information.

To enable more merging in the control flow graph than is enabled by the simple form of tail merging, the SELF compiler includes a technique we call *reverse requirements analysis* that identifies the subset of the available type information that actually is used when compiling a terminal branch of the control flow graph. The compiler allows tail merging whenever two branches have the same *required* type information, ignoring any unused type information.

Integrating forward type analysis, splitting, and tail merging decisions with backwards requirements analysis is somewhat tricky. The compiler type-analyzes the control flow graph in depth-first manner. Once a tip of the control flow graph is reached (i.e., some sort of return node in the graph with no successors), the compiler scans backwards through the graph to a branch point with a successor that has not been forward-analyzed yet.



As part of this backwards traversal, the compiler accumulates the assumptions (the requirements) that generated nodes make on the type information. In this way, the compiler determines the subset of type information that was used during the forwards compilation of a branch of the control flow graph. Later, when determining whether a to-be-generated branch can be merged with a previously-compiled branch, the compiler checks to see if the type information available at the to-be-generated branch is compatible with that required by the previously-compiled branch, as computed by reverse requirements analysis. If compatible, then the compiler merges the two branches together. Otherwise, the compiler continues generating the to-be-compiled branch separately, checking for potential merges at later points in the compilation process.

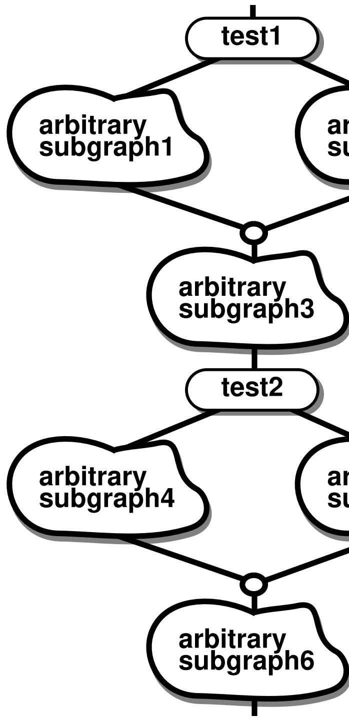
Loops complicate reverse requirements analysis much as they complicate forward data flow analysis in traditional compilers. The requirements of a loop head are computed from the requirements imposed by the loop body (and all branches downstream of the loop body), which in turn depend on the requirements of the loop head, since the requirements imposed by the loop tail depends on the requirements imposed by the loop head. Tail merging within loop bodies is even more complicated, since merging depends on requirements analysis that can only be performed after the loop has been completely constructed. The current implementation of requirements analysis side-steps these problems by assuming conservatively that the body of the loop depends on *all* the type information available at the loop head. This eliminates the problem of iteratively computing requirements but sacrifices some opportunities for tail merging. We do not know how important this sacrifice is, however.

Unfortunately, even with tail merging based on requirements analysis, eager splitting still takes too much compile time: according to the results reported in section B.4.4 of Appendix B, this most sophisticated form of eager splitting consumes twice as much compile time and compiled code space as local reluctant splitting. Ultimately, eager splitting may be deemed unusable because it is inherently inflexible relative to other strategies such as reluctant splitting. With reluctant splitting, the compiler writer can trade-off compilation speed and execution speed by varying the cost and weight splitting threshold values. With eager splitting, however, it is very difficult to "reign in" the compiler to generate less optimized code and so speed compilation. Once the eager-splitting compiler has assumed a certain piece of type information when generating code for a control flow graph node, this decision cannot be easily changed if the cost of such a decision later is deemed too great.

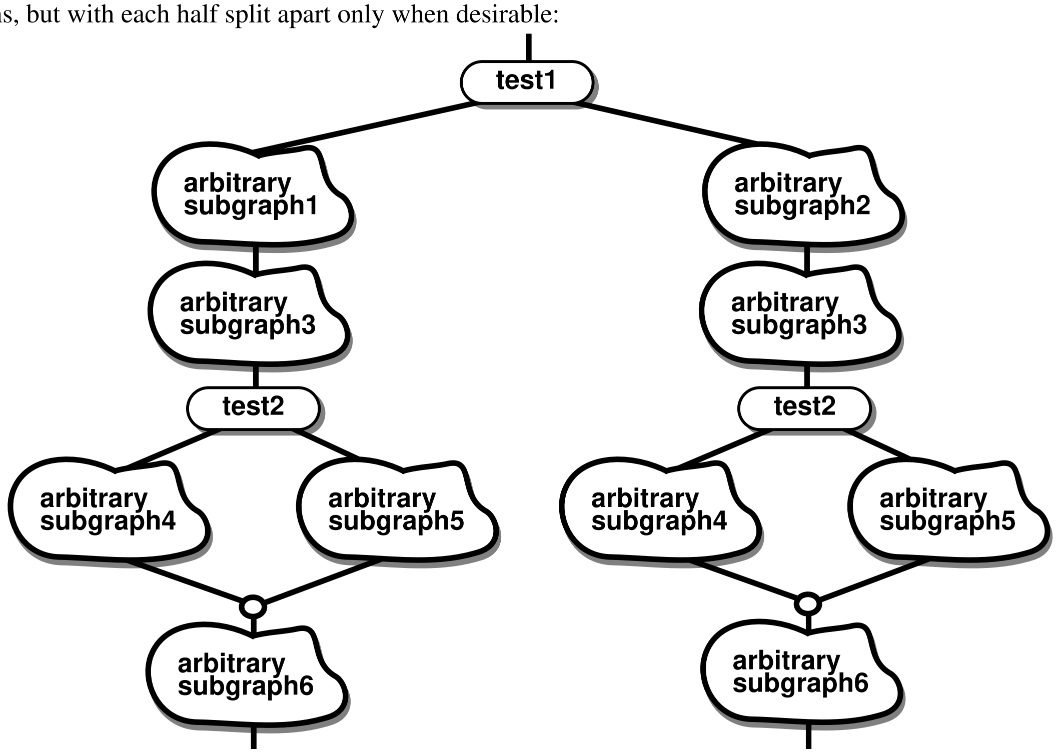
Consider an example where the compiler pursues a branch of the control flow graph, and in the last node assumes that the value of some variable is the constant **`0`**. Then a second branch is generated, and it is identical to the first branch except that the last node assumes that the value of the variable is the constant **`1`**. The cost of depending on the value of this constant may be very high, if the two branches are long and would otherwise be identical. But the compiler would have a very hard time detecting that the cost of this particular use of this particular constant type will be high without actually generating the two branches to completion. Reluctant splitting does not have this problem, since the compiler can decide at the point it is compiling the last node whether or not to split earlier branches apart. Eager splitting errs on the side of splitting branches apart, while reluctant splitting errs on the side of keeping branches merged together. The behavior of reluctant splitting probably is preferable to that of eager splitting.

10.4 Divided Splitting

Divided splitting is a practical compromise between reluctant splitting and eager splitting. Its description is simple: the compiler eagerly separates the common-case paths from the uncommon-case paths, but only reluctantly separates individual common-case paths from each other. In general, divided splitting transforms control flow graphs like the following:



into control flow graphs like the following, with the common-case paths completely separated from the uncommon-case paths, but with each half split apart only when desirable:



Divided splitting should at most double the size of the control flow graph over straight reluctant splitting instead of exponentially growing the size of the control flow graph as is characteristic of pure eager splitting.

Divided splitting is an engineering compromise motivated by practical concerns and empirical observations of programs. In most cases, the compiler will want to split common-case paths apart from uncommon-case paths. The common-case paths usually have many more sends inlined away, leading to more precise type information, more values available for common subexpression elimination, and fewer exposed blocks. Divided splitting therefore always splits apart the common-case and the uncommon-case paths. Since splitting apart various common-case paths one from another is not nearly as cut-and-dried, the more conservative reluctant splitting algorithm is used in those cases.

Divided splitting improves over the current path-based reluctant splitting algorithm by effectively placing a *firewall* between the type information along common-case paths and the type information along uncommon paths; the two sets of type information never mix across this firewall. This prevents the uncommon-case type information from diluting the common-case type information. If reluctant splitting were perfect, in that there was no loss or degradation of type information if a decision to merge branches together were later changed and no opportunities for optimizations were missed in the process, then divided splitting would not be very interesting. In reality, however, reluctant splitting is *not* perfect, especially as currently implemented with path objects used only for the value/type bindings. If common-case and uncommon-case paths continually merged together only to be split apart later, as is typically the case with pure reluctant splitting, then any loss of type information caused by imprecision in the implementation of reluctant splitting will take its toll over and over again. This typically would mean that the type information would not contain any available values or heap cell information, since the uncommon-case type information would not normally have them and the current reluctant splitting implementation does not parameterize this information by path. In contrast, with divided splitting, the common-case type information is never mixed with the uncommon-case information, and so uncommon-case paths do not cause type information along common-case paths to be lost. Since in many cases there is only one common case path, divided reluctant splitting should approach the code quality of pure eager splitting with only the compile time costs of pure reluctant splitting. According to results reported in section B.4.5 of Appendix B, divided splitting speeds SELF programs by up to 20% over the base local reluctant splitting algorithm, with no additional compile time or compiled code space costs. Divided splitting allows the SELF implementation to use an imperfect, faster implementation of reluctant splitting without sacrificing too much code quality.

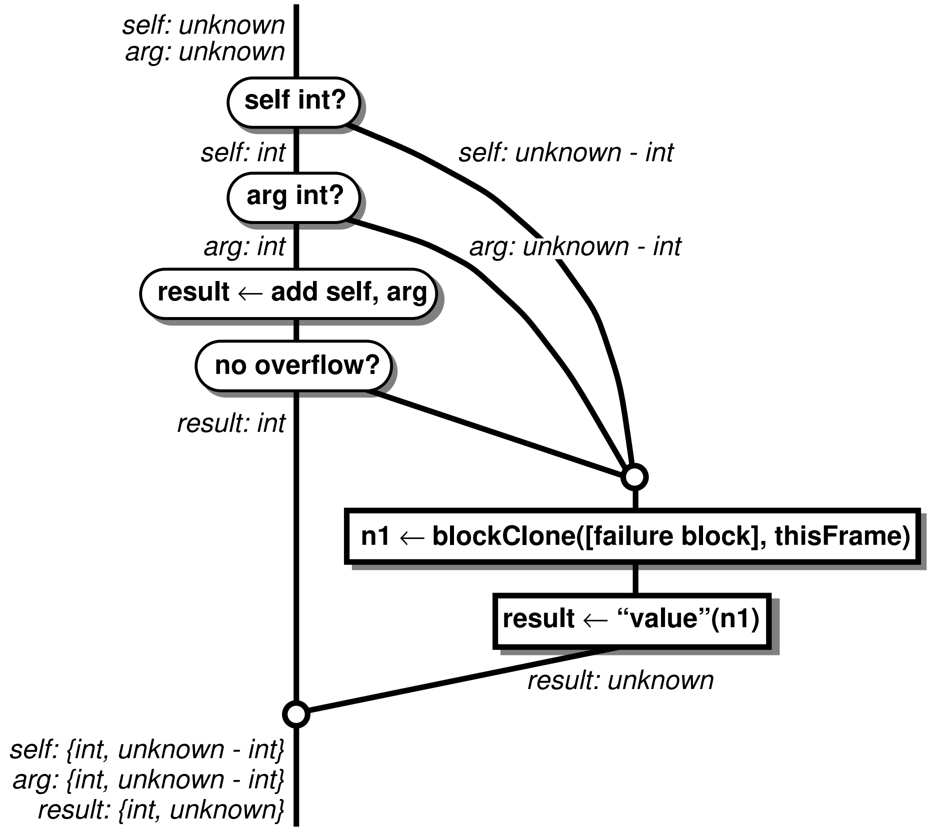
10.5 Lazy Compilation of Uncommon Branches

Many of the techniques in the SELF compiler distinguish between the common-case parts of the control flow graph and the uncommon-case parts. Some techniques such as divided splitting are designed specifically to separate the common-case paths from the uncommon-case paths. The burden of compiling the uncommon-case paths is fairly high: these parts of the control flow graph can be quite large, handling all the unusual events that might happen. Furthermore, uncommon branches are frequent in the compiled code. For example, most primitive operations such as arithmetic can fail in one way or another, leading to an uncommon-case branch that needs to be compiled. To add insult to injury, uncommon cases are expected to be uncommon, occurring rarely; most possible uncommon events never occur in practice. Thus, much of the compiler's effort is wasted.

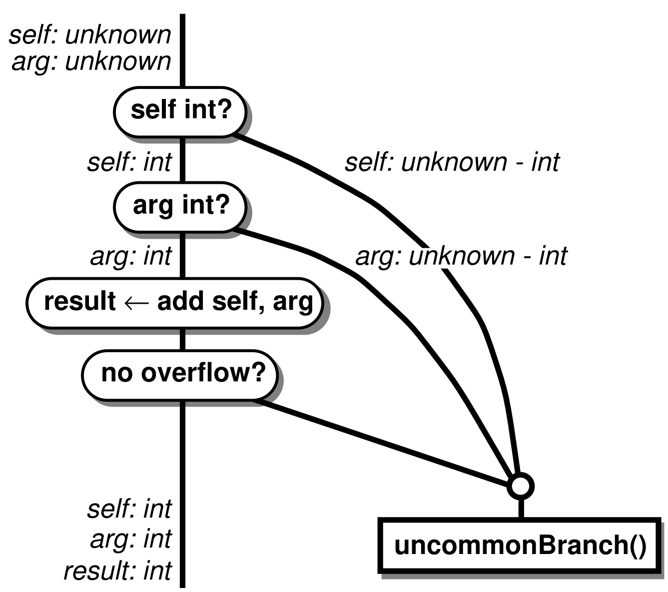
The SELF compiler takes advantage of this skewed distribution of execution frequency for different parts of the control flow graph by only compiling the uncommon parts of methods when the uncommon events actually occur; we call this technique *lazy compilation of uncommon branches*.* When the compiler is compiling a method for the first time, it only pursues the common-case paths of the control flow graph. At the entrances to all uncommon-case paths (for instance after a failed type prediction conditional branch or after a branch-on-overflow conditional branch) the

* Lazy compilation of uncommon branches was first suggested to us in 1989 by John Maloney, then a graduate student at the University of Washington.

compiler generates code to call a routine in the run-time system. Thus for a simple integer addition primitive, instead of generating the following graph:



the compiler will generate this graph:



Lazy compilation has a number of advantages. Foremost, it dramatically reduces both compile time and compiled code space by not compiling uncommon branches unless actually used in practice. The uncommon branches are typically much larger than the normal common-case code since the uncommon branch code has far more run-time type tests and space-consuming message sends than the common case branches. These expectations are borne out in practice: measurements reported in section 14.3 show that lazy compilation of uncommon branches speeds compilation by between a factor of 2 and a factor of 5; compiled code space costs fall by the same amount.

Lazy compilation has several other more subtle advantages. For one, lazy compilation automatically provides the advantages of divided splitting. Uncommon branches effectively are split off eagerly from the common-case paths, just as in divided splitting, by not compiling them at all. By providing the advantages of divided splitting as a fringe benefit, lazy compilation presents the rest of the type analysis and splitting system with a much simpler control flow graph, frequently one with only *one* remaining path. This allows the compiler to rely on a simpler, less accurate type analysis and splitting implementation without sacrificing much in performance and without actually needing to implement divided splitting.

Another subtle advantage of lazy compilation is that it eases the problem of good register allocation. As described in section 12.1, the SELF compiler performs global register allocation that allocates each name to a single location for its entire lifetime. Names that must survive across calls cannot be allocated to certain locations, such as caller-save registers. Since uncommon-case branches contain far more calls than common-case branches, eliminating the uncommon-case branches eliminates many register allocation restrictions and allows the SELF compiler's simple allocation strategy to produce a reasonable allocation. Together with improved type analysis, lazy compilation improves the execution speed of SELF programs by up to 30%, according to the results in section 14.3.

If the **`uncommonBranch`** procedure is called, the compiler starts up and compiles an additional chunk of code, called an *uncommon branch extension method*, to handle the control flow graph from the point of failure that led to the uncommon branch to the end of the original method. In the current implementation, an uncommon branch extension method takes over the original common-case method's run-time stack frame and returns to the original common-case method's caller, thus completing the original method's charter.* Since the uncommon branch extension method does *not* return to the middle of the common-case version, the common-case version is freed from handling uncommon cases merging back into the main common-case flow of control. This allows the compiler to generate excellent code for the common cases, at the cost of somewhat larger compiled code space usage. Analogously to in-line caching (described in section 8.5.1), the compiler backpatches the call to the **`uncommonBranch`** routine, replacing it with a call to the newly-compiled uncommon branch extension code. Since the compiler's beliefs about what is uncommon have already proved incorrect, the compiler does not distinguish common from uncommon branches in an uncommon branch extension. The compiler applies a conservative splitting strategy (currently local reluctant splitting) when compiling an uncommon branch extension, and the compiler does not perform lazy compilation within an uncommon branch extension method. This strategy limits the amount of compile time and compiled code space spent on uncommon branches.

The current implementation for compiling uncommon branches treats the extensions as new methods that are in some ways subroutines called by the common-case version of the method. An alternative strategy would recompile the original method when an uncommon branch was taken, this time with all uncommon branches compiled in-line (i.e., without any stub routines), replace the old common-case-only code with the more general all-cases code, modify the existing stack frame accordingly, and then restart the new method at the point of the entrance to the uncommon branch. The current strategy is significantly simpler to implement than this alternative strategy but may take up more compiled code space and compile time in cases where several different uncommon-branch entrances are taken on different invocations of the original method. With the current strategy each separate uncommon branch entry point will lead to a separate uncommon-branch extension method, while with the alternative strategy the original common-case-only method will be generalized to handle all future uncommon branches in one step. Fortunately, we have not observed many multiple uncommon branch extensions for the same common-case method, and so we are content to remain with the simpler strategy.

* The first operations performed by the uncommon branch extension adjust the stack frame size and execute any register moves required to shift from the common-case method's register allocation to the uncommon extension's register allocation.

10.6 Summary

The SELF compiler uses splitting to transform a single polymorphic piece of code into multiple monomorphic versions that can be further optimized independently, thus trading away some compiled code space for significantly improved execution speed. Reluctant splitting records enough information at merge points so that the merge can be reversed later if the more specific type information available before the merge would be helpful; the merge is simply postponed until after the point which desires the more specific information. Path objects enable the compiler to narrow multiple union types after a split, thus keeping type information fairly accurate in the face of repeated merging and splitting. The compiler includes heuristics based on the estimated space cost of the split and the estimated execution frequency of the split nodes to decide when the benefits of a potential split outweigh the costs. The demand-driven nature of reluctant splitting works well in practice, balancing compilation time against the quality of the generated code by exploiting most splitting opportunities with only a single forward pass over the control flow graph.

Eager splitting is an alternative splitting strategy that initially always splits merges apart, assuming that most such merges will be split apart anyway. Eager splitting offers a simpler, possibly faster implementation of splitting with no backtracking and no loss of type information from merges that are later split apart. Unfortunately, pure eager splitting leads to an exponential increase in the size of the control flow graph and hence in compilation time. To avoid unnecessary code duplication, eager splitting can be augmented with tail merging, a technique that merges two branches together if the second branch would generate the same control flow graph as was generated for the first branch. The compiler implements two forms of tail merging, one based on forward-computed available type information and the other based on more precise reverse-computed required type information. These tail merging extensions reduce compilation time for eager splitting somewhat, but in the end compilation time is still too long for eager splitting as currently implemented to be a practical splitting alternative.

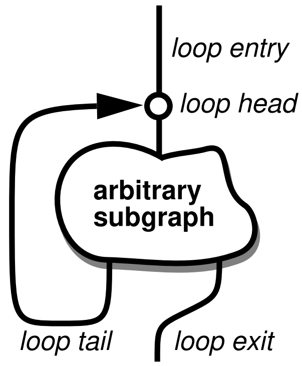
Divided splitting is a hybrid approach that strives to provide the precision of eager splitting with the costs of reluctant splitting. With divided splitting, the compiler eagerly splits apart common-case paths from uncommon-case paths, but uses reluctant splitting to split apart common-case paths from one another. Divided splitting overcomes some of the weaknesses in the current implementation of reluctant splitting by placing a firewall between the relatively precise type information available for the common-case paths and the relatively imprecise type information for the uncommon-case paths. The compilation speed of divided splitting is roughly the same as the compilation speed of pure reluctant splitting, with significantly better code quality.

The SELF compiler saves a great deal of compilation time and compiled code space and even improves execution performance by compiling uncommon branches lazily. When a method is first compiled, only the common-case paths are compiled; stubs are generated for all branches into uncommon-case paths that simply call a routine in the run-time system. Only if one of these uncommon branches is taken does the compiler actually generate code for the uncommon branches. This division automatically provides the effect of divided splitting with no additional effort over that for lazy compilation.

Lazy compilation enables the SELF implementation to “have its cake and eat it, too.” Since the common-case paths typically correspond to those paths that are the only ones supported by the semantics of traditional languages such as C, the compile-time and run-time speed of SELF is close to that for traditional languages (both systems end up generating much the same code), but SELF also supports the extra power of, for example, generic arithmetic; the programmer pays for these advanced features only when they are used.

Chapter 11 Type Analysis and Splitting of Loops

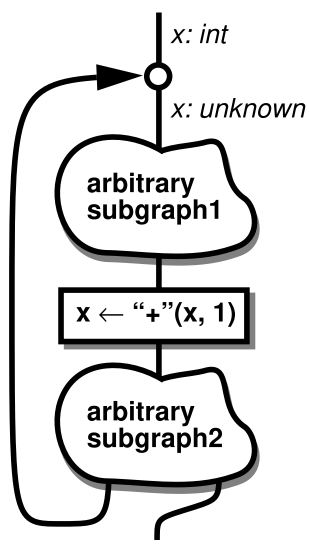
Loops pose special problems for type analysis. The basic problem is that the loop head is a kind of merge node, but the type information for some of the predecessors of the loop head (namely, the loop tail backwards branch) depends on the type information of the loop head, creating a circularity in the type analysis.



This chapter describes type analysis in the presence of loops and discusses synergistic interactions between loop type analysis and splitting.

11.1 Pessimistic Type Analysis

One approach to breaking the circularity would assume the most general possible value (the unknown value) and type (the unknown type) at the head of the loop for all names potentially assigned within the loop. These bindings would be guaranteed to be compatible with whatever bindings are subsequently computed for the loop tail, and so the type analysis would remain conservative and always produce legal control flow graphs. For example, in the following graph, the type of **x** would be automatically generalized to the unknown type at the loop head since there is an assignment to **x** within the loop:



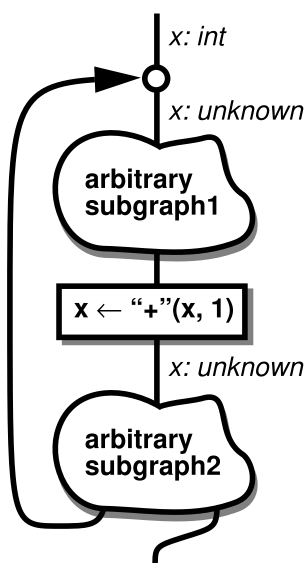
Unfortunately, this approach, which we call *pessimistic type analysis*, would sacrifice type information at precisely those places in the program that need the information the most to generate the best possible code: the inner loops of programs.

11.2 Traditional Iterative Data Flow Analysis

Traditional compilers resolve similar circularities in data flow analysis algorithms by performing the analysis *iteratively*. Iterative data flow analysis begins by assuming some data flow information for the loop head (usually just the information computed for the loop entry branch), analyzes the loop body, and then recomputes the data flow information at the loop head, this time including the results for the loop tail. If the information is unchanged (the *fixed point* in the analysis has been reached), then the analysis is done. If the information for the loop head has changed, then the previous analysis results are incorrect and the compiler must reanalyze the loop body with the new loop head information. This iteration will eventually terminate with a final least fixed point if the domain of the computed information is a finite lattice (a partial ordering with unique least and greatest elements representing the best and worst possible fixed points) and if the data flow propagation functions are monotonically increasing.

As traditionally applied, iterative data flow analysis operates on a fixed control flow graph. The compiler constructs a control flow graph, performs iterative flow analysis to compute some property of interest for all nodes in the graph, and then performs some optimization or transformation of the graph based on the computed information. If other optimizations need to be applied to the modified control flow graph, then most data flow analysis frameworks require recomputing all interesting information from scratch based on the new control flow graph. Incrementally updating data flow information after some modification to the control flow graph is an open research problem.

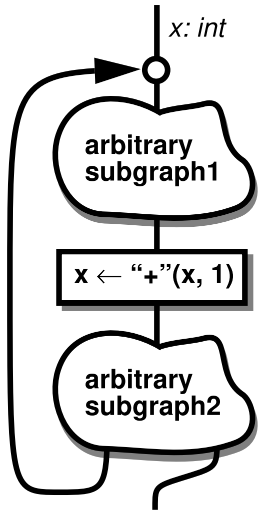
If the SELF compiler computed type information within loops by naively applying an iterative flow analyzer to the control flow graph prior to using the type information to perform optimizations such as inlining and splitting, then the computed type information would be very poor, virtually as bad as that computed using pessimistic type analysis. This surprising result is a consequence of SELF's use of messages for all computation, even computation as simple as arithmetic and instance variable accesses. If no messages within the loop are inlined, and an iterative flow analyzer is applied to compute the types of variables accessed within the loop, then any variables assigned the results of messages (including messages like **+**) must conservatively be assumed to be of unknown type, since the compiler would not know the type of the result of the non-inlined message. For example, naive iterative flow analysis would also compute that the type of **x** in the following graph must be unknown, since the type of the result of the non-inlined **+** message would be unknown:



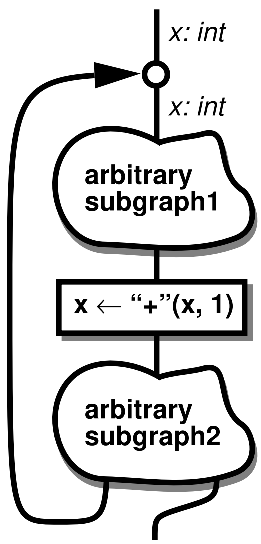
11.3 Iterative Type Analysis

The SELF compiler uses a technique we call *iterative type analysis* to compute precise types for variables modified in loops. Iterative type analysis is an extension of traditional iterative data flow analysis designed to cope with changing control flow graphs. As in iterative data flow analysis, the SELF compiler begins compiling a loop by assuming a certain set of types at the loop head. Unlike traditional flow analysis, however, the SELF compiler goes ahead and compiles the loop based on the initial assumed types. The SELF compiler's analysis may change the body of the loop as part of the analysis, such as by inlining messages or splitting apart sections of the loop body. Once the loop tail is reached, the types computed by the analysis for the loop tail are compared to the types assumed at the loop head. If they are compatible, then the loop tail is connected up to the loop head, and the compiler is done. Otherwise, the analysis must iterate, compiling a new version of the loop for the more general types.

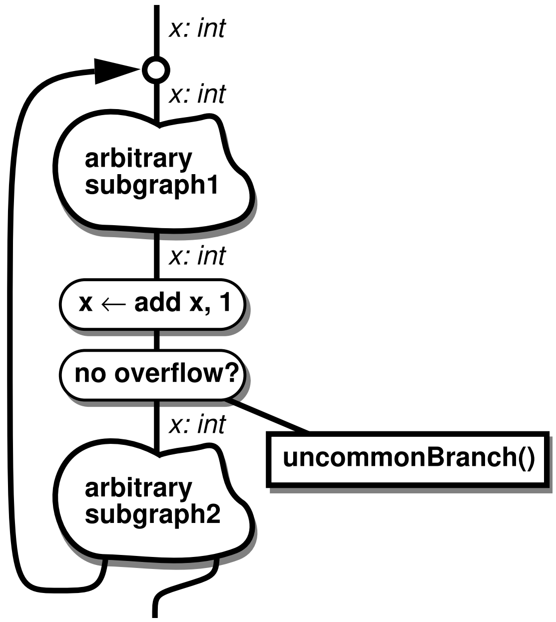
For example, when faced with the same control flow graph as before:



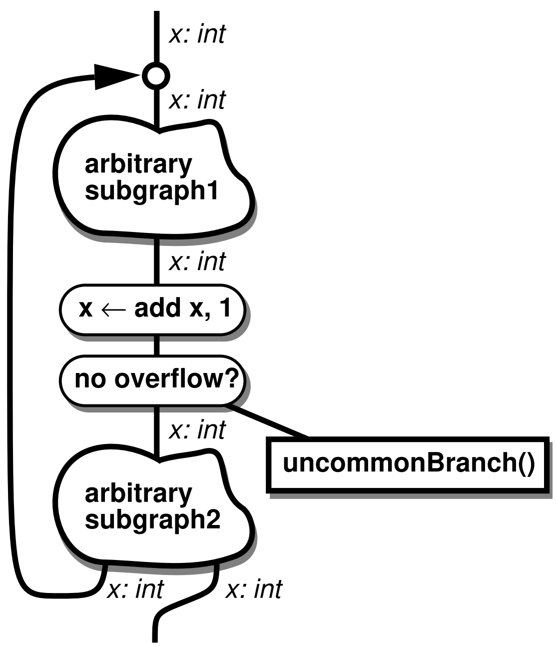
with iterative type analysis the compiler will first assume a set of types for the loop head derived from the loop entry types, in this case that **x** is an integer:



The compiler will then compile the body of the loop under these assumed types. When the compiler reaches the **+** message, its receiver, **x**, will be determined to be an integer (assuming no other assignments to **x** within the loop), and the compiler will inline the **+** message and the contained **intAdd** primitive to the following **add** machine instruction and overflow check:



After analyzing the rest of the control flow graph, **x** will be determined to be an integer at the loop tail, which is compatible with the types assumed at the loop head, and so the compiler is done:



The type information computed with this iterative type analysis is much better than the type information computed with either pessimistic type analysis or naive iterative data flow analysis. The **+** message has been inlined away with no extra overhead since iterative type analysis can compute that the type of **x** remains an integer throughout the loop. The other two approaches compute that the type of **x** is unknown at the top of the loop, and so at best a run-time type test would need to be inserted to check for an integer receiver of **+**. For other messages that are not be optimized using type prediction, the compiler would be prevented from inlining them at all without iterative type analysis.

The earlier description of the iterative type analysis algorithm purposely was vague on three points:

- What type information is assumed for the loop head initially?
- When is the type information computed for the loop tail compatible with the type information assumed at the loop head, allowing the loop tail to be connected with the loop head?
- How exactly does the analysis iterate if the loop tail type information is not compatible with the loop head type information?

Each of these questions has more than one reasonable answer, and different combinations of answers will result in different trade-offs among execution speed, compilation speed, and compiler simplicity. Iterative type analysis is thus revealed as a *framework* for a family of algorithms. The next three subsections provide the current SELF compiler's answers to these questions as well as some different answers from earlier SELF compilers. The question of compatibility will be taken up first, since the answer to this question impacts the answers for the other questions.

11.3.1 Compatibility

One central question is when the types computed at the loop tail are considered compatible with the types assumed at the loop head. One extreme approach would consider a loop tail type compatible with a loop head type if the loop head type *contains* the loop tail type (treating types as sets of values as described in section 9.1.4). This approach would make loop tails compatible with loop heads as often as possible, thus helping iterative analysis reach the fixed point quickly and saving compile time. Unfortunately, the approach also could sacrifice much of the type information available at the loop tail in the form of precise types by connecting a loop tail to a loop head compiled assuming much less specific type information. For example, if a variable were bound to the unknown type at the loop head and to the integer map type at a loop tail, then this compatibility rule would consider these two bindings compatible and allow the loop tail to be connected to the loop head. While this is legal, it would sacrifice type information available at the loop tail that could lead to better generated code if the loop tail were not connected to the loop head. We want a type compatibility rule that will not sacrifice this much of the compiler's hard-won type information.

The opposite extreme position on compatibility would only consider two types compatible if they are exactly the same. This position would avoid any loss of type information, since the loop tail must have exactly the same type information as a loop head to connect to it, but it could easily lead to many iterations as part of iterative type analysis. For example, there are $2^{2^{30}} - 1$ different types relating only to integers (the power set of the 2^{30} different integer constants, less the empty set), and type analysis could iterate this many times with such a definition of compatibility. Thus it is impractical to define type compatibility this narrowly.

The current SELF compiler uses a type compatibility rule midway between these two extreme positions. A type at a loop tail is considered compatible with a type at a loop head if the loop head type contains the loop tail type, and the loop head type does not lose any map-level type information. If the loop tail type is a constant type or an integer subrange type, then the loop head type can be no more general than the enclosing map type to be compatible. If the loop tail type is a union type whose components have different maps, then the loop head type can be no more general than the union of the corresponding map types. This compatibility rule implements the heuristic that map-level type information is the most important kind of type information and provides the lion's share of optimization opportunities, since map-level type information is the most general kind that still supports static binding and inlining of messages. Also, since in practice the compiler does not encounter more than a few different map types for a single variable,* the fixed point in the iterative type analysis is reached fairly quickly.

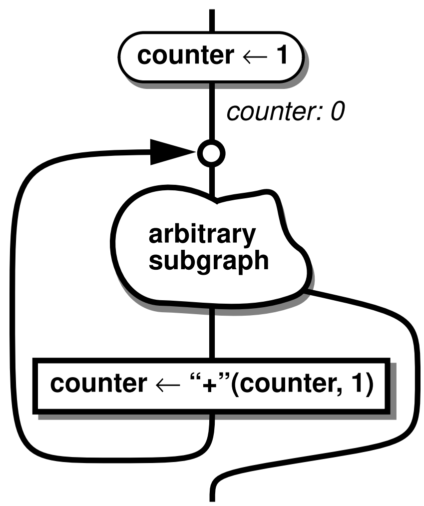
This strategy is amended in situations in which both common-case and uncommon-case branches exist, such as in uncommon branch extension methods (described in section 10.5) or when simulating a version of the system without lazy compilation of uncommon branches. Since the compiler is much more conservative when compiling uncommon branches, connecting a common-case loop tail to an uncommon-case loop head would also sacrifice the performance of that common-case path. Therefore, the compiler considers a loop tail compatible with a loop head only when the weight of the loop head is at least as great as the weight of the loop tail. Additionally, to minimize compilation time spent on uncommon branches, the compiler uses the most conservative strategy for type compatibility, considering an

* This is true mainly because currently there are only a few ways that the compiler can infer map-level type information. This characteristic may not be true in the future with the introduction of adaptive recompilation using polymorphic in-line caches [HCU91], and so this type compatibility rule may need to be revised to keep down the number of iterations needed to reach the fixed point.

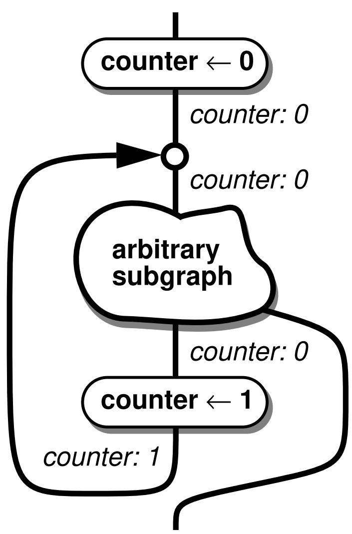
uncommon-case loop tail compatible with an uncommon-case loop head if the types at the loop head merely contain the types at the loop tail. Along uncommon branches, conserving compilation time and space is more important than preserving map-level type information.

11.3.2 Initial Loop Head Type Information

Another question to be answered is exactly what types are assumed initially at the loop head. An obvious strategy would assume that the loop head's type information was the same as the loop entry's type information. This approach is simple and precise: beginning with the loop entry's type information would enable the compiler to compute the most precise fixed points. Unfortunately, this approach frequently would cause the algorithm to iterate at least once, since in most cases at least one name would be assigned a type in the body of the loop different from that initially assumed at the entrance to the loop. For example, the following loop is typical of a standard **`for`** loop written in SELF, with a loop counter initialized to zero and incremented at the end of the body of the loop:

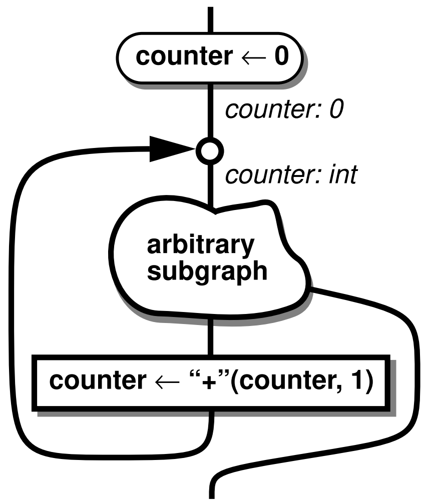


If the loop head types were assumed to be the same as the loop entry types, then after one iteration (and inlining the **`+`** message and constant-folding the **`intAdd`** primitive) the compiler would produce this graph:

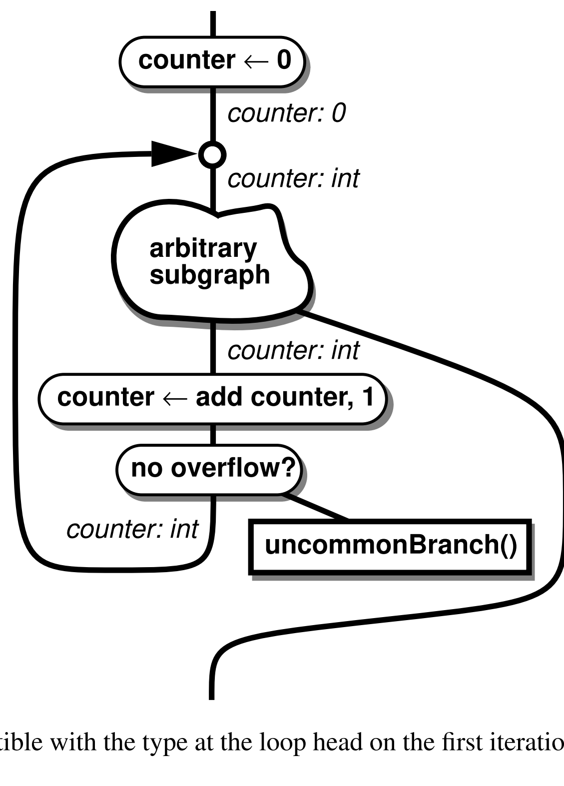


The type of **`counter`** computed at the loop tail would not be compatible with the type assumed at the loop head, since the type assumed at the loop head would not contain the corresponding type at the loop tail. Consequently, the analysis would be required to iterate with some more general type than the **`0`** constant type. Since most loops modify some loop counter from its initial constant value, this initial throw-away iteration would be expected for most loop structures.

Assignable local variables are likely to be assigned within the body of the loop a value different from the initial value computed at the entrance to the loop. To avoid an extra iteration, the current SELF compiler *generalizes* the types of assignable names at the loop head over the corresponding types at the loop entrance. To preserve map-level type information, this generalization is only to the enclosing map type (or union of map types if the loop entrance type is a union type). In the above **`for`** loop example, the compiler will generalize the initial type of the loop counter to the integer map type:



Then after compiling the loop body the compiler will reach the following graph:



The type at the loop tail is now compatible with the type at the loop head on the first iteration; no additional iterations are needed.

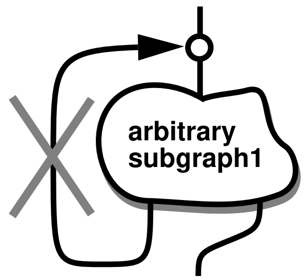
This generalization heuristic works well in the cases in which the assignments to local variables are of the same map as the initial value, such as is the case with integer loop counters. It also usually saves an iteration in iterative type analysis over the version of iterative type analysis in the earlier compiler. However, there are some situations in which this generalization of all assignable names is overly pessimistic. Some variables may be assignable but not actually assigned within the loop being analyzed; generalizing their types may lose type information. For example, in a doubly-nested loop, both loop counters are assignable, but the outer loop counter is not assigned within the inner loop. At the inner loop head the current SELF compiler will generalize the type of the outer loop counter from some integer subrange type (computed after the initial comparison against the outer loop's upper bound) to the more general integer map type. This unnecessary generalization can sacrifice possibilities for optimization, such as applying integer subrange analysis to eliminate unneeded array bounds checks or arithmetic overflow checks involving the outer loop counter. A better approach that would not lose this kind of type information would determine which variables might be assigned within a particular loop and only generalize those variables. This analysis is complicated by the fact that most of these assignments occur within blocks making up the body of the loop control structure, and accurately determining which variables might be assigned within some loop would require computing some sort of transitive closure of all blocks that could be invoked by messages sent within the loop. If implemented, this approach would solve the doubly-nested loop problem.

To minimize compilation time for uncommon-case loops, the SELF compiler generalizes the types of assignable variables all the way to the unknown type at uncommon-case loop heads, instead of just the enclosing map type as with common-case loop heads. In conjunction with the lenient rules for type compatibility of uncommon-case loop tails, this generalization ensures that an uncommon-case loop tail always connects to an uncommon-case loop head on the first pass; no iterations are needed for uncommon-case loops.

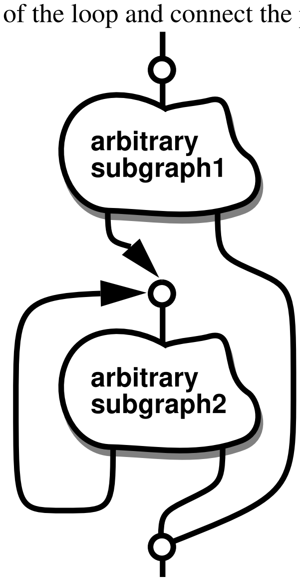
11.3.3 Iterating the Analysis

The final question remaining to be answered to complete the description of iterative type analysis is what happens when the loop tail is not compatible with a loop head, forcing the analysis to iterate. In traditional iterative data flow analysis, the information previously computed about the nodes in the loop is thrown away, new information is computed for the loop head by combining the information assumed at the previous iteration with the information computed at the loop tail, and finally the information for the body of the loop is recomputed based on the new, more general loop head information. This approach cannot be directly applied to iterative type analysis, since as part of analyzing the body of the loop the control flow graph is optimized based on the information. If the type information assumed at the loop head is determined to be overly optimistic, the optimizations performed based on that information are no longer valid. Backing out of optimizations like inlining and splitting would be very difficult.

The approach taken in the SELF compiler's iterative type analysis is to create a fresh new copy of the part of the control flow graph representing the loop body, and reapply type analysis to this fresh unoptimized copy. The fresh loop head is simply connected downstream of the previous iteration's incompatible loop tail, in effect *unrolling* the loop for the new type information. For example, if after reaching the loop tail on the first iteration it proves incompatible with the loop head:



the compiler will simply create a new copy of the loop and connect the previous copy's loop tail to it:



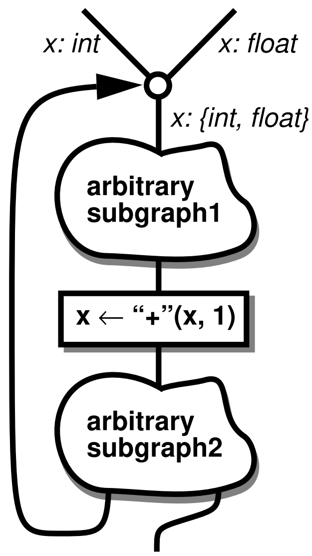
Unrolling loop bodies this way has important fringe benefits. Type tests and other code can get "hoisted" out of the normal loop body into earlier versions. For example, if the type of some variable is unknown at the top of the initial loop body, then the first loop head will assume the type of the variable is unknown. If the variable is treated like an integer inside the loop, such as by sending it **+** messages, then the compiler will use type prediction to insert a run-time type test and use splitting to compile a separate path through the loop body where the variable is known to be bound to an integer. When the loop tail is reached, the integer type will not be compatible with the unknown type at the loop head type, and so a separate version of the loop body will be compiled just for when the variable is bound to an integer. This second version of the loop will contain no type tests, since through the loop body the variable will be bound to an integer. If this turns out to be the common case (as is expected), then control will pass through the initial version of the loop once and thereafter remain in the faster integer-specific version of the loop.

This unrolling approach is quite simple to implement. However, this strategy may waste some compiled code space (and hence compile time) if the unrolled copies are very similar. Some compiled code space is reclaimed by generalizing the types of assignable names before the first iteration, as described in section 11.3.2, thus usually saving an iteration and correspondingly a whole copy of the loop body. An alternative approach would replace the original overly optimized loop body with the fresh new loop body, redirecting the predecessors of the original loop head to flow instead into the fresh new loop head. This would allow the compiler to conserve compiled code space and compile time by sharing similar parts of the loop body across what would have been separate copies. Unfortunately, this alternative approach would be more complex to implement and would require some mechanism to ensure that the new loop head starts out with more general type information that just what its predecessors indicate. Path-based type information exacerbates both of these problems. Future research could explore designs for iterating the type analysis that conserve compiled code space and compile time, work with path-based type information, and minimize compiler complexity.

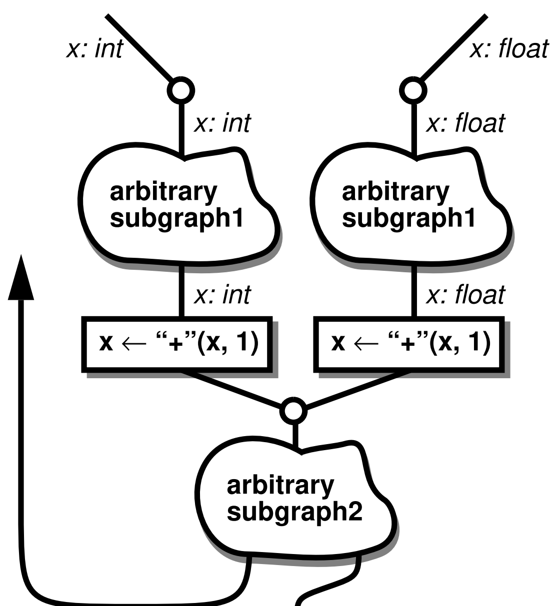
11.4 Iterative Type Analysis and Splitting

Iterative type analysis of loops and splitting have been carefully crafted to enable the SELF compiler to compile *multiple versions* of a loop, each version optimized for different combinations of run-time types. The inner loops of programs are the most important to optimize well, and by compiling separate versions of loops for different run-time types the SELF compiler is able to generate very good code for these loops.

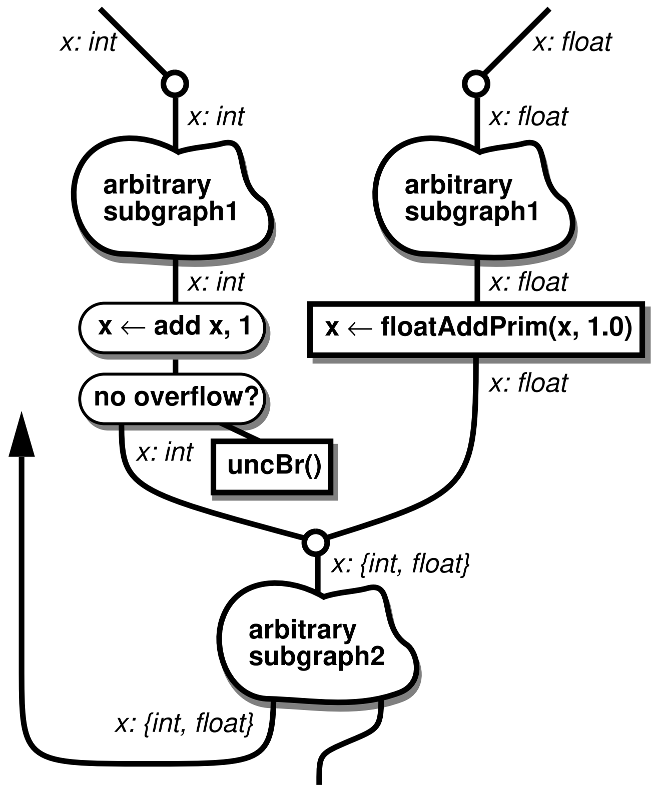
Since loop heads are just a special kind of merge node, they can be split apart just like merge nodes. For example, if two branches merge together at a loop head, and along one branch a variable has type integer, and along the other branch the variable has type floating point number:



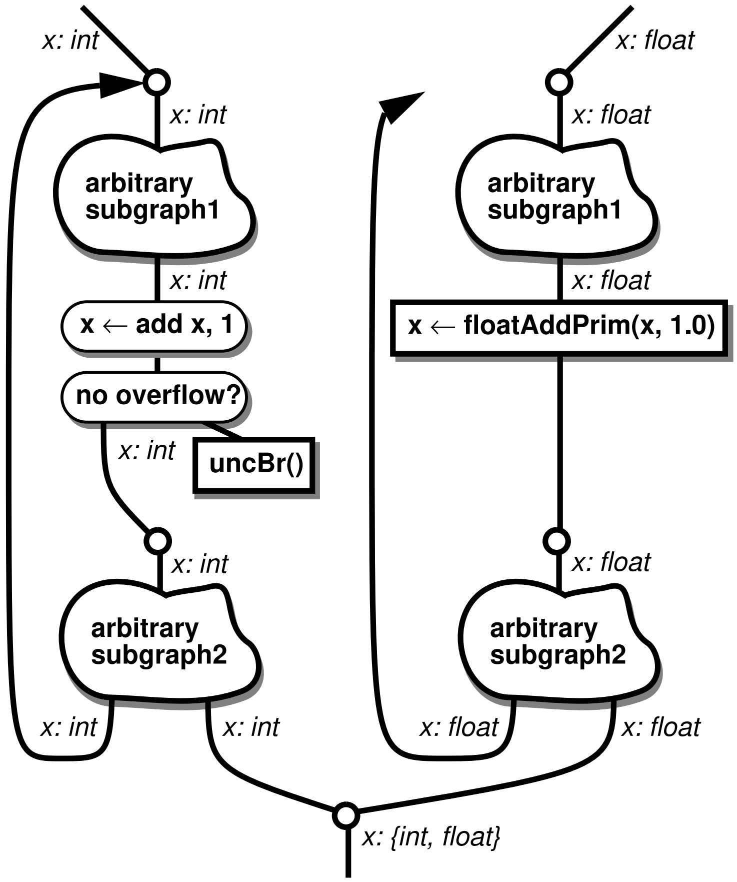
the compiler can split apart the loop head merge node if the variable is sent a message within the loop:



This splitting can create multiple loop heads for a single loop, each loop head for different types. The current SELF compiler's unrolling approach to iteration also creates multiple loop heads each with different types. Consequently, a loop tail may have several loop heads from which to choose when trying to connect to a type-compatible loop head. The compiler tries to find the highest-weight loop head with the most compatible types to connect a loop tail to. If no loop head is compatible, then the compiler tries to split the loop tail to match some loop head. For example, after inlining away each of the **+** messages and then compiling the rest of the loop the compiler reaches the following graph:

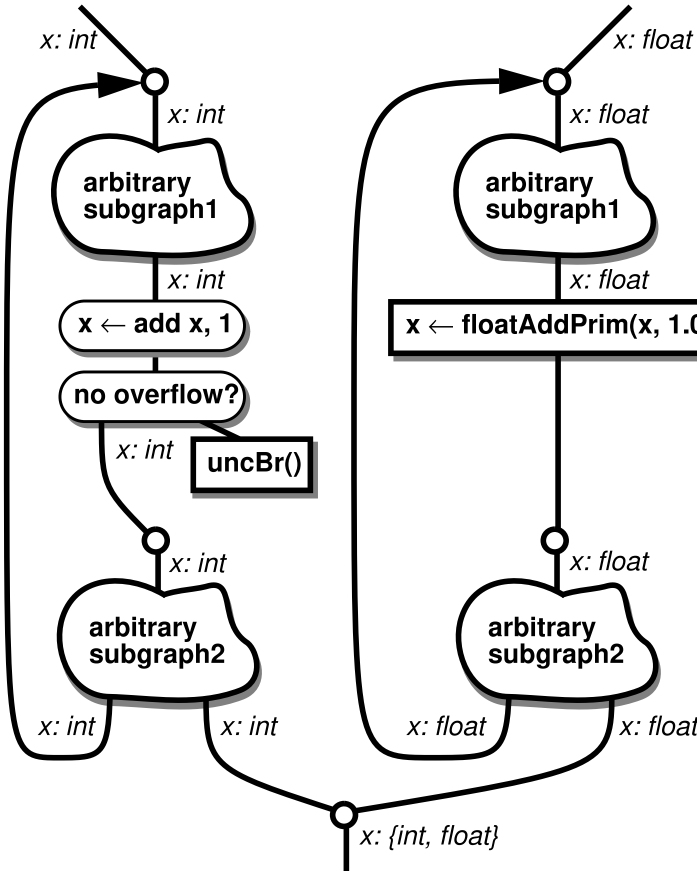


The loop tail type *{int, float}* does not directly match either of the loop head types. So the compiler tries to find a subset of the paths through the loop tail that can be split off and connected to a loop head. The compiler determines that the left-hand path producing the integer type binding would be compatible with the left-hand loop head if split off, so the compiler splits the loop tail accordingly (assuming that the cost of splitting is low enough) and connects the split loop tail to get the following graph:



After splitting a loop tail and connecting one of the copies, the compiler turns its attention to the remaining loop tail copy. This loop tail is itself checked against the available loop heads, and possibly split again if it cannot be connected directly to a loop head. This process continues until either the loop tail is successfully connected to a loop head, in which case the compiler has finished compiling the loop, or the loop tail cannot be either split or connected to a loop head, in which case the compiler falls back on its unrolling strategy, creating a fresh copy of the loop (and a new loop head) for the loop tail's combination of types. This process of splitting loop tails to match loop heads is guaranteed to terminate with either a successful connection or by unrolling the loop since at each split the number of paths in the leftover loop tail decreases.

In the case of the example loop, the compiler determines that the leftover loop tail is compatible with the right-hand loop head and connects it up to get this control flow graph:



This effect of splitting loop heads and subsequently splitting loop tails often leads the compiler to generate multiple versions of a single source loop, each compiled version for different type bindings. In the above example, the compiler has produced two completely independent compiled versions of the original source loop, with one version for the case where **x** is an integer and a separate version for the case where **x** is a floating point number. Each version of the loop is optimized for a particular combination of types, and so can be much faster than a single general version of the loop could be. In the above example, if the compiler were constrained to generate only a single version of the loop, then the compiler would be forced to use type casing to optimize the **+** message for the integer and floating point cases, introducing extra run-time overhead for the type test. This extra overhead can become significant as the number of operations in the loop that could be split apart increases.

This splitting of loops for different types enables the SELF compiler to compete with optimizing compilers for traditional languages in execution performance without sacrificing the extra power available to SELF programmers, such as pure message passing and generic arithmetic support. Typically, one version (or sometimes a few versions) of a loop will be split off and optimized for the common-case types; these types include those that would be used in the equivalent program in a traditional language, such as integers and fixed-length arrays. This common-case version of the loop can achieve quite good performance, since the compiler is compiling a version of the loop specific to those common-case types. The SELF compiler gets nearly the same information that is available to the compiler for the traditional language and hence can generate code that runs nearly as fast as that generated by the compiler for the traditional language. The SELF compiler will also generate an extra version of the loop to handle any other types or situations that might arise that do not fall into the category of common-case types or situations; this extra version of the loop implements the additional semantics available in SELF in the form of pure message passing and generic arithmetic. With lazy compilation, this extra version will be in the uncommon branch extension method and usually is never actually compiled at all. The separation of the common-case version(s) from the uncommon-case version(s) is the primary mechanism used by the SELF compiler to rival the performance of traditional languages.

11.5 Summary

The SELF compiler uses iterative type analysis to infer relatively precise types in the presence of loops. This algorithm performs optimizations such as inlining and splitting on the body of the loop as part of each iteration in the analysis. As a result, the compiler computes much more precise type information than would be possible with a standard data flow analysis algorithm. The SELF compiler uses various heuristics to reach the fixed point as quickly as possible without sacrificing a significant amount of type information. These heuristics include automatically generalizing the types of assignable names to the enclosing map type at loop heads and connecting loop tails to loop heads as long as the loop head does not sacrifice map-level type information. When a loop tail does not match any available loop head, the loop is unrolled by appending a fresh copy of the loop body to the loop tail and continuing the analysis.

The SELF compiler treats loop heads like merge nodes and will split a loop head apart if a node downstream of the loop head wants some type information that was diluted by either the loop head merge node or by a merge node before the loop head. This splitting creates multiple loop heads from which to choose when connecting a loop tail to a loop head. Unrolling a loop when a loop tail is not compatible with any available loop head also creates several loop heads from which a loop tail can choose. A loop tail itself can be split apart when it does not directly match any loop head, but a subset of the paths that reach the loop tail do match a loop head. With both loop heads and loop tails being split apart, the SELF compiler frequently ends up creating several completely independent versions of a loop, with each loop head leading to a matching loop tail. Each of these versions can be much faster than could a single general version of the loop, and consequently generating multiple versions of loops each for different run-time types is one of the key sources of SELF's execution speed. When combined with lazy compilation of uncommon branches, the SELF compiler frequently compiles one or two common-case versions of a loop to handle the standard situations, with an additional general version of the loop compiled only if and when needed.

Chapter 12 Back-End of the Compiler

Type analysis, inlining, and splitting are performed simultaneously as the first pass of the SELF compiler. This chapter describes the remaining passes of the SELF compiler, describing those optimizations included to be competitive with traditional optimizing compilers.

12.1 Global Register Allocation

In many ways, good register allocation is the most important "optimization" in traditional compilers. The effect of register allocation is felt through all compiled code, unlike most other optimizations, and a poor allocator can hurt performance so much that any improvements caused by other optimizations are inconsequential. However, register allocation historically has been much less important in implementations of pure object-oriented languages (and many other kinds of high-level languages), primarily because these high-level languages contained so many procedure calls that register allocation became of secondary importance. Register allocation algorithms must flush caller-save registers to their home stack locations across procedure calls, and in language implementations with many procedure calls there is not enough straight-line code between calls to justify any but the simplest of allocators. In contrast, the SELF compiler is designed to eliminate most procedure calls and frequently compiles whole versions of loops with no internal calls. Global register allocation thus again becomes important, especially in light of our goal to rival the performance of traditional optimizing compilers.

A straightforward register allocator would assign each variable in the source program to a register or stack location, such that two simultaneously live variables are not assigned the same register or stack location. This naive approach would work poorly in an environment like SELF where method inlining is commonplace, since a formal variable name of an inlined method will be simultaneously live with the corresponding actual parameter, and consequently the two variables will be allocated separate locations, even though both variables will always contain the same value. With many layers of inlined methods this duplication of what conceptually should require only a single location can introduce a huge overhead of register copying as layers of inlined methods are entered and exited.

A better system could perform copy propagation prior to register allocation, replacing uses of each of the formal variables in inlined routines with uses of the corresponding actuals. Unfortunately, straightforward application of copy propagation could make supporting complete source-level debugging more difficult, since a given variable name might be replaced by several different variable names at different points during its lifetime.

The SELF compiler incorporates an alternative strategy, introducing a new abstraction between a name and its assigned location called a *variable*. Names that are always *aliases* of one another (such as actuals and formals) are mapped to the same variable object, and each variable object either is assigned a single run-time location (i.e., a register or a stack location) or is marked as a compile-time constant (and so does not need a run-time location). By assigning locations to variables rather than names, the SELF compiler eliminates the impact of artificial name distinctions caused by inlining and reduces the SELF register allocation problem to one similar to that faced by traditional optimizing compilers with built-in control structures and limited inlining.

An earlier version of the SELF compiler included a much more ambitious register allocator. Instead of allocating locations to names or even to variables, the earlier compiler allocated locations to values. This makes perfect sense, since values are precisely those run-time "bit patterns" that are manipulated by the program; names are merely convenient handles by which programmers and the compiler refer to values. The earlier compiler went to the extreme of allowing each use of a value and each straight-line inter-use segment of a value's lifetime to be independently allocated, either in a location or marked as a compile-time constant with no run-time existence. This design provided the allocator with a lot of freedom to implement various parts of a value's lifetime in different ways. Unfortunately, this earlier register allocation design had three main drawbacks:

- The earlier allocator was very slow. This was primarily caused by the extremely fine granularity of allocation (a single use or lifetime segment). The current implementation, like most other register allocators, allocates whole lifetimes in a single step and so runs much faster.
- The earlier allocator had to work hard to make sure that long sequences of lifetime segments and uses of a single value were all allocated to the same location, since otherwise register moves might be inserted in the middle of a value's lifetime unnecessarily. Additionally, a name that is bound to several variables over its lifetime, such as a

counter that gets incremented each time through a loop, should get all its values allocated to the same location so that unnecessary register moves are not inserted at assignments to the name. Naturally the allocator was imperfect, sometimes introducing register moves in awkward places in the generated code, and consequently the earlier SELF compiler did not achieve the same level of code quality as that in a traditional register allocator (traditional register allocators have a much easier job since usually they support neither complete source-level debugging nor massive inlining of user-defined control structures). The current SELF compiler's register allocator overcompensates by requiring that a name is bound to a single variable for its entire lifetime, and a variable be allocated to a single location for all its lifetime.

- The earlier register allocator never supported the source-level debugger. The intention was for the compiler to generate tables describing the mappings from names to values and values to registers as part of the information generated for use by the debugger. (The current version of this debugging information will be described in detail in section 13.1.) These mappings would be indexed with the hardware program counter, since the mappings depended on where in the compiled code the program was stopped. Unfortunately, the design of a space- and time-efficient representation of these mappings proved difficult and was never completed. The current SELF compiler avoids the complexity of time-varying allocations by restricting the mappings to be position-independent.

Compared to the earlier more ambitious register allocation strategy, the current strategy is simple, fast, and reasonably effective.

12.1.1 Assigning Variables to Names

The compiler constructs the name/variable mapping as part of type analysis. In the compiler's internal representation, each name object refers to its corresponding variable object, and each variable keeps track of which names refer to itself, called the variable's *alias set*. When a name is first assigned an initial value, the name is simply added to the alias set of the variable associated with the right-hand-side of the assignment. If the name is subsequently assigned a new value, it can no longer be considered an alias of the initializing expression, and consequently the name is removed from its old alias set and allocated its own fresh variable object. Once type analysis completes, each name will be associated with a single variable that represents the name's alias set: those names that always contain the same value and hence can be allocated the same location.

Also during type analysis, the compiler records extra information with names for each occurrence of the name as an operand of a run-time expression. This per-use information records the preferred location (such as the particular register or stack location required as part of parameter passing) or kind of location (such as any address register on the 68000 architecture) for that use, plus the weight of the control flow graph node containing the use (weights are described in section 10.2.3.2). This extra information helps the register allocator to avoid unnecessary register movement and to concentrate its efforts on the most frequently executed parts of the control flow graph.

12.1.2 Live Variable Analysis

After eliminating redundant constants (described later in section 12.2) and unneeded computations (described in section 12.3), the compiler begins the actual register allocation phase. As with most global register allocators, the SELF compiler computes the *register interference graph* to determine when the lifetimes of two names overlap. The nodes in the interference graph are variable objects rather than name objects.

To compute the register interference graph, the compiler makes a backwards pass over the control flow graph. As the compiler passes over the graph, it maintains a set of *live variables*, computed as the set of variables that have been used downstream in the control flow graph but have not yet had their initial definitions. When the compiler reaches a variable's initial definition, the compiler removes the variable from the live variable set. When the compiler reaches a use of a variable not already in the live variable set, i.e., the last use of some variable, the compiler adds the newly-live variable to the live variable set and adds interference links between the newly-live variable and the other variables already in the live variable set. By scanning backwards through the control flow graph, the compiler can detect the end of a variable's lifetime when it encounters its last use (which will be the first occurrence of the variable in the backwards pass). The beginning of a variable's lifetime can be detected without a corresponding forward scan since the initial assignment to any name is known statically and is represented by a different kind of control flow graph node when the control flow graph is constructed.

Merges, branches, and loops complicate this backwards traversal. When the compiler reaches a merge node in the backwards traversal, it places all the merge's predecessors but the first on a special stack of <control flow graph node,

live variable set> pairs representing work pending, and then the compiler processes the merge's first predecessor. When the compiler reaches a particular branch node for the first time during the live variable analysis, the compiler must stop processing this path and wait for the other branch successor to be analyzed before the compiler can process the branch's predecessors. The compiler records the current live variable set with the branch node, pops a <node, live variable set> pair off the pending nodes stack, and resumes processing this other node with the corresponding live variable set. Once the branch node is reached for the second time from the other successor branch, the branch's predecessor can be processed; the live variable set for the branch's predecessor is the union of the current live variable set and the live variable set recorded as part of the earlier visit to the branch node. In the absence of loops, this search pattern guarantees that a node is processed after all of its successors have been processed (i.e., traverses the graph in reverse topological order), ensuring correctness of the computed live variable sets.

Unfortunately, in the presence of loops, not all nodes can be processed before their successors, and some sort of iterative algorithm is needed to compute the fixed point of the live variable sets. Whenever the compiler runs out of pending <node, live variable set> pairs to process, but there remains at least one branch node whose predecessor is not processed yet (e.g., because its other successor eventually leads to a **`_Restart`** backward branch to an earlier loop head), the compiler simply forces the branch node to process its predecessor early. The compiler initially assumes that the live variables from the unprocessed successor are a subset of the live variables from the processed successor. Once the other branch node's successor is finally processed, the compiler can verify whether this assumption was correct. If it was, then the compiler is done with this branch node, and can find other nodes to process. If, however, the second successor had some live variables that are not in the first successor's live variable set, then the earlier analysis of the branch node's predecessor was inadequate, and the live variable analysis must iterate by reprocessing the branch node's predecessor with a larger live variable set.

Instead of completely repeating the analysis as would be the case with a normal iterative data flow algorithm, the SELF compiler uses a special mode of live variable analysis to update the earlier results incrementally. In this mode, the compiler propagates the *difference* between the previous live variable set and the new live variable set. This *incremental live variable set* is initialized as the set difference between the live variable sets of the second predecessor and the first predecessor. Incremental analysis differs from normal non-incremental analysis in that no variable ever needs to be added to the incremental live variable set, since the variable would have been present in the previous normal live variable set and so is not present in the difference between the previous and new live variable sets. Variables still are removed from the incremental live variable set as their initial definitions are reached, since they no longer appear in the new live variable set. Once the incremental live variable set becomes empty, the previous and new live variable sets are the same, and incremental analysis of the path can stop. Incremental re-analysis is faster than non-incremental re-analysis since the incremental live variable sets are smaller and it is faster to check whether the incremental live variable set is empty than to check whether the previous and current live variable sets are the same.

The compiler represents both live variable sets and the per-variable sets of interfering variables with a dual bit-vector/linked-list data structure. This representation of sets supports both constant-time set member testing via bit vectors (each variable is allocated a unique integer index used as its position in the bit vectors) and linear-time (in the size of the set) iteration through the elements of the set via linked lists. Execution profiling of the SELF compiler shows that neither the extra space cost for two representations of the same set nor the extra compilation time cost to copy two representations of the same set at merge nodes is significant. Therefore, this dual representation supports set operations better than either traditional representation.

12.1.3 Register Selection

After the register interference graph is built, the compiler visits each variable and allocates it a location. The compiler uses the weights of the uses of the names associated with the variables to determine the order in which to allocate variables to registers, with the highest weight variables allocated first. If the variable has no run-time uses and either the variable is a compile-time constant or the variable is not a source-level variable and so not visible to the outside world, then it is not allocated a run-time location. Otherwise, the preferences specified with the uses of the names associated with the variable are used to pick a register or stack location that is different from any already chosen for an interfering variable.

This allocation strategy is simple and fast. However, it is limited by only allocating a single location or compile-time constant to each variable. Since each name is associated with a single variable, completely disjoint portions of a name's lifetime, sometimes created by splitting, cannot be allocated to different registers or more importantly cannot be associated with different compile-time constants. This is especially common for names bound to either the **`true`** constant or the **`false`** constant, such as the results of comparison operations. If the contents of this name might be visible from the debugger, then with the current register allocation limitation the name must be allocated a run-time location and this location must be initialized at run-time with either **`true`** or **`false`**. This is true even though the program itself will have no need for the run-time value, since after splitting the value will be encoded in the position in the program (just as traditional compilers encode the result of boolean tests within **`if`** statements by positions in the compiled code).

Fortunately, in most common circumstances this particular problem can be side-stepped by creating new names for formals of inlined methods (rather than reusing the names of the actuals as would be more natural and concise) and specially handling the debugger's view of expressions used as arguments to inlined methods. For example, if the **`ifTrue:`** message is sent to the result of a comparison, then typically two **`ifTrue:`** methods will get inlined, one for receivers of type *true* and another for receivers of type *false*. If name of a formal were the same as the name of the corresponding actual, then in this case the single name which is bound to the result of the comparison would be reused as the names of the receiver formals in both inlined versions of **`ifTrue:`**. If this formal is visible to the debugger, then this approach would cause the register allocator to allocate a run-time location to hold the value of the result of the comparison. Instead, the current compiler creates new names for formals, and assigns the actuals to the formals. Then the receivers of the two **`ifTrue:`** methods have their own names, which can correctly be allocated to compile-time constants. The original comparison result name is no longer used by the debugger, and so can be left unallocated. Thus for this common case the compiler will not introduce extra run-time overhead.

However, in general, the limitation of a single compile-time constant or location for each name is still a problem. The earlier register allocator was able to solve this problem by allowing the allocation to be different for different parts of the control flow graph, but proved too inefficient for practical use. Register allocators for other compilers frequently allocate disjoint subregions of a variable's lifetime to different registers, and sometimes can even split a variable's lifetime into separately-allocatable regions [CH84, CH90], but these systems do not simultaneously support complete source-level debugging. We continue to search for some "happy medium" register allocator that combines some form of flexible position-dependent allocation with high allocation speed and compact debugging information.

12.1.4 Inserting Register Moves

After allocation, a pass through the control flow graph inserts register move control flow graph nodes where needed. Places where the compiler inserts register moves include before assignments nodes whose left- and right-hand-sides are allocated to different registers and before message send nodes to move arguments into the locations required by the calling convention.

12.1.5 Future Work

The compiler's computation of variables as the set of aliased names is not always as large as possible. For example, consider the following simple sequence appearing in most looping control structures:

```
i: i + 1.
```

After executing the **`+`** message but before executing the **`i:`** message, both the original value of **`i`** and the result of **`+`** are simultaneously live and holding different values. Consequently, the compiler assigns the two expressions different variables. However, if the compiler knows that **`i`** is an integer and inlines the **`+`** and then the **`intAdd`** primitive call down to a simple **`add`** instruction (ignoring the overflow check for the moment):

```
i ← add i, 1
```

the original value of **`i`** is never used after the **`add`** instruction. Thus the actual run-time lifetimes of **`i`** and the result of the **`+`** message do *not* overlap, and they could (and usually should) be allocated to the same register.

We say that two different variables are *adjacent* if their lifetimes do not overlap and one variable is assigned to the other. Adjacent variables usually should be allocated to the same location to minimize register moves when one variable is assigned to the other. One way of achieving this would be to combine the two variables into a single larger variable; this transformation is called *subsumption*. Unfortunately, this combining cannot be performed as part of forwards type analysis, since when the second variable's lifetime begins (such as after the **`+`** message above) the compiler does not yet know that there will be no additional uses of the first variable. However, it may be possible to augment the live variable analysis pass to detect adjacent variables, mark them as such, and extend the register allocator to allocate adjacent variables to the same location. An alternative approach would compute variables in a separate backwards pass after type analysis, thereby providing last-use information to the analysis and implicitly coalescing together what would have been different but adjacent variables.

12.2 Common Subexpression Elimination of Constants

After type analysis, the compiler performs a pass over the control flow graph to eliminate redundant loads of constants. This pass creates extra names to represent available constants and replaces subsequent redundant loads of constants with simple assignments to these new names.

Unfortunately, this technique does not interact well with the current register allocation strategy. New variable objects are created during this phase for the newly created names of constants. Since distinct variables are allocated independently and hence frequently to different registers (as described in section 12.1.3), extra register moves may be inserted at both the point of the initial constant load instruction (if the register for the new variable is different from the register for the result of the load instruction) and at each eliminated use of the constant (if the new variable is allocated to a register different from the original variable for the use). These extra register moves diminish the benefits of common subexpression elimination of constants. Some of these problems could be avoided by performing common subexpression elimination of constants in the same pass as type analysis and name/variable mapping construction, thus eliminating the extra register move when using a saved constant; unfortunately, subtle interactions with other parts of the compiler complicate this approach. Also, the techniques for allocating adjacent variables to the same location as described in section 12.1.5 might help.

However, even ignoring these problems, common subexpression elimination of constants (and in fact common subexpression elimination of any relatively cheap computation such as an arithmetic operation) can still *slow programs down*, if the variable for the constant is allocated to a stack location rather than a register. Storing and subsequently fetching a constant from a stack location is significantly costlier than redundantly loading a 32-bit constant value into a register. Common subexpression elimination and register allocation should to work together to avoid these sorts of inadvertent pessimizations. Unfortunately this kind of inter-pass cooperation is notoriously difficult to manage.

In practice, common subexpression elimination of constants has a mixed effect. According to the results shown in section B.3.10 of Appendix B, some benchmarks speed up with this optimization, while others slow down. Clearly this technique should be redesigned and reimplemented to improve its effectiveness.

12.3 Eliminating Unneeded Computations

In a third phase the SELF compiler scans all names to find those whose value is computed but never used. If such a computation exists that has no side-effects, such as a simple assignment, a load-constant instruction, a memory fetch instruction, an arithmetic instruction, an object clone primitive call, or a call to any other side-effect-free primitive, then the compiler simply splices the unnecessary operation out of the control flow graph.

This optimization improves the performance of SELF programs by a few percent on average, according to the measurements shown in section 14.3.

12.4 Filling Delay Slots

After register allocation, the SPARC version of the SELF compiler attempts to fill branch and call delay slots. In the SPARC architecture, branch instructions have a one-instruction delay slot that either is always executed or is executed only if the branch is taken (an *annulled* branch). The SELF compiler attempts to fill these delay slots with useful instructions, preferably from before the branch instruction, otherwise from the most likely successor, and finally from the remaining branch successor. Call and jump/return instructions similarly have one-instruction delay slots that the SELF compiler attempts to fill from before the instruction.

Many nodes in the control flow graph do not generate machine instructions, such as assignment nodes and other bookkeeping nodes, and therefore this searching for an instruction to move to a delay slot may scan several control flow graph nodes before either locating a node that generates a single instruction or finding a node that cannot fit into a delay slot (such as a message send node). If a node is found to put in the delay slot, then all the nodes between the branch or call and the target node are spliced out of the control flow graph (adjusting the predecessor or successor links of the branch or call appropriately) and the removed chain of nodes is stored in the branch or call node for later code generation.

If when searching for an instruction after a branch node the compiler encounters a merge node, the compiler attempts to locate a candidate instruction from after the merge node. If it finds one, then the compiler duplicates the control flow graph between the merge and the located node, delaying the merge until after the located node. Then the normal mechanism of splicing out the nodes between the branch and the target node applies; the branch target is automatically adjusted to jump to the instruction after the original merge point as part of the splicing operation. This "splitting" of merges when filling delay slots is important to fill as many delay slots as possible. No additional code space is needed for this kind of splitting, since at most one real instruction is copied and inserted into a delay slot that would be wasted otherwise.

Filling delay slots is important for good performance. According to the results shown in section 14.3, delay slot filling improves performance by more than 10% for many benchmarks, and reduces compiled code space consumption by more than 10% on average.

Other code generators for RISC machines perform additional kinds of scheduling of instructions, such as reordering memory loads to minimize stalls caused by waiting for the result of a load. The current SELF compiler currently does no such instruction scheduling.

12.5 Code Generation

The final pass of the compiler traverses the control flow graph and generates native machine code. Instruction selection is nearly trivial in the SELF compiler, since the compiler is designed primarily for modern RISC machines with simple, regular instruction sets. No peephole-style optimizations are performed, since (in most cases) none are needed. The compiler also generates debugging information (described in section 13.1) in this pass. Once generation is complete, the compiler passes the buffers used to hold the instructions and debugging information to the compiled code cache manager, which creates a new compiled method and adds it to the cache (possibly throwing out other old methods or compacting the cache to make room for the new method). The compiler then returns the starting address of the compiled code to the run-time message lookup routine which invoked the compiler. This lookup routine then completes the message send by jumping into the newly generated machine code.

12.6 Portability Issues

The SELF compiler has not been designed specifically to be retargetable to a new machine architecture without programming changes, but it has been designed and implemented to make porting the compiler to a related architecture relatively straightforward. For example, the compiler describes the set of registers on the target machine and other register- and stack-related calling conventions through a table of compile-time constants that can be easily changed for an architecture with a similar system of registers. Other parts of the compiler use a minimal number of **`#ifdef`**'s to isolate machine dependencies. Most control flow graph nodes are intended to map to a single target machine instruction (to allow as much low-level optimization such as delay slot filling as possible), and so there is some dependence on the kind of instructions provided by the target architecture in the kinds of control flow graph nodes. This approach works best for RISC-style architectures (the SELF compiler's primary intended target) and less well for CISC or other machine styles; some sort of peephole optimizer or generalized instruction selector would be needed to support these other kinds of architectures effectively. The largest machine dependency in the compiler is on instruction formats; this dependency is isolated in a per-machine subsystem of the compiler that is responsible for producing machine-specific instruction formats for higher-level control flow graph nodes such as **`add`** and **`branch`**. Some control flow graph nodes are only used for some architectures (such as the **`sethi`** control flow graph node on the SPARC), and some phases such as delay slot filling are executed only on architectures that need them. The difference between machines with three-operand instructions and machines with only two-operand instructions is handled in the register allocator by constraining the destination register to be the same as one of the sources for two-operand machines.

We expect that the SELF compiler could be ported to a new RISC architecture and generate reasonably good code in a week; other parts of the SELF implementation such as the memory system or some of the low-level assembly code support might take longer to port. CISC machines could also be supported in the same amount of time, but without a peephole optimizer or instruction selector run-time performance may not be as good as other compilers for the same machine. Just like other compilers, the SELF compiler could benefit from current work in table-driven compiler compilers to improve portability.

12.7 Summary

After the type analysis/inlining/splitting phase, the compiler executes several additional phases on the way to generating machine code:

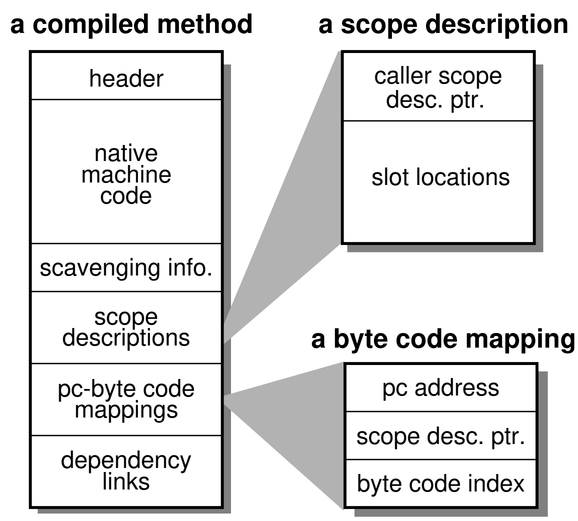
- The compiler eliminates redundant loads of constants. This currently does not interact well with register allocation and variable assignment, and so much of the benefit is wasted in unnecessary register traffic.
- The compiler eliminates unneeded computations such as arithmetic and memory loads.
- The compiler performs live variable analysis, allocates registers to variables, and inserts any needed register moves. The compiler uses variables to represent sets of aliased names in response to the heavy use of inlining in the SELF compiler. The register allocator should be extended to avoid inserting register moves for adjacent variables and to support finer grained position-dependent allocation.
- The compiler fills delay slots of branches and calls. This could be extended to schedule load delays as well.
- The compiler generates native machine instructions and debugging information, and adds the new compiled method to the compiled method cache.

Porting the SELF compiler to a new architecture would likely be neither overly difficult nor trivial.

Chapter 13 Programming Environment Support

The SELF system is designed to be an interactive exploratory programming environment, and so the SELF implementation must support both rapid turn-around for programming changes and complete source-level debugging. These features are fairly easy to support in an interpretive environment but are much more difficult to achieve in a high-performance optimizing compiler environment, particularly one based on aggressive inlining. Other researchers have investigated the problem of enabling compilation and optimization to coexist gracefully with the programming environment [Hen82, Zel84, CMR88, ZJ91]. This chapter describes the techniques used in the SELF implementation to support the programming environment, focusing on the support provided by the compiler.

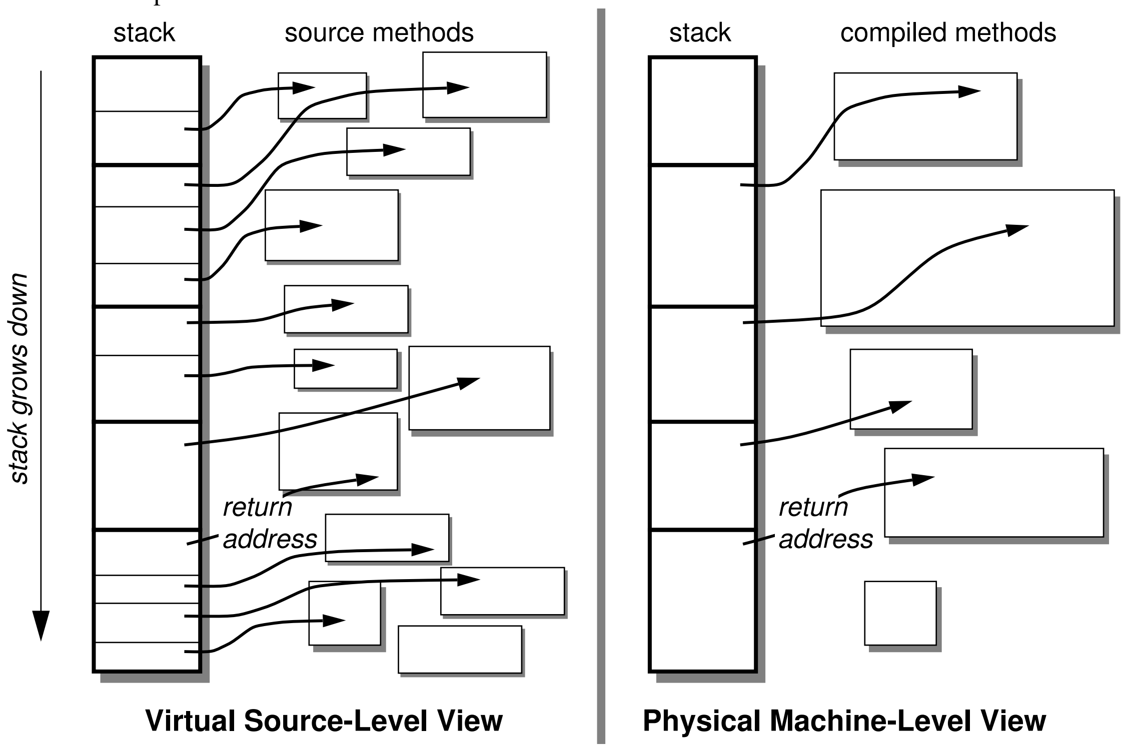
A compiled method contains more than just instructions. First, it includes a list of the offsets within the instructions of embedded object references, used by the garbage collector to modify the compiled code if a referenced object is moved. Second, a compiled method includes descriptions of the inlined methods, which are used to find the values of local slots of the method and to display source-level call stacks. Third, a compiled method contains a bidirectional mapping between source-level byte codes and actual program counter values. These two kinds of debugger-related information are described in section 13.1. Finally, a compiled method includes dependency links to support selective invalidation of methods after programming changes; these are described in detail in section 13.2.



Parts of a Compiled Method

13.1 Support for Source-Level Debugging

A good programming environment must include a source-level debugger. The SELF debugger presents the program execution state in terms of the programmer's execution model: the state of the source code interpreter, with *no* optimizations. This requires that the debugger be able to examine the state of the compiled, optimized SELF program and construct a view of that state (the *virtual* state) in terms of the byte-coded execution model. Examining the execution state is complicated by having activation records in the *virtual call stack* actually be inlined within other activation records in the *physical call stack*, and by allocating the slots of virtual methods to registers and/or stack locations in the compiled methods.



13.1.1 Compiler-Generated Debugging Information

To allow the debugger to reconstruct the virtual call stack from the physical call stack, the SELF compiler appends debugging information to each compiled method.

- For each scope compiled (the initial method plus any methods or block methods inlined within it), the compiler outputs information describing that scope's place in the virtual call tree within the compiled method's single physical stack frame.
- For each argument and local slot in the scope, the compiler outputs either the value of the slot (if it is a constant known at compile-time, as many slots are) or the register or stack location allocated to hold the value of the slot at run-time.
- For each subexpression within the compiled method, the compiler describes either the compile-time constant value of the subexpression or the register or stack location allocated for the subexpression. This information is used to reconstruct the stack of evaluated expressions that are waiting to be consumed by later message sends.

For example, consider a simple method to compute the minimum of two values:

```
min: arg = (
  < arg ifTrue: [self] False: [arg] ).
```

If **`min:`** is sent to an integer, the compiler will generate a compiled version of the **`min:`** source method customized for integers. (Customization was the subject of Chapter 8.) The **<** method for integers will be looked up at compile-time, locating the following method:

```
< x = ( _IntLT: x IfFail: [...] ).
```

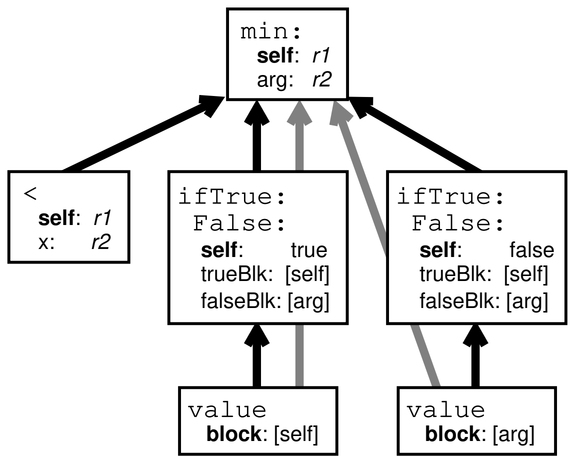
This method and the contained call to the **`intLT`** primitive will be inlined. The result of the **`<`** message will be either the **`true`** object or the **`false`** object (in the common case), leading the compiler to split the succeeding **`ifTrue:False:`** message along these two possible outcomes. (Splitting was the subject of Chapter 10.). The compiler looks up the definition of **`ifTrue:False:`** for **`true`**:

```
ifTrue: trueBlk False: falseBlk = ( trueBlk value ).
```

and for **`false`**:

```
ifTrue: trueBlk False: falseBlk = ( falseBlk value ).
```

These two methods will be inlined, as will the nested **`value`** messages within each method. When generating code for this compiler method, the compiler outputs debugging information to represent this tree of inlined methods:



Each scope description refers to its calling scope description (black arrows in the diagram); a block scope also references its lexically-enclosing scope description (gray arrows in the diagram). For each slot within a scope, the debugging information identifies either the slot's compile-time value or its run-time location. For the **`min:`** example, only the initial arguments have run-time locations (registers **`r1`** and **`r2`** in this case); all other slot contents are known statically at compile-time. The expression stack debugging information is omitted from this illustration.

This additional debugging information is fairly space-consuming. As described in section B.6 of Appendix B, scope descriptions take up between 1.5 and 5 times as much space as do the compiled machine instructions, depending on the degree of inlining performed when compiling the routine. Fortunately, this information can be paged out when the associated routine is not being debugged. Also, it might be possible to avoid storing the debugging information, instead regenerating the debugging information upon demand by re-executing the compiler.

13.1.2 Virtual/Physical Program Counter Translation

The SELF compiler also outputs debugging information to support translation between the source-level *virtual program counter* within a virtual stack frame (the pair of a scope description and a byte code index within the scope description) and the machine-level *physical program counter* within a physical stack frame. This information is used to translate a hardware physical return address of a stack frame into a byte code index within a virtual stack frame of the physical frame, such as when displaying the current virtual execution stack. The mapping also is used to locate the physical program counter corresponding to a particular virtual program counter, such as when setting breakpoints at particular source positions.

The p.c./byte code mapping is not one-to-one but instead is many-to-many. Several virtual program counter addresses can map to the same physical program counter address: a sequence of source-level messages can get inlined and optimized away completely (such as most of the messages sent during the execution of a user-defined control structure), generating no machine code for any of the eliminated messages, and so each of the messages will end up mapping to the same physical machine address. Additionally, several physical program counter addresses may map to the same virtual program counter address: a single source-level message can get split and compiled in more than one place, thus leading several physical program counter addresses to map to the same source-level message. Consequently, the compiler treats this mapping as a simple relation and generates a long list of three-word tuples, each tuple consisting of a physical program counter address (the physical view) and the pair of a pointer to a virtual scope description and a byte code index within the scope (the virtual source-level view).

As reported in section B.6 of Appendix B, the p.c./byte code mapping is fairly concise, only requiring space that is about 25% of that taken up by compiled instructions. One reason for its comparatively small size is that the compiler only generates tuples that correspond to call sites in the compiled code, since those are the only places that the system might suspend the method and examine its p.c./byte code mapping.

13.1.3 Current Debugging Primitives

The current SELF implementation includes partial support for interactive debugging. The system supports displaying the virtual execution call stack, complete with the current values of all local variables of all virtual activation records; all optimizations including inlining are completely invisible. The system also supports manipulating individual activation records directly as SELF objects, querying the contents of their local slots, examining their expression stack, and navigating around the dynamic and static call chains. The current implementation does not yet support modifying the contents of local variables in activation records, but we do not think that adding this facility would be too difficult. The system supports breakpoints and single-stepping through process control primitives. The programmer can set a breakpoint by editing in a call to user-defined SELF code which eventually invokes the process suspend primitive. A suspended process can be single-stepped by invoking other process control primitives.*

13.1.4 Interactions between Debugging and Optimization

Most optimizing compilers do not support complete source-level debugging because the optimizations they perform prevent the virtual source-level state from being completely reconstructed. For example, tail call optimizations prevent the programmer from examining the elided stack frames, and dead variable elimination and dead assignment elimination prevent the programmer from examining the contents of a variable that is in scope but no longer needed by the compiled code. The SELF compiler performs no optimization that would prevent the debugger from completely reconstructing the virtual source-level execution state as if no optimizations had been performed. Even so, the SELF compiler still can perform many effective optimizations including inlining, splitting, and common subexpression elimination, since these optimizations can be “undone” at debug-time given the appropriate debugging information.

The SELF compiler’s job of balancing debugging support against various optimizations is eased by only requiring debugger support at those places in which a debugging primitive might be invoked, such as message sends and **`_Restart`** loop tails.** The compiler is not required to support debugging at arbitrary instruction boundaries (as would be required if an interrupt could occur at any point in the program) or even at source-level byte code boundaries (as would be required if the user could single-step through optimized code; single stepping is implemented by recompiling methods with no optimization and then stopping at every call site). Since the debugger can be invoked only at well-defined locations in the compiled code, the compiler can perform optimizations between these potential interruption points that would be difficult or impossible to perform if instruction-level or byte-code-level debugging information were required. For example, the compiler can reuse the register of a dead variable as long as there are no subsequent call sites or interruption points in which the variable is still in scope.

Unfortunately, the current representation of debugging information places restrictions on the compiler that can hurt performance. As mentioned in section 12.1, the current SELF register allocator either can allocate a particular name to a single location for its entire lifetime or can mark the name as bound to a particular compile-time constant for its entire lifetime. The restriction that the allocation be constant over the name’s entire lifetime primarily is caused by the limited abilities of the debugging information to describe the allocation; there is no easy way to have different allocations for different subranges of the name’s lifetime. The system might be able to get added flexibility within the constraints of the current representation by making copies of virtual scope descriptions whenever a name had different allocations for different parts of its lifetime, and using the physical/virtual program counter mapping to select the appropriate virtual scope description for the physical program counter. This approach would support some form of position-varying allocation, but could lead to a lot of duplicated debugging information. A better approach would be to redesign the debugging information representation from scratch to efficiently support names with position-varying allocations.

* Urs Hölzle implemented most of the process control and activation record manipulating primitives, and Bay-Wei Chang integrated these primitives into the graphical user interface.

** The debugger might run at **`_Restart`** points since these points check for interrupts (as described in section 6.3.1), and the user-defined interrupt handler might call the debugger.

13.2 Support for Programming Changes

As described in section 7.1, the SELF compiler assumes that certain hard-to-change parts of objects will remain constant, and the compiler performs optimizations based on these assumptions. For example, the compiler assumes that the set of slots of a particular object, such as an integer or the **`true`** object, will remain the same, and this allows the compiler to perform message lookup at compile-time. Similarly, the compiler assumes that the contents of a non-assignable data slot will never change and that the offset of an assignable data slot will never change. These assumptions enable the compiler to inline the bodies of methods and replace data slot access methods with load and store instructions, thereby generating much faster code.

These assumptions will always be correct if the only object mutations available to programs and programmers are normal assignments to assignable data slots (the compiler explicitly avoids depending on the contents of an assignable slot). However, additional operations are available in a programming environment to mutate objects in other ways, such as adding and removing slots or changing the contents of non-assignable data slots. These modifications may invalidate the assumptions made by the compiler when compiling and optimizing methods. If executed, such out-of-date compiled code can lead to incorrect behavior or even system crashes.

13.2.1 Ways of Supporting Programming Changes

Traditional batch compiling systems support programming changes by requiring the programmer to recompile manually those files that are out-of-date, relink the program, and restart the application. At best, some of this process can be automated by using utilities to determine which files need to be recompiled after a set of programming changes. Turn-around time for a single programming change can be quite long, at least tens of seconds and more typically minutes or tens of minutes. Programmer productivity suffers greatly with turn-around times of this length for simple programming changes.

Interactive systems are designed to support rapid programming turn-around times, on the order of a few seconds or less. They usually achieve this level of interactive performance by limiting the dependencies among components of a system, so individual components can be replaced as simply and easily as possible without requiring complex time-consuming system relinking or recompilation of other components not directly altered by a programming change. However, execution performance tends to be much lower than in the traditional optimizing environment, since inter-component information is not used for optimizations. The SELF compiler clearly violates the basic assumptions of this style of system, since the SELF compiler delights in performing optimizations such as inlining that create many inter-component dependencies.

SELF's run-time compilation architecture offers one possible solution to this dilemma. After a programming change that might have invalidated the assumptions used to compile some method, the system could simply *flush* all the compiled code from the compiled code cache. New code will be compiled as needed from the (possibly changed) source methods, using the new correct assumptions about the relatively unchanging parts of the object structure. Since compiled code cache flushing is fast, this would seem to solve the programming turn-around time problem.

Unfortunately, this approach simply shifts the cost of the programming change from the flushing operation to immediately after the flushing. Since after the flush no methods are compiled, nearly every message send will require new compiled code to be generated, leading to a long sequence of compiler pauses immediately after the flush. While these compiler pauses will be spread out somewhat over the next chunk of program execution, and SELF's compiler architecture will allow code to be generated and "relinked" much faster than a traditional file-based environment, turn-around time usually will still be much longer than the second or two supported by the unoptimizing interactive system.

To avoid these lengthy recompilation pauses, the SELF system maintains enough inter-component dependency information to *selectively invalidate* only those compiled methods that are affected by a programming change. If this set is small, then the number of pauses to recompile invalidated methods can be kept small and the overall perceived turn-around time can be kept short. In these situations, selective invalidation enables our SELF system to support both fast turn-around on programming changes and fast run-time execution between changes.

Selective invalidation is not a cure-all, however. If some extremely common method that is inlined in many places is changed, such as the definition of **+** for integers, then the selective invalidation approach reduces to the expensive total flush approach, producing the same lengthy compilation pauses as the whole system gets recompiled with the new definition. Fortunately, this has not been a problem in practice, since the common methods used throughout the system are changed very rarely; in other systems such sweeping changes could not be made at all. The current trade-off between run-time performance and programming turn-around time favors run-time performance over turn-around time for these kinds of short, commonly-used “system” methods.

13.2.2 Dependency Links

To support selective invalidation, the compiler maintains two-way *change dependency links* between each compiled method in the cache and the information that the compiler assumed would remain constant. This information used to compile code—the set of slots in objects, the offsets of assignable data slots, and the contents of non-assignable slots—is precisely the information stored in maps. (Maps were described in section 6.1.1 and in Appendix A.) Therefore, the system only needs to maintain dependency links between maps and compiled methods. Since many compiled methods may depend on a particular map, and each compiled method may depend on many maps, these dependency links must support a many-to-many mapping between maps and compiled methods.

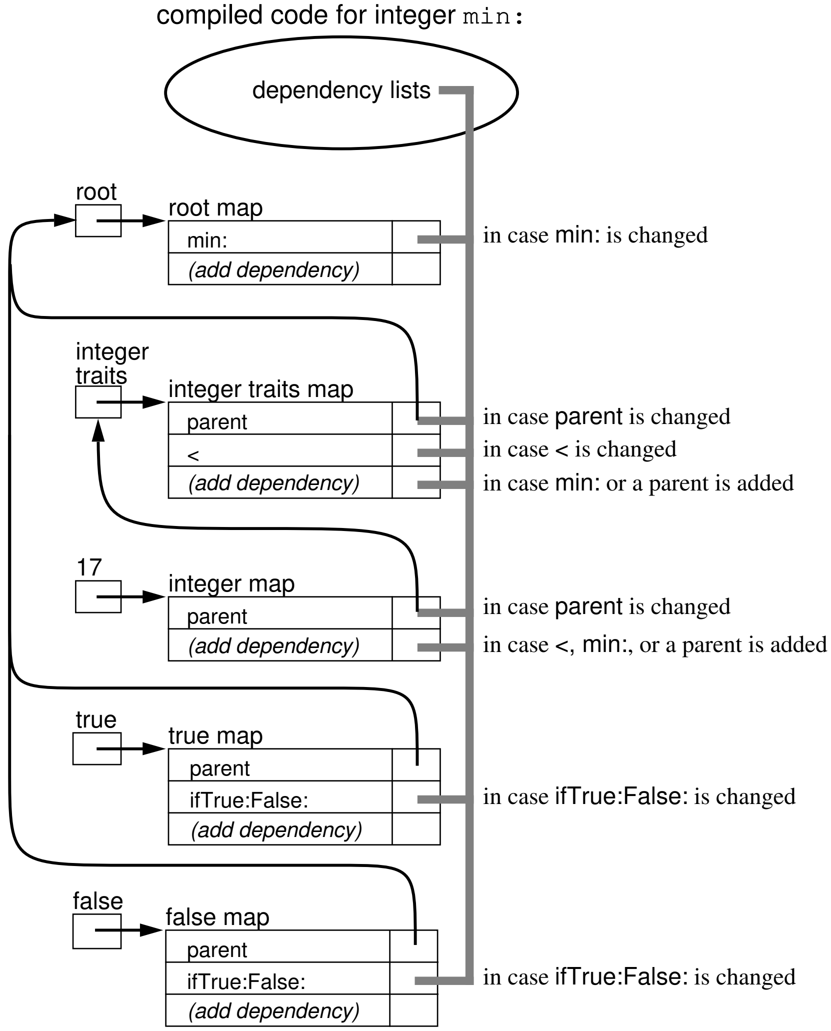
Dependency links are created as a result of message lookup to record those aspects of objects traversed during lookup that if they were modified potentially could change the result of the lookup and consequently the correctness of the compiled code. Clearly the compiled code depends on the result of the message lookup, and so the system leaves behind a dependency link between the matching slot found at the end of the lookup and the method being compiled. If the matching slot is later changed (either by changing the contents of a constant slot such as a method or parent slot or by changing the offset in the object of the contents of an assignable data slot) or removed altogether, all linked compiled methods are flushed from the compiled code cache.

The lookup system scans other parts of objects which affect the outcome of the lookup and so need dependencies. The message lookup system fetches the contents of a parent slot when searching an object’s parents. If the parent slot is later changed or removed, the outcome of the message send could change. To record this fact, the system creates a dependency link between the parent slot and the compiled code for the method eventually found as the result of the lookup. Then if the parent slot is modified or removed, all linked compiled methods will be flushed appropriately.

A more subtle kind of link handles the problem that a slot may be added to an object that affects the outcome of a message send. The message lookup system frequently searches an object for a matching slot and is unsuccessful; the object’s parents are searched in turn for a matching slot. If either a matching slot or a parent slot that inherits a matching slot is later added to the object, then the results of the earlier message would likely be changed, possibly invalidating some compiled code. To handle this problem, the compiler creates a special *add dependency link* between compiled code and the maps of objects unsuccessfully searched for a particular slot; this dependency is not associated with any slot in the map but instead with the map as a whole. If a slot is ever added to the map, then all compiled methods linked by the add dependency are flushed, since the added slot might affect their lookup results.

Unlike slot-specific dependency links, add dependency links are imprecise. Since they do not record exactly which message names were unsuccessfully scanned previously, methods may be flushed that do not need to be flushed. This could significantly reduce the selectiveness of the flushing, possibly leading to long compile pauses after a programming change in which slots were added. On the other hand, recording exactly which message names have been searched unsuccessfully for each map would consume a lot of space, and many maps would have similar long lists of unsuccessful matches. We are currently exploring alternative mechanisms that would support selective invalidation even for slot additions.

The following diagram illustrates the dependency links that are created when compiling the **`min:`** method described earlier in this chapter. The gray line represents eight separate dependency links, each link connecting a slot in a map (or the map as a whole in the case of the add dependency links) to the compiled code for **`min:`**.



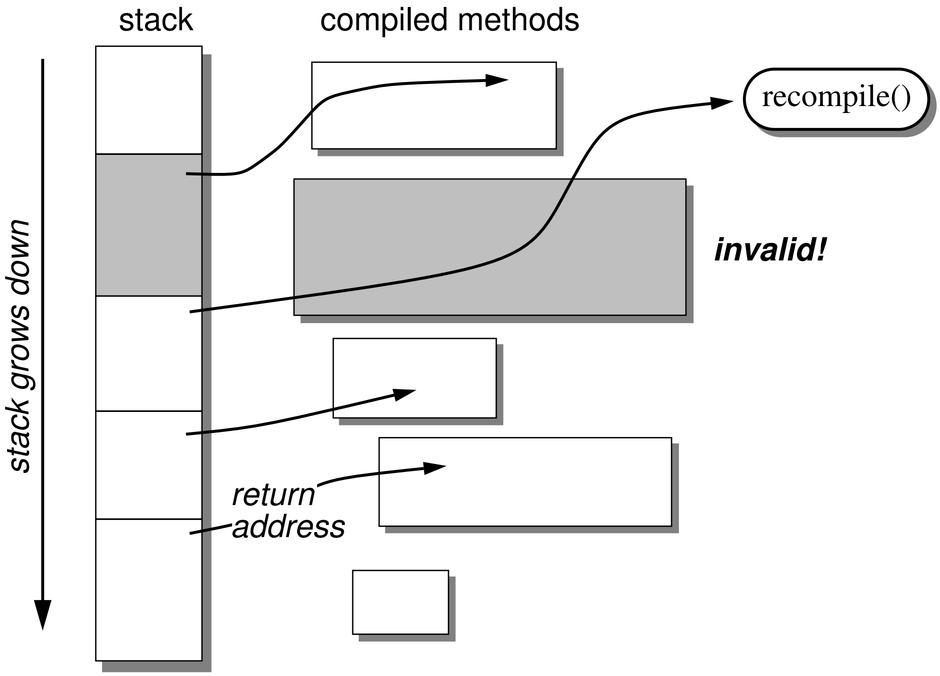
13.2.3 Invalidation

After a programming change, the compiler traverses dependency links to invalidate compiled methods linked to the updated information. Invalidation is normally quite straightforward, simply requiring the invalid compiled method to be thrown out of the compiled code cache. However, if a compiled method is currently running (i.e., if there is a stack frame suspended within the compiled method), then this invalidation becomes complicated. These compiled methods cannot just be flushed, because they are still executing and will be returned to. Nor can they remain untouched, since they have been optimized based on information that is no longer correct. The approach taken in the SELF system is to recompile the out-of-date compiled method and rebuild its stack frame based on the data stored in the old stack frame.*

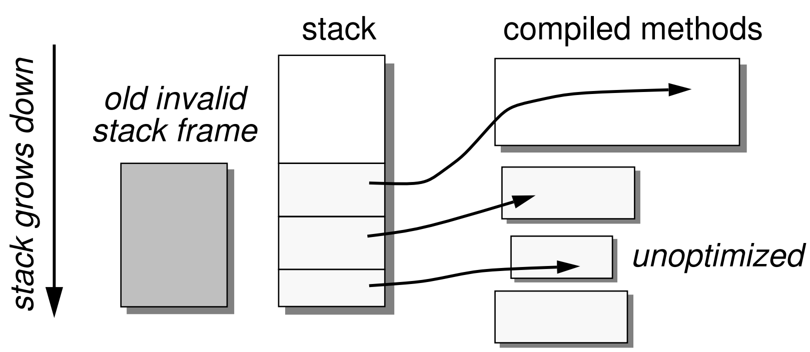
The SELF system performs this conversion *lazily* to make this recompilation easier and less intrusive [HCU92]. When a compiled method suspended on the stack is first invalidated as part of the execution of some programming primitive, the system marks the method as invalid and removes it from the lookup cache (so that future message sends will not be bound to the invalid compiled method), but the system does not yet flush the compiled method from the compiled code cache. Instead the system adjusts the return address of the stack frame that would have returned to the invalid

* The data in the stack frame is still valid. Only the compiled code of the invalidated method is suspect.

method's stack frame to instead "return" to a special support routine in the run-time system. The system then returns control to the programming primitive which will eventually return to the SELF process that invoked it.



Eventually the stack frame below the one for the invalid compiled method will "return," calling the special run-time support routine. This routine recompiles the invalid compiled method and builds new stack frames that represent the same abstract state as did the invalidated compiled method's stack frame, which because of the lazy recompilation is now on the bottom of the stack of SELF activation records. To make it easy to fill in the state for the new stack frame and to keep recompilation pauses short, the new method is compiled without optimization. Since the invalidated compiled method was probably compiled with optimization, including inlining, the system may need to compile several unoptimized methods to represent the same abstract state as the invalidated compiled method, one unoptimized method and physical stack frame for each virtual stack frame inlined into the single physical stack frame of the invalidated compiled method at the point of call.



To complete the conversion process, the recompiling routine returns into the appropriate point in the new compiled method for the topmost stack frame. The invalidated compiled method can be flushed from the compiled code cache if the old invalid stack frame is the last activation of this method.*

Lazy conversion spreads the load of recompilation out across a longer period of time, reducing the perceived pauses after a programming change. If several programming changes occur before returning to an invalid method, then less overall work may be performed since the method will not be recompiled after each programming change. Lazy

* Urs Hölzle implemented the mechanisms to lazily recompile invalid methods on the stack.

conversion also simplifies and speeds the conversion process by limiting recompilation and stack frame creation to the top of the stack. This eliminates the need to copy whole stacks and adjust interior addresses when recompiling and rebuilding some stack frame buried in the middle of the stack.

13.3 Summary

The SELF compiler is designed to coexist with an interactive exploratory programming environment. This kind of environment requires complete source-level debugging to be available at all times and “down time” caused by programming changes to be limited to a few seconds at most. The SELF compiler supports complete source-level debugging in the face of optimizations such as inlining and splitting by generating additional information that allows the debugger to view a single physical stack frame as several source-level virtual stack frames. Fast turn-around time for programming changes is supported by a selective invalidation mechanism based on dependency links that flushes out-of-date compiled methods from the compiled code cache. This invalidation is performed lazily for compiled methods currently executing on the stack.

Chapter 14 Performance Evaluation

We illustrated the new techniques described in this dissertation on small examples in previous chapters, and in that context they appeared to be quite effective. In this chapter we explore the effectiveness of our implementation of these techniques on actual SELF programs by measuring the execution speed, compilation time, and compiled code space consumption of a suite of SELF programs ranging in size from a few lines to a thousand lines. We will evaluate our work from three perspectives:

- How effectively does the combination of these new techniques narrow the gap in performance between pure object-oriented languages such as SELF and traditional languages such as optimized C or optimized Lisp?
- Which of the new techniques are most effective? Which are most costly?
- What are promising areas for future work?

Our answers to these questions are presented in sections 14.2, 14.3, and 14.4, respectively. The next section prefaces these results with a description of our measurement methodology.

14.1 Methodology

14.1.1 The Benchmarks

We analyzed the performance of the SELF implementation using a selection of benchmark programs. These benchmarks include several "micro-benchmarks" gathered from various sources, the Stanford integer benchmark suite [Hen88], the Richards benchmark [Deu88], and several SELF programs originally written without benchmarking in mind. Source code for all these benchmarks is available from the author upon request.

The eleven micro-benchmarks are very short and typically stress only a few aspects of the implementation, such as the speed of integer calculations or the speed of procedure calls. The Stanford integer benchmark suite is composed of the **`perm`**, **`towers`**, **`queens`**, **`intmm`**, **`quick`**, **`bubble`**, **`tree`**, and **`puzzle`** benchmarks, originally collected by John Hennessy and Peter Nye to help design and measure RISC microprocessors and compilers. The Stanford integer benchmarks are larger than the micro-benchmarks, and all exercise integer calculations, generic arithmetic, and user-defined control structures (particularly **`for`**-style loops); all but **`tree`** also stress array accessing. For SELF and Smalltalk, all the Stanford integer benchmarks other than **`puzzle`** were written in two versions: one as similar to the C version as possible, and one taking advantage of the message passing features of SELF and Smalltalk by associating the core of the benchmark code with the data structures manipulated by the code. Both versions of each benchmark perform the same algorithm, with no source-level optimizations made in either version, but the object-oriented version sends more messages to **`self`** and fewer messages to other objects than the procedure-oriented version. Results for the versions of the benchmarks in the more object-oriented style are identified by the **`oo-`** prefix in the benchmark names. Section B.1 of Appendix B describes the individual micro-benchmarks and Stanford integer benchmarks in more detail.

None of the micro-benchmarks or Stanford integer benchmarks is particularly object-oriented. This is to be expected, since we translated them into SELF from traditional non-object-oriented languages such as C. This lack of object-orientation implies that any conclusions that may be drawn from measurements of these benchmarks must be limited to the effectiveness of the SELF implementation at running smaller, more traditional programs. Any conclusions about the performance of SELF on more object-oriented programs based on the performance of these benchmarks must be more tentative and speculative. However, most object-oriented programs contain more traditional portions that must scan through an array or perform simple arithmetic calculations; these small non-object-oriented sections in fact may comprise much of the running time of an application. Therefore, an improvement in performance for non-object-oriented code is likely to improve overall system performance.

The **`richards`** benchmark is a 400-line operating system simulation benchmark originally written by Martin Richards to test BCPL compilers. This benchmark maintains a queue of tasks and spends its time primarily removing the first task from the queue, processing it, and usually appending it to the end of a different queue for further processing; the benchmark begins by initializing the queue with a few tasks and ends when the main task queue is empty. The **`richards`** benchmark is different from the previous ones in that it is large enough not to overly stress any particular operations and has no tight loops or recursions. Instead the **`richards`** benchmark manipulates data structures and therefore is perhaps more representative of typical object-oriented programs than the previous benchmarks. The **richards** benchmark is also unusual in including a message send that actually invokes different methods for different calls: the **runTask** message which is sent to each task after it is removed from the head of the queue and is defined differently for different kinds of tasks.

To measure the effectiveness of the new techniques on truly object-oriented programs, we also measured several SELF programs originally written to be useful in their own right and not as benchmarks. The **pathCache** and **primMaker** programs are in active use today.

- **pathCache** computes a mapping from objects to names inferred from the object structure, used as part of the SELF user interface. **pathCache** is 140 lines long, excluding the code for other supporting data structures such as dictionaries (SELF's key-value mapping data structure).
- **primMaker** generates SELF and C++ wrapper functions for user-defined primitives from a textual description of the primitives. The benchmark was run on a test file that exercises all the different possible descriptions and so ends up being dominated by compile time. **primMaker** is 1000 lines long.
- **parser** parses an earlier version of SELF. The benchmark was run on four relatively short expressions. **parser** is 400 lines long.

These benchmarks use features of SELF such as prototype-based design and dynamic inheritance that are not available in any of the other languages, and so versions of these benchmarks exist only for SELF. These benchmarks cannot be used to compare the performance of SELF to other languages and systems, but they can be used to measure the effectiveness of the various techniques developed for SELF by comparing one configuration of the SELF system against another.

The following table summarizes the benchmarks we measured:

small	eleven micro-benchmarks (1 to 10 lines long)
stanford	seven Stanford integer benchmarks, written in a traditional procedure-oriented style (20 to 60 lines long)
oo-stanford	seven Stanford integer benchmarks, written in a more object-oriented style (different from **stanford** only for SELF and Smalltalk)
puzzle	the largest Stanford integer benchmark, written in a traditional procedure-oriented style (170 lines long)
richards	a medium-sized operating system simulation benchmark (400 lines long)
pathCache	a short SELF program in frequent use today (140 lines long)
primMaker	a larger SELF program in occasional use today (1000 lines long)
parser	a medium-sized SELF program originally written as a programming exercise (400 lines long)

Section B.2 of Appendix B describes our measurement procedures in detail. Appendix C contains the raw data for all the measurements.

14.1.2 The Hardware

We did all our measurements on a Sun-4/260 workstation configured with 48MB of main memory. The Sun-4/260 workstation is based on the SPARC, a RISC-style microprocessor with hardware register windows and delayed branches and calls [HP90]. The implementation of the SPARC on the Sun-4/260 has a 62ns cycle time, 8 register windows, two-cycle loads (if the target register is not used in the following instruction, three cycles otherwise), and three-cycle stores, assuming cache hits. The SPARC also provides limited hardware support for tagged arithmetic, although none of the systems measured, including the SELF system, currently exploit this hardware.

14.1.3 Charting Technique

In all charts, bigger bars are better in the sense that they correspond to a more efficient implementation. Execution and compilation performance is reported in terms of speed, with taller bars corresponding to faster systems. Compiled code space costs are reported in terms of density (the inverse of space usage), with taller bars corresponding to more space-efficient systems (systems requiring less space for compiled code). Compiled code space numbers include only space for machine instructions and exclude space costs for debugging information and the like. Section B.6 of Appendix B reports measurements of these additional space costs for the SELF system.

14.2 Performance versus Other Languages

To determine the overall effectiveness of the new techniques described in this dissertation, we compared the performance of the SELF implementation to implementations of several other languages.

- We want to compare our SELF implementation to that of a traditional optimized language implementation. This comparison will tell us how well our techniques do in narrowing the gap in performance that previously existed between pure object-oriented languages and traditional optimized languages. Also, the performance of a traditional optimized language places something of an upper bound on the performance we can reasonably expect from our SELF implementation. We measured the optimizing C compiler provided standard with SunOS 4.0.3 UNIX, invoked with the **`-O2`** flag. The C version of the **`richards`** benchmark really is written in C++ (version 1.2 of AT&T's **`cfront`** C++-to-C translator) so that the **`runTask`** message can be implemented by a C++ virtual function call.
- We also want to compare our SELF implementation to the performance of the best existing implementation of a pure object-oriented language, to compare the effectiveness of our techniques to previous techniques for implementing pure object-oriented languages. We measured the performance of ParcPlace Smalltalk-80, version V2.4 β2, the fastest commercial implementation of Smalltalk. ParcPlace's Smalltalk-80 implementation includes the Deutsch-Schiffman techniques for constructing a fast Smalltalk implementation, described in section 3.1.2. Unfortunately, only execution speed results are available for Smalltalk; neither compilation speed nor compiled code space usage could be measured.
- Finally, we also are interested in the relative performance of our SELF system and some sort of Lisp-based system. Lisps provide many of the same features as does SELF, such as support for generic arithmetic, closures, and dynamic type-checking. Unlike SELF, however, Lisps also provide direct procedure calls, direct variable accesses, and built-in control structures. Comparing SELF to a Lisp system can help determine how well SELF handles generic arithmetic compared to existing techniques in Lisp systems and how well SELF optimizes away the extra overhead of message passing and completely user-defined control structures. We measured the ORBIT compiler (version 3.1) for T, a dialect of Scheme, described in section 3.3.2. This system is widely regarded as a high-quality Lisp implementation that supposedly competes well against traditional language implementations.

  Many Lisps include special low-level primitive operations that avoid the overhead of the corresponding general, safe primitive operations. For example, T includes the **`fx+`** operation which assumes its arguments are *fixnums* (Lisp jargon for fixed-precision integers that fit in a machine word, equivalent to **`SmallInteger`** objects in Smalltalk and **`int`** in C). **`fx+`** is more efficient than the generic **`+`** operation since **`fx+`** assumes its arguments are fixnums and can be compiled down to a single machine **`add`** instruction, much like the **`+`** operator in C. However, **`fx+`** does *not* verify the assumption that its arguments are in fact fixnums before it adds them, nor does it check for overflows after the addition; consequently, **`fx+`** is unsafe. This distinction between slow, safe operations and fast, unsafe operations forces programmers to choose explicitly where to use normal generic arithmetic (and be willing to pay the cost of the better semantics) and where to use the faster **`fx+`**-style arithmetic (and be willing to take responsibility for the unverified assumptions).

  Lisps also typically include directives that allow the programmer to invoke additional optimizations such as inlining. T includes a **`define-integrable`** form that works just like **`define`** (i.e., binding a name to a value or function) except that the compiler will inline the bound value or function whenever the name is evaluated. In T, inlining is an optimization that must be invoked explicitly by programmers.

  We expect that most T programmers normally would use **`+`** and **`define`**. Sometimes, however, T programmers may feel the need to use unsafe operations and explicit directives to get better performance. In fact, nearly all benchmark performance results for Lisps are reported after such hand optimizations, with **`+`** replaced with **`fx+`** where possible and with **`define-integrable`** sprinkled in where desired. To capture these two styles of usage, we wrote *two* T versions of each benchmark: one as we would expect a normal programmer to have written the benchmark (and thus measuring the expected performance of T for the average programmer), and another hand-optimized version of the benchmark using **`fx+`** and **`define-integrable`** more like what a benchmarker would measure (and thus measuring the best possible performance of T as seen by the expert).

  The two versions of the **`richards`** benchmark use T's structure facility to implement task objects, and declare **`runTask`** as a T operation, with different methods invoked for different types of task objects.

The following table summarizes the language implementations we measured:

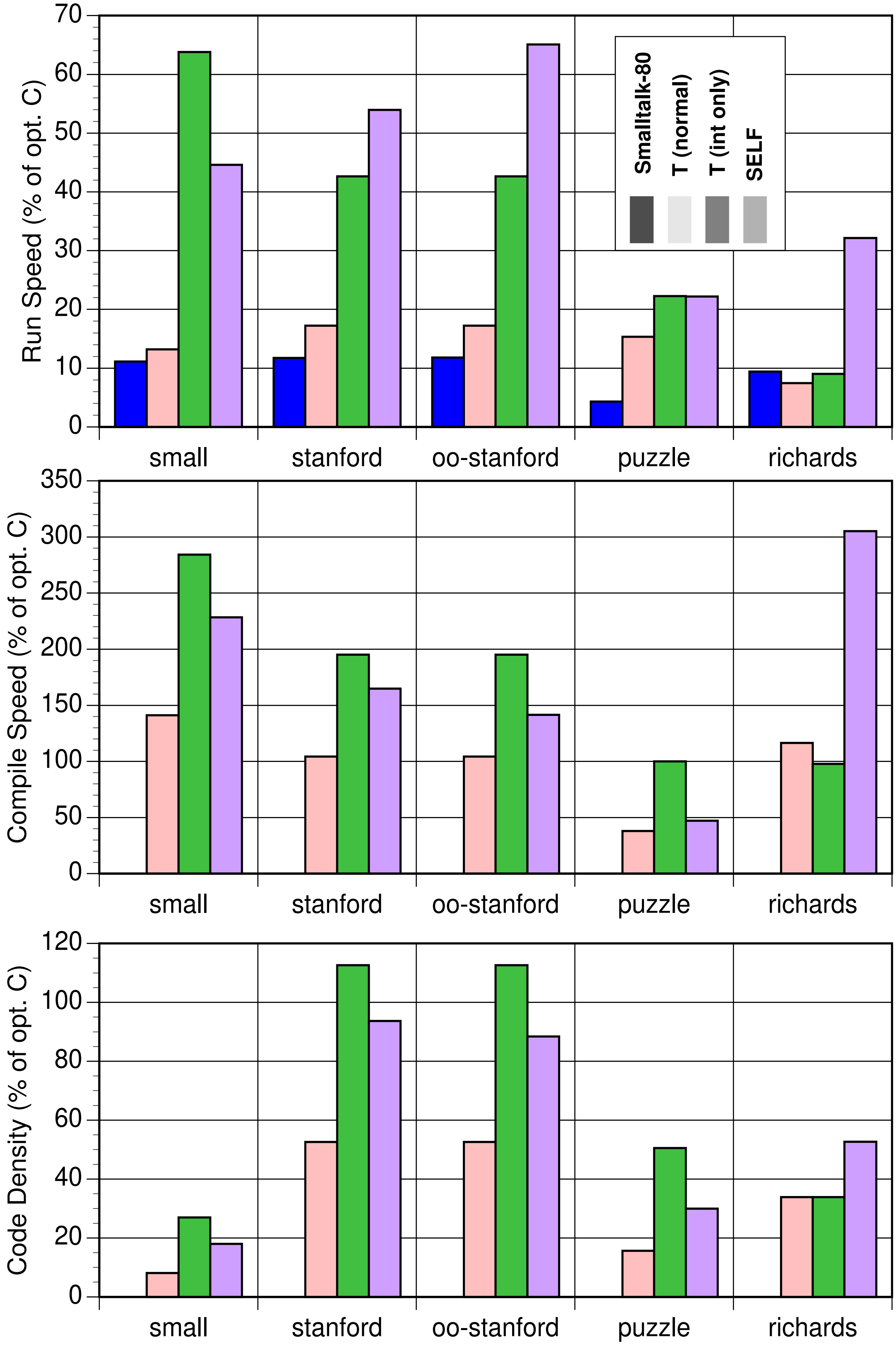
C	SunOS 4.0.3 standard optimizing C compiler (**`-O2`**)
Smalltalk-80	ParcPlace Systems Smalltalk-80 V2.4 β2, the fastest commercial Smalltalk implementation
T (normal)	ORBIT compiler for T (version 3.1), “normal” coding style
T (int only)	ORBIT compiler for T (version 3.1), integer-specific hand-optimized coding style
SELF	SELF implementation

The charts on the next page compare the execution speed, compilation speed, and compiled code space efficiency of these systems on the benchmarks (compilation speed and compiled code space efficiency results are unavailable for Smalltalk-80). All results are reported as a percentage of the results for optimized C.

These SELF benchmark suites and programs run between 22% and 65% of the speed of optimized C. This level of performance is about 5 times better than the performance of ParcPlace Smalltalk-80, despite the fact that SELF actually is harder to compile efficiently than Smalltalk (primarily because SELF accesses variables via messages while Smalltalk accesses them directly). Perhaps surprisingly, SELF runs between 1.4 and 4 times faster than T compiled by the ORBIT compiler, when running “normal” T programs, even though the T benchmarks use only direct procedure calls (excluding the one operation invocation site in **`richards`**) and only built-in control structures. SELF outperforms T in most cases even when comparing against fixnum-specific hand-optimized T programs. These comparisons provide evidence that the SELF compiler is doing an excellent job of eliminating the run-time overhead associated with message passing, user-defined control structures, and generic arithmetic.

The compilation speed of the SELF implementation is quite reasonable when compared to other optimizing language implementations. SELF’s compilation speed is better than optimized C’s for all cases except **`puzzle`**; SELF compiles **`richards`** 3 times faster than C++. SELF compiles faster than the ORBIT compiler on the “normal” T benchmarks; ORBIT usually compiles the integer-specific version of the T benchmarks faster than the SELF compiler. We could not measure the compilation speed for ParcPlace Smalltalk-80, but we believe that it is significantly faster than any of the other implementations, principally because the Smalltalk-80 compiler does virtually no optimization during compilation. Unfortunately, the performance of the SELF compiler is still too slow to go unnoticed by the SELF programmer, as was one of our original goals for the project. Speeding the compiler further without sacrificing execution performance continues to be an active area of research.

SELF is not as space-efficient as optimized C. SELF uses about 6 times as much space as C for the micro-benchmarks and about 3 times as much space for the **`puzzle`** benchmark. However, for the Stanford integer benchmarks, SELF incurs a space overhead of less than 20% compared to C, and for **`richards`** (the largest, most data-structure-oriented of these benchmarks), SELF consumes only twice as much space as optimized C++. These relatively low space overheads for the majority of the benchmarks are remarkable, considering that many of the SELF compiler’s techniques such as splitting trade away compiled code space to get faster execution times. Given the falling costs of memory, the SELF system’s extra space requirements seem quite reasonable. SELF uses less compiled code space than normal T programs compiled by the ORBIT compiler, and not much more space than integer-specific T programs. These results confirm the practicality of our compilation techniques in terms of space costs.


Smalltalk-80
T (normal)
T (int only)
SELF
Run Speed (% of opt. C)
0
10
20
30
40
50
60
70
small
stanford
oo-stanford
puzzle
richards
Compile Speed (% of opt. C)
0
50
100
150
200
250
300
350
small
stanford
oo-stanford
puzzle
richards
Code Density (% of opt. C)
0
20
40
60
80
100
120
small
stanford
oo-stanford
puzzle
richards

Language implementations typically trade off execution speed and compilation speed against one another: the compiler usually has to work harder to produce code that runs faster. The practicality of a particular language depends in large part on how well its implementation balances these two competing goals. The chart below summarizes the performance of the language implementations by scatter-plotting the execution and compilation speed results for each system, averaging together all the benchmarks. We assigned the Smalltalk-80 implementation a compilation speed 4 times faster than the optimizing C compiler. This figure is only a guess, intended to roughly illustrate the Smalltalk-80 implementation's position in the chart; the execution speed figure is accurate, however.

paste chart-page-164.ps here

SELF performs better in execution speed than either version of the T/ORBIT system, and compiles faster than ORBIT on the normal version of the T benchmarks. SELF runs these benchmarks at an average of 50% the speed of optimized C and with faster compilation speed than the optimizing C compiler.

14.3 Relative Effectiveness of the Techniques

Now that we understand the overall effect of the new techniques, we can explore in more detail the relative effectiveness, and accompanying costs, of each of the new techniques developed as part of the SELF compiler. The results will help identify those techniques that are worth including in any future implementation of a language like SELF.

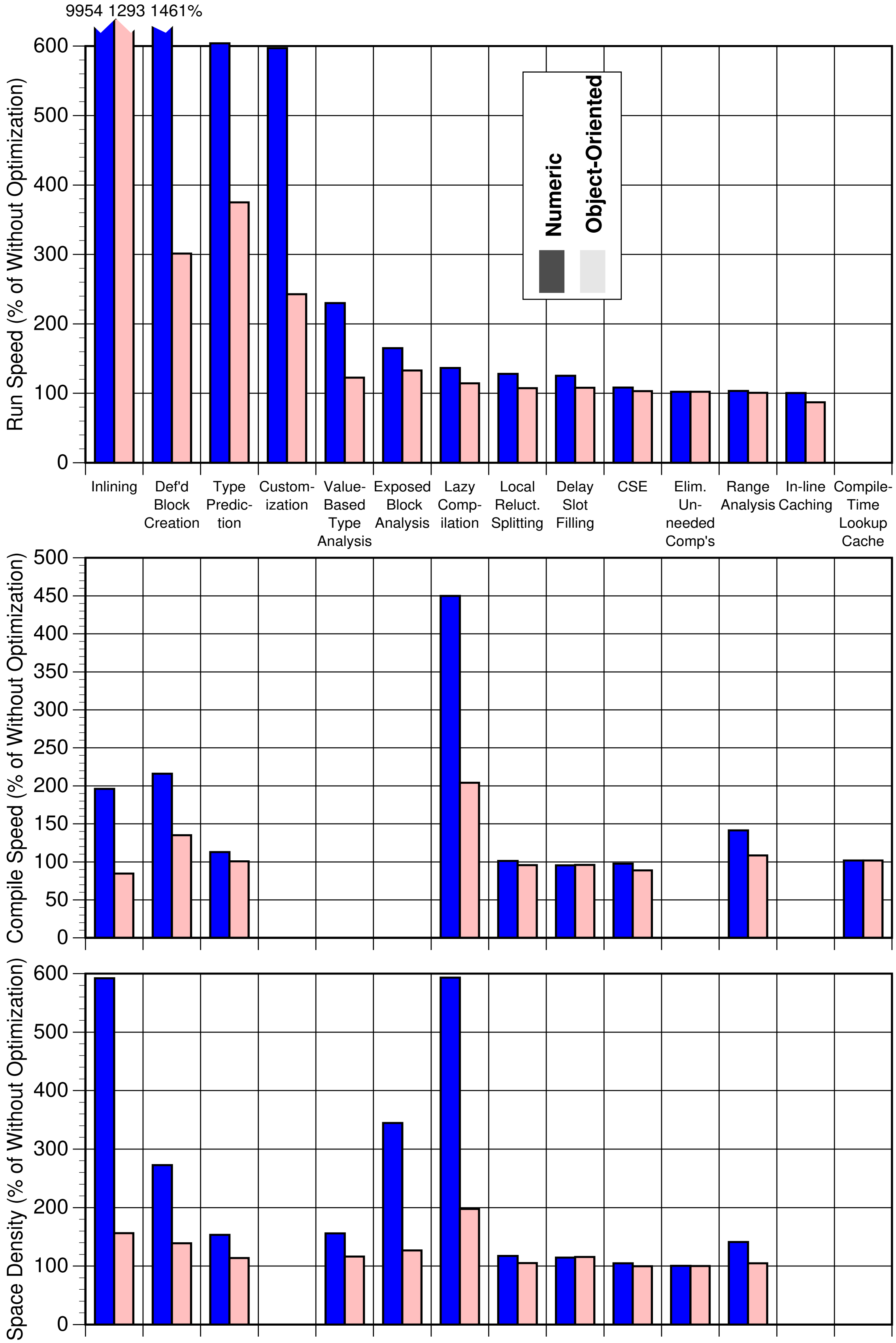
We calculate the benefits and costs of each technique by comparing the performance of the standard SELF configuration to the performance of a configuration with the technique disabled. In the charts, the effect of an optimization is shown by reporting the performance of the normal SELF system including the optimization relative to the performance of the SELF system with the optimization disabled; this approach remains consistent with our general visual theme of bigger bars indicating better results, in this case bigger bars indicating a more important or effective optimization.

The charts on the following page summarize the effect on performance of the various optimizations implemented in the SELF compiler. Detailed information for each optimization may be found in sections B.3 and B.4 of Appendix B.

Four optimizations are clear winners: inlining, deferred block creations, type prediction, and customization. Type analysis presumably is also a winner, but its impact could not be directly measured. Value-based type analysis, exposed block analysis, lazy compilation, local reluctant splitting, and delay slot filling also make significant contributions; lazy compilation in particular makes a huge improvement in compilation speed and compiled code space utilization. Of course, register allocation is also very important, but the impact of the particular details of the SELF register allocator, such as allocating variables instead of names, were not measured.

Common subexpression elimination, eliminating unneeded computations, range analysis, in-line caching, and caching compile-time lookups do not appear particularly important. Range analysis improves compilation speed and compiled code space density, therefore redeeming its rather slim execution speed improvement. As described in section B.3.4, in-line caching would appear more effective for the larger, more object-oriented benchmarks if not for a performance bug in the run-time system that halves the performance of the **`pathCache`** benchmark. Common subexpression elimination provides significant improvements for some benchmarks such as **`oo-bubble`** (see Appendix C), but overall performance improves by less than 5%.

In nearly all cases, the optimizations made a bigger difference for the smaller, more numeric benchmarks than for the larger, more object-oriented benchmarks. Part of this disparity probably stems from our concentration on the smaller benchmarks as the SELF compiler was being designed and implemented. Had we concentrated more on the object-oriented benchmarks, the trend might have been the reverse. Since both styles of program are important, we view this difference in effectiveness as an opportunity for future work.


9954 1293 1461%
Run Speed (% of Without Optimization)
0
100
200
300
400
500
600
Numeric
Object-Oriented
Inlining
Def'd Block Creation
Type Predic- tion
Custom- ization
Value- Based Type Analysis
Exposed Block Analysis
Lazy Comp- ilation
Local Reluct. Splitting
Delay Slot Filling
CSE
Elim. Un- needed Comp's
Range Analysis
In-line Caching
Compile- Time Lookup Cache
Compile Speed (% of Without Optimization)
0
50
100
150
200
250
300
350
400
450
500
Space Density (% of Without Optimization)
0
100
200
300
400
500
600

14.4 Some Remaining Sources of Overhead

The SELF system does not (yet) run as fast as a traditional language implementation such as optimized C. We would like to know the sources of this remaining gap in performance, to guide future work. In this section, we examine many of the sources of overhead that traditionally have slowed down pure object-oriented languages, such as extra run-time type tests, overflow checks, and array bounds checks. We can use the results to see how well the new techniques applied to SELF reduce these traditional sources of overhead, and we can direct future research towards reducing the cost of any overhead that remains significant.

We measure the cost of a particular source of overhead by constructing a version of the system without the source of overhead and comparing the speed of this version to the standard version of the system. Frequently, this new version of the system is not a legal SELF implementation in that not all SELF programs will run correctly on it, but for the benchmarks we measure, the altered system runs correctly. Unfortunately, not every possible source of overhead could be measured this way. For several key sources of overhead, it was too difficult to produce configurations that would simulate their absence. For example, the overhead of message passing, inheritance, dynamic typing, and user-defined control structures is nearly impossible to remove and measure directly; too much of the system internals and the benchmark source code relies on these features being present. Also, any overhead introduced as a result of deficiencies in the implementation of the SELF compiler cannot be measured in this way.

In the charts on the next page, we summarize the cost of those sources of overhead we were able to measure by displaying the performance of a configuration with the source of overhead removed relative to the performance of the standard SELF system; bigger bars mean that the overhead is more costly given SELF's current implementation technology. We also report the performance of two additional configurations:

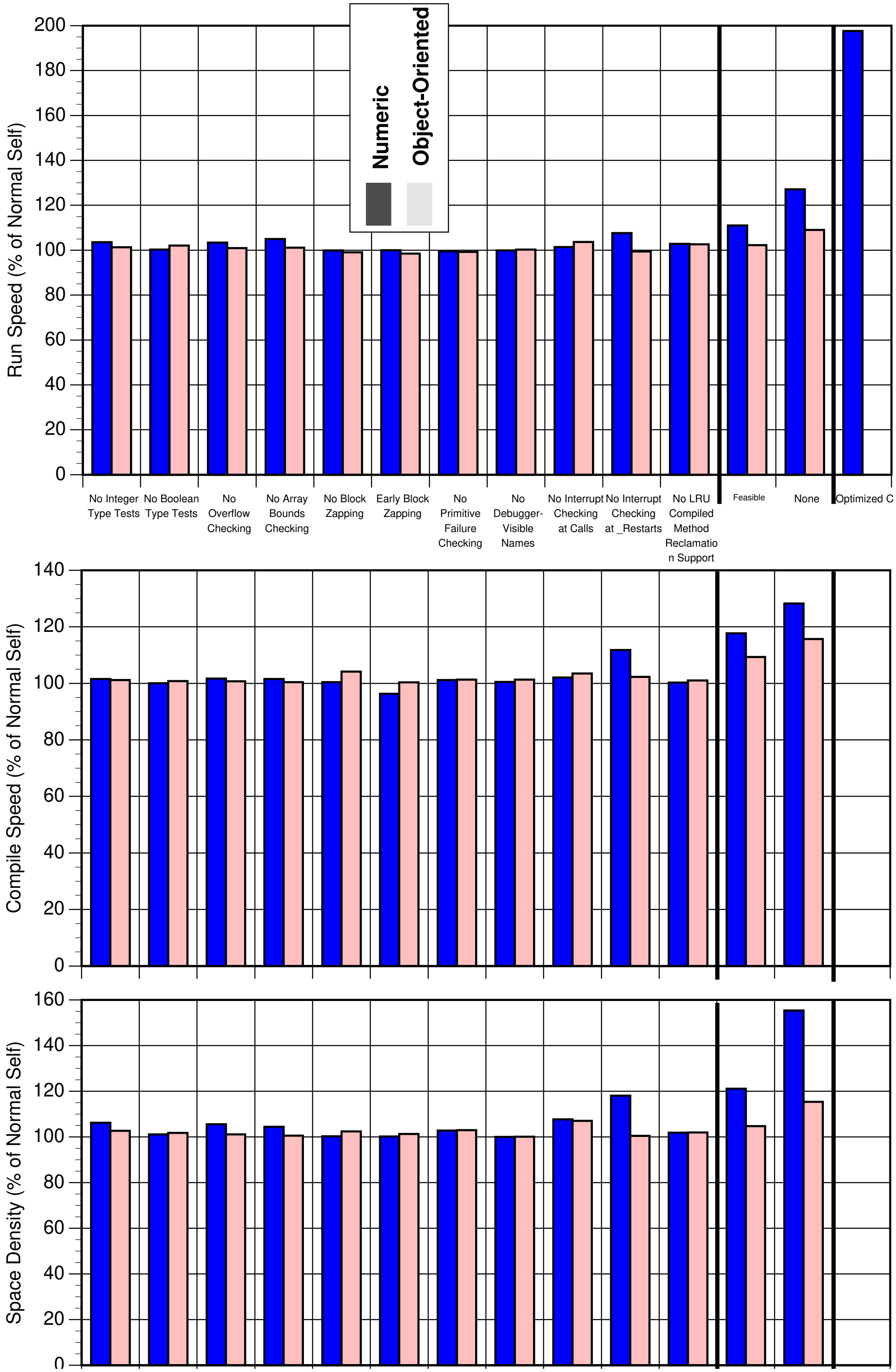
- **`feasible`** excludes those sources of overhead that might be avoided by implementing parts of the system differently: block zapping, interrupt checking via polling, and LRU compiled method reclamation support.
- **`none`** excludes all language and implementation overhead we measured, to try to produce a system that is as fast as possible.

We also report the execution speed of optimized C for comparison. More detailed analysis of each of the sources of overhead may be found in section B.5 of Appendix B.

Surprisingly, no single traditional source of overhead that we could measure imposes a significant execution speed cost. Array bounds checking, type testing, overflow checking, interrupt checking, and LRU compiled method reclamation support are the only measured sources of overhead with a non-trivial execution time cost, and none incurs more than 10% cost. This low cost reinforces the earlier overall performance measurements in illustrating how well the techniques implemented in the SELF compiler work at reducing or eliminating the sources of overhead that historically have plagued implementations of pure object-oriented languages.

There is still a fairly sizable gap in performance between the fastest version of SELF and optimized C that remains unaccounted for. Unless this gap stems solely from poorer implementation in the SELF compiler of traditional optimizations such as global register allocation or loop-invariant code motion, further improvements in the speed of pure object-oriented languages await additional insight.

Compilation time costs and compiled code space costs for the sources of overhead we can measure are fairly small. Polling-style interrupt checking and block zapping are the most compile-time- and compiled-code-space-consuming features to support. Interestingly, these worst offenders are all related to the particular system architecture of the SELF implementation rather than to any required language or environment features, so there is hope that this overhead could be reduced by redesigning various parts of the implementation.


Numeric
Object-Oriented
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
No Integer Type Tests
No Boolean Type Tests
No Overflow Checking
No Array Bounds Checking
No Block Zapping
Early Block Zapping
No Primitive Failure Checking
No Debugger-Visible Names
No Interrupt Checking at Calls
No Interrupt Checking at _Restarts
No LRU Compiled Method Reclamatio n Support
Feasible
None
Optimized C
0
20
40
60
80
100
120
140
160
180
200

14.5 Summary

By and large, the new techniques we developed for efficiently implementing SELF work well: SELF runs about half as fast as optimized C for the benchmarks we measured, a five-fold improvement over the best previous implementation of a similar pure object-oriented language (ParcPlace Smalltalk-80). SELF's performance is more than double the speed of a well-respected implementation of a similarly dynamically-typed language supporting generic arithmetic but also direct procedure calls and built-in control structures (the ORBIT compiler for T). Compile time and compiled code space costs are comparable to these other language implementations, with the exception of ParcPlace Smalltalk-80 which we believe compiles much faster than the other language implementations.

The bulk of the SELF compiler's good performance can be attributed to a few optimizations. Certain techniques simply must be applied to have any hope of decent performance, including inlining and deferred block creations. Type prediction and customization each improve execution performance by about a factor of four or five. Value-based type analysis, exposed block analysis, splitting, lazy compilation of uncommon branches, and delay slot filling also make significant contributions to run-time performance. Other optimizations have more modest benefits; common subexpression elimination and integer subrange analysis in particular were fairly complex to implement and were expected to have greater pay-offs. Some important techniques could not be measured, such as the effectiveness of type analysis or the space costs of customization, since these techniques were too integral to the system to disable.

Measurements of the remaining cost of some traditional sources of overhead in implementations of pure object-oriented languages indicate that none of the traditional bottlenecks continues to incur a significant cost. This result confirms the effectiveness of the new techniques, but unfortunately does not provide much help in guiding future research down profitable avenues.

Chapter 15 Conclusions

15.1 Results of this Work

With this work we have striven to demonstrate that pure object-oriented languages can be efficient on stock hardware, given suitable implementation techniques. To achieve high efficiency, we had to design and implement several new implementation techniques, including *customization*, *type prediction*, *iterative type analysis*, and *splitting*. Customization and type prediction extract representation-level type information from untyped source programs, and type analysis and splitting preserve this valuable information as long as possible. The accumulated type information then is used to statically-bind and inline away messages, especially those involved in user-defined control structures and generic arithmetic, leading to dramatic performance improvements.

By lazily compiling uncommon cases such as arithmetic overflows and primitive failures, the compiler can concentrate its efforts on the common-case parts of a program while still supporting the uncommon events if they should occur. This strategy resolves the tension between fast execution and powerful language features by providing the best of both worlds: good execution and compilation speed for common cases and support for more powerful but less common cases.

With these techniques the current SELF implementation runs small- to medium-sized benchmarks at about half the speed of optimized C, with compile times that are comparable to the optimizing C compiler and compiled code space usage that is less than double that of the optimizing C compiler. This new-found execution speed is more than five times faster than the fastest previous implementation of a similar language, ParcPlace Smalltalk-80.

Several general themes underlie this work. Our techniques frequently trade away space for speed, compiling multiple specialized versions of a single piece of source code; customization and splitting exemplify this approach. To minimize the compile time and compiled code space costs of this approach, many of our techniques are applied *lazily*. Methods are compiled and specialized lazily, only when first invoked; uncommon parts of the control flow graph are compiled lazily, only when first taken. Lazy compilation appears to be the saving grace which makes specialization practical.

15.2 Applicability of the Techniques

The techniques developed for SELF optimize programs that make heavy use of message passing. These techniques also apply to other languages that share these properties. Clearly, other pure, dynamically-typed object-oriented languages such as Smalltalk-80 could benefit directly from these techniques. As discussed in section 5.2.2, our techniques also apply to relatively pure, statically-typed object-oriented languages such as Trellis/Owl and Eiffel. Hybrid languages such as C++ and many object-oriented Lisps have less need for our techniques, since performance-critical parts of programs can be written in the lower-level non-object-oriented subset of the language. However, our techniques would still be useful to the extent that implementations wish to support and encourage the use of the object-oriented features of their languages. One researcher already has proposed extending C++ to support a form of customization [Lea90].

Our techniques also could improve the performance of many languages that do not claim to be object-oriented. These languages include powerful features in which several different representations of objects can be used interchangeably within programs. This ability is essentially the same as message passing, except that the set of possible representations usually is not user-extensible, and so we argue that these languages contain object-oriented subsets. Our techniques would be useful in improving performance of these languages to the extent that these "object-oriented" features are used by programs. For example, non-object-oriented languages supporting generic arithmetic, such as most Lisps and Icon, could significantly benefit from the inclusion of type analysis, type prediction, splitting, and lazy compilation of uncommon cases to extract and preserve representation-level information for optimization. To illustrate, our SELF implementation generates code for benchmarks using generic arithmetic that runs more than twice as fast as the code generated by the ORBIT compiler for the T dialect of Scheme, even though the T benchmarks use no message passing or user-defined control structures. Even when the T version of the benchmarks is rewritten to use unsafe integer-specific arithmetic (giving up the semantics of generic arithmetic), SELF still runs faster. Based on this result, we argue that language designers, implementors, and users should abandon unsafe integer-specific arithmetic in favor of safe, expressive generic arithmetic combined with optimization techniques like ours.

Language features other than generic arithmetic might benefit from our techniques. APL allows programs to manipulate scalars, arrays, and matrices of arbitrary dimension interchangeably, and our techniques might be used to lazily compile dimension-specific code to speed APL programs. Implementations of logic programming languages such as Prolog might benefit from knowing that along certain branches some logic variable is guaranteed to be instantiated; this knowledge could come from techniques related to type analysis and splitting. Similarly, implementations of programming languages supporting *futures* such as Multilisp [Hal85] and Mul-T [KHM89] could distinguish between known and unknown futures, compiling specialized code for each case (or perhaps just for the common case of known futures). Thus, our techniques may be more broadly applicable to a variety of modern programming languages beyond only pure object-oriented languages.

15.3 Future Work

While significant progress has been made in moving SELF and other pure object-oriented programming languages into the realm of practicality, more work remains to complete the task. Some applications require the maximum in efficiency, such as scientific and numerical applications like those traditionally written in Fortran. SELF as currently implemented is probably not efficient enough for such demanding users. One avenue of future research therefore would push the upper limits of performance towards that achieved for traditional languages and to extend the current implementation techniques to handle floating point representations as efficiently as integer representations are currently handled in the SELF implementation.

A related direction would attempt to validate that these techniques scale to much larger systems than have been measured so far. Several of our techniques rely on trading away compiled code space for run-time speed. For the systems measured, in the 100- to 1000-line program range, this potential space explosion has not been a problem in practice, but for larger programs, on the order of 10,000 or 100,000 lines, the concern still remains. More research could be done to ensure that the techniques are robust in the face of such large systems.

A third direction would focus on further improving the performance of object-oriented programs. Only a few of the benchmarks measured so far make significant use of the extra power of the SELF language beyond what is available in traditional languages. The question remains of how well our techniques will fare for programs that make heavy use of the advanced features of the language. Ideally, object-oriented programs would be written much faster and would be easier to change and extend than equivalent non-object-oriented programs, and would run just as fast as the non-object-oriented versions. This goal is not yet met by the current SELF implementation, which for the **`richards`** benchmark runs about a third the speed of the optimized C version. Some initial work has already begun in this direction [HCU91].

A final direction would address more of the programming environment issues. While the current SELF compiler compiles as fast as the optimizing C compiler on small- and medium-sized benchmarks (and compiles more than twice as fast as an optimizing C++ compiler), the fact that compilation takes place at run-time for SELF holds our system up to a higher standard. Users tend to become distracted by pauses of more than a fraction of a second, either from garbage collection or from run-time compilation, and their productivity drops correspondingly. Pauses of more than a dozen seconds or so bring about an even more severe distraction and decline in productivity. The current SELF implementation might meet the second level of performance but unfortunately is still not at the level of fraction-of-a-second compile pauses. To maintain a high-productivity environment, more research is needed to reconcile unnoticeable compiler pauses with good run-time performance. Fortunately, this problem also is being actively pursued [HCU91] and early results are quite promising.

15.4 Conclusion

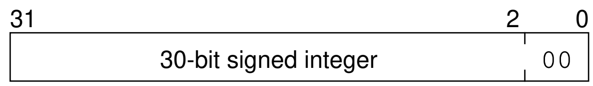
We believe that this work has demonstrated the feasibility of the new techniques and consequently the practicality of pure object-oriented languages for a wide range of applications. We hope that this demonstration will convince future language designers to avoid compromises in their designs motivated solely by concerns over the efficiency of a pure message passing model. We also hope that language users will begin to demand such simple, flexible languages.

Appendix A Object Formats

This appendix details the representation of objects used in the SELF memory system. Section 6.1 presents an overview of the SELF memory system.

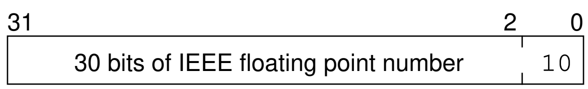
A.1 Tag Formats

A SELF memory space is organized as a linear array of aligned 32-bit words. Each word contains a low-order 2-bit tag field used to interpret the remaining 30 bits of information. A reference to an integer or floating point number encodes the number directly in the reference itself. References to other SELF objects and references to map objects embed the address of the object in the reference (there is no object table). The remaining tag format is used to mark the first header word of each object, as required by the scanning scheme discussed in section 6.1.2. Pointers to virtual machine functions and other objects not in the SELF heap are represented using raw machine addresses; since their addresses are at least 16-bit half-word aligned the scavenger will interpret them as immediates (either integers or floats) and so will not try to relocate them.

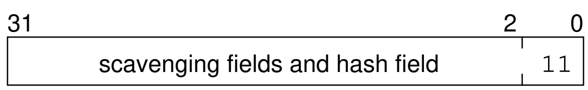


integer immediate (or virtual machine address)

reference to SELF heap object



floating point immediate (or virtual machine address)



mark header word (begins SELF heap object)

These tag formats were chosen to speed common operations on references. Tagged integers may be added, subtracted, or compared directly, as if the integers weren't tagged at all (since the tag field is zero for integers). Overflow checking is also free of additional overhead since the tag bits are low-order and overflows on the tagged format are detected by the hardware just as would be overflows on normal untagged integers. Integer multiplies and divides require an extra shift instruction prior to invoking the corresponding untagged operation. Other conversions between tagged and untagged integers require only an arithmetic shift instruction. Another nice benefit of this integer tag format is that object array accesses may use the tagged index directly to access the elements of the array; no extra multiplies or shifts are required to convert the index into an offset as would be required for untagged integers in a traditional language.

The tag format for references to heap objects is also relatively free of overhead. Since SELF objects always begin on word boundaries, on byte-addressed architectures the 2-bit low-order tag format does not reduce the available address space (the **`01`** tag is simply replaced with a **`00`** tag to turn a tagged reference into a raw word-aligned address). Additionally, on machines with a register-offset addressing mode, the tag can be stripped off automatically when accessing a field within the referenced object at a constant offset by folding the decrement into the offset in the instruction. For example, to access the n^{th} word in an object (origin 0), where n is a compile-time constant, the compiler can simply generate (in SPARC assembly syntax)

```
ld [%object + (n*bytes_per_word - 01)], %t
```

where the offset in the load instruction is a compile-time constant.

Testing whether a reference is an integer immediate requires a simple

```
andcc %object, #3, %g0
bz _integer
```

sequence to check the two low-order bits for a **`00`** tag. Testing for a heap object reference also only requires a

```
andcc %object, #1, %g0
bnz _heap
```

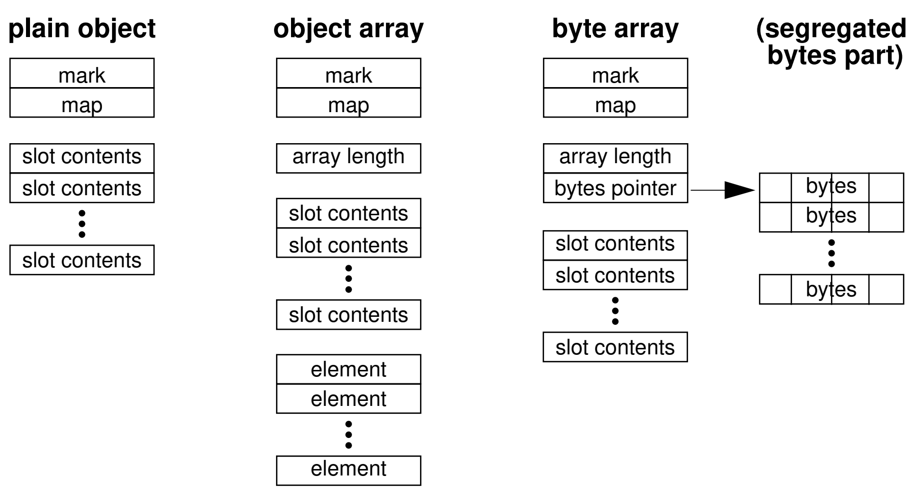
sequence to test whether the low-order bit is non-zero; this sequence assumes that the reference cannot be a mark word, which is the case for all object references manipulated by user programs. Testing for a floating point immediate similarly only requires a

```
andcc %object, #2, %g0
bnz _float
```

sequence to test the second-to-lowest-order bit (again, assuming the reference cannot be a mark word).

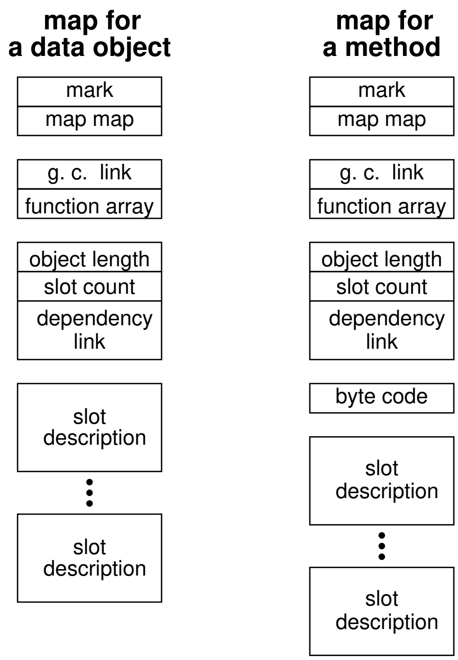
A.2 Object Layout

Each heap object begins with two header words. The first word is the mark word, identifying the beginning of the object. The mark word contains several bitfields used by the scavenger and a bitfield used by the SELF **`_IdentityHash`** primitive. The second word of an object is a tagged reference to the object's map. A SELF object with assignable slots contains additional words to represent their contents. Array objects include their length (tagged as a SELF integer to prevent interactions with scavenging and scanning) before any assignable slot contents. Object arrays include their elements (tagged object references) after any assignable slot contents (afterwards so that assignable slot contents have constant offsets within a given clone family), while byte arrays include an untagged pointer to a word-aligned sequence of 8-bit bytes before any assignable slot contents (segregation of packed bytes parts is described in section 6.1.2).

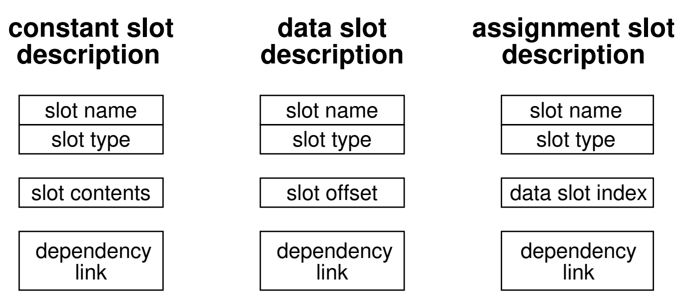


Like other objects, map objects begin with mark and map words. All map objects share the same map, called the *map map*; the map map is its own map. All maps in new-space are linked together by their third words; after a scavenge the system traverses this list to perform special *finalization* of inaccessible maps. The fourth word of a map contains the virtual machine address of an array of function pointers; these functions perform type-dependent operations on objects or their maps.*

For maps of objects with slots, the fifth word specifies the size of the object in words. The sixth word indicates the number of slots in the object and the number of slot descriptions in the map. The next two words contain the change dependency link for the map, as described in section 13.2.2. These four words are tagged as integers. If the map is for a method, the ninth word references the byte code object representing the method's source code (byte code objects are described in section 6.2).

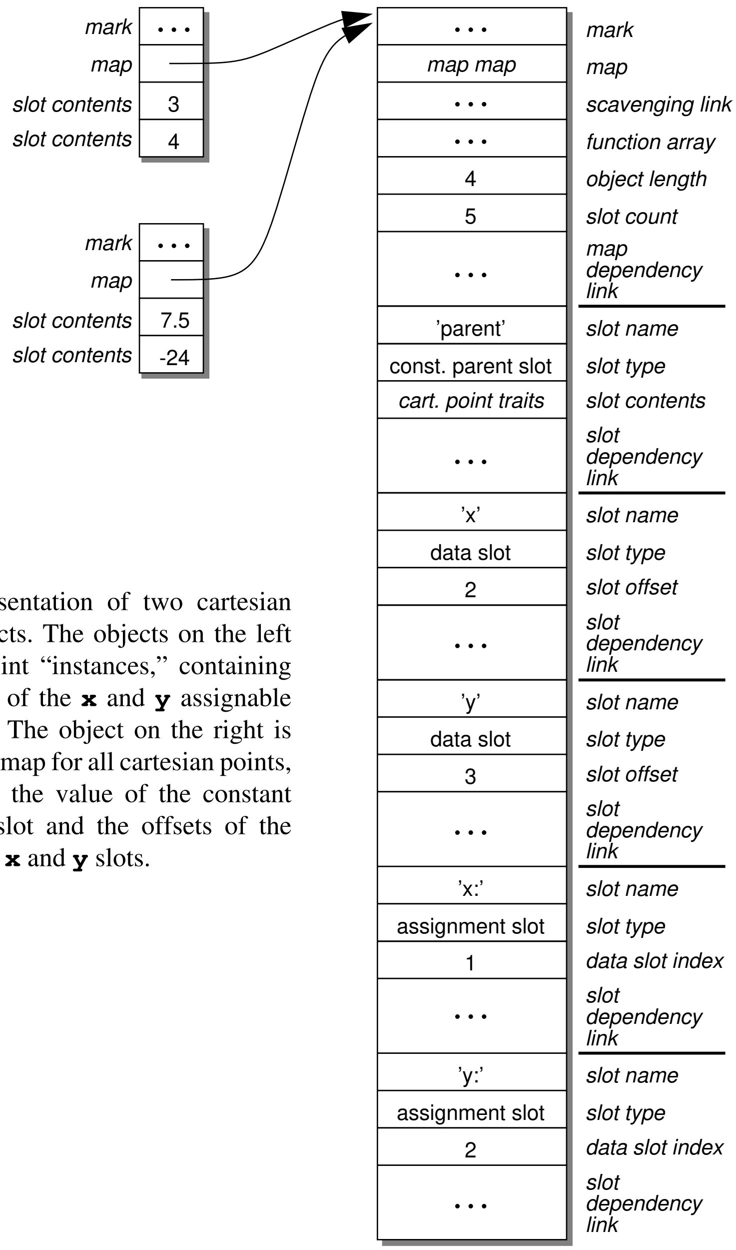


Finally, the map includes a five-word description for each of the object's slots. The first word points to a SELF string object representing the name of the slot. The next word describes both the type of the slot (either constant data slot, assignable data slot, or assignment slot) and whether the slot is a parent slot.** The third word of a slot description contains either the contents of the slot (if the slot is a constant slot), the offset within the object of the contents of the slot (if the slot is an assignable data slot), or the index of the corresponding data slot description (if the slot is an assignment slot). The last two words of each slot contain the change dependency link for that slot, as described in section 13.2.2.



The representations of a pair of cartesian points and their map are displayed on the next page.

* This function pointer array is the virtual function array generated by the C++ compiler.
** In SELF parents are prioritized; the priority of a parent slot is also stored in the second word of the slot description.



The representation of two cartesian point objects. The objects on the left are the point "instances," containing the values of the **x** and **y** assignable data slots. The object on the right is the shared map for all cartesian points, containing the value of the constant **parent** slot and the offsets of the assignable **x** and **y** slots.

Appendix B Detailed Performance Evaluation

The first two sections of this appendix contain additional information about the benchmarks and the measurement procedures. This appendix also includes detailed analyses of the effectiveness of the various techniques included in the SELF implementation (in sections B.3 and B.4) and of some of the remaining sources of overhead that slow SELF down compared to optimized C (in section B.5). Finally, section B.6 discusses the space costs of the extra information generated by the SELF compiler beyond just native machine instructions.

B.1 Detailed Description of the Benchmarks

We measured the following micro-benchmarks:

- **`recur`** is a tiny recursive benchmark that stresses method call and integer comparison and subtraction, adapted from the **`testActivationReturn`** Smalltalk-80 micro-benchmark [Kra83].
- **`sumTo`** adds up all the numbers from its receiver (1) to its argument (10000), 100 times over. **`fastSumTo`** is the same as **`sumTo`**, except that the body of **`sumTo`** is inlined into the outer loop (manually in the C, Smalltalk, and T versions, automatically in the SELF version). **`sumFromTo`** is similar to **`sumTo`**, except that it adds up all the numbers from its first argument (1) to its second (10000), initializing the accumulator to its receiver (0). These benchmarks stress generic arithmetic applied to integers and user-defined control structures. **`sumFromTo`** is a benchmark used to measure the TS Typed Smalltalk compiler (described in section 3.1.3).
- **`nestedLoop`** increments a counter inside a doubly-nested **`for`**-style loop, each loop iterating 100 times; this test is itself run 100 times. This test also stresses generic arithmetic applied to integers and user-defined control structures, and is another benchmark used to measure the TS Typed Smalltalk compiler.
- **`atAllPut`** stores an integer (7) into all elements of a 100,000-element-long vector, and so stresses iterating through and storing into arrays. It was originally suggested to us by Peter Deutsch as an interesting micro-benchmark [Deu89].
- **`sumAll`** adds up all the elements of a 100,000-element-long vector. This test stresses iterating through arrays and generic arithmetic of integers, and is another benchmark used on the TS Typed Smalltalk compiler.
- **`incrementAll`** increments all the elements of a 100,000-element-long vector, stressing iterating through arrays, storing into arrays, and generic arithmetic on integers. This benchmark is another TS Typed Smalltalk benchmark.
- **`sieve`** finds all the primes between 1 and 8190 using Eratosthenes' sieve algorithm and stresses integer calculations, integer comparisons, and accessing arrays of booleans.
- **`tak`** executes the recursive Tak benchmark from the Gabriel Lisp benchmarks [Gab85], which stresses method calling and integer arithmetic. **`takl`** performs the same algorithm, but uses lists of cons-cells to represent integers, and so additionally stresses list traversals and memory allocation.

Most of these micro-benchmarks are 1 to 3 lines long. **`sieve`** is 8 lines long, **`tak`** is 6 lines long, and **`takl`** is 10 lines long.

We measured the following Stanford integer benchmarks, each of which exercises integer calculations, generic arithmetic, array accessing, and user-defined control structures (particularly **`for`**-style loops):

- **`perm`** and **`oo-perm`** are recursive permutation programs (25 lines long each).
- **`towers`** and **`oo-towers`** recursively solve the Towers of Hanoi problem for 14 disks (60 lines long each).
- **`queens`** and **`oo-queens`** solve the eight-queens placement problem 50 times (35 lines long each).
- **`intmm`** and **`oo-intmm`** multiply two 40-by-40 matrices of random integers. Since two-dimensional matrices are not supported directly by SELF, Smalltalk, or T, in these languages a matrix is represented by an array of arrays, in contrast to the contiguous representation used in the C version. These benchmarks are each 30 lines long.
- **`quick`** and **`oo-quick`** sort an array of 5000 random integers using the quicksort algorithm (35 lines long each).
- **`bubble`** and **`oo-bubble`** sort an array of 500 random integers using the bubblesort algorithm (20 lines long each).

- **`tree`** and **`oo-tree`** sort 5000 random integers using insertion into a sorted binary tree data structure (25 lines long each). These benchmarks stress memory allocation and data structure manipulation instead of array accessing.
- **`puzzle`** solves a time-consuming placement problem (170 lines long). This benchmark is unusual in that its source code size is dominated by code to initialize the puzzle data structures, and so demands good compiler speed more than the other benchmarks.

Source code for all benchmarks is available from the author upon request.

B.2 Measurement Procedures

We measured the speed of the SELF, C, and T implementations in milliseconds of CPU time (user plus system time), as reported by the UNIX **`getrusage`** system call for SELF and C and the **`time`** function for T. The Smalltalk-80 implementation only reports elapsed real time, so all measurements of the performance of Smalltalk programs are in milliseconds of real time rather than CPU time. Since the Smalltalk-80 real-time numbers did not fluctuate significantly from run to run (indicating that any extra overhead included in real time for Smalltalk-80 was relatively constant), and the real-time measurements for SELF and for C were about the same as the corresponding CPU-time measurements (indicating that any extra overhead included in real time for SELF and C was small), we believe that the CPU time for Smalltalk-80 is likely to be close to the real-time measurements. Therefore, we feel that comparisons among Smalltalk-80's real-time measurements and the other language's CPU-time measurements are reasonably valid.

We measured the run time of each benchmark by executing the benchmark in a loop of 10 iterations, reading the elapsed time before and after the loop, and dividing the resulting difference by 10. This measurement of 10 iterations helps to increase the effective resolution of the hardware clock (which is only to the nearest 10ms on a Sun-4/260) and smooth over any variations from run to run.

The compile times for the C version of each benchmark were calculated by using the UNIX **`time`** command around the compilation and optimization of a source file containing only the benchmark being measured. Thus, compilation time for C includes the time for reading and writing files but not the time for linking the output object file into an executable program. The compile times for the ORBIT compiler were computed similarly by using T's **`time`** function around the invocation of the **`orbit`** function on the name of the file containing the benchmark. Thus, T's compile time includes the time to read and write files but not the time to load the resulting output file into the running T system.

Separating compile time from run time in SELF and Smalltalk is complicated by dynamic compilation. The first execution of a piece of code includes both execution time and compilation time. Our technique for calculating compilation and execution time for a SELF benchmark is first to flush the compiled code cache (to make sure none of the code the benchmark will use is already compiled in the cache). Then the benchmark is run once; this first run includes both compilation time and run time. Then the benchmark is run 10 more times; these runs include only execution, since the compiled code is already in the compiled code cache thanks to the first run. The final execution time is calculated as the time of the last 10 runs divided by 10 (just as for C and T), and the compilation time is calculated as the time for the first run minus the average execution time for the last 10 runs. This approach assumes that the execution time of the first run of the benchmark is equal to the average subsequent execution, which may not be completely accurate (e.g., because of hardware caching and paging effects), but is probably close enough to use the calculated compile times as a rough measure of the actual compile-time overhead of our implementation. Unfortunately, the Smalltalk-80 system does not provide a mechanism to flush the compiled code cache, so compilation speed numbers are unreliable. Consequently, the same process for measuring SELF is performed for Smalltalk (ignoring the cache flush step), but only the average execution time numbers are retained. Compiled code space figures also are available for Smalltalk.

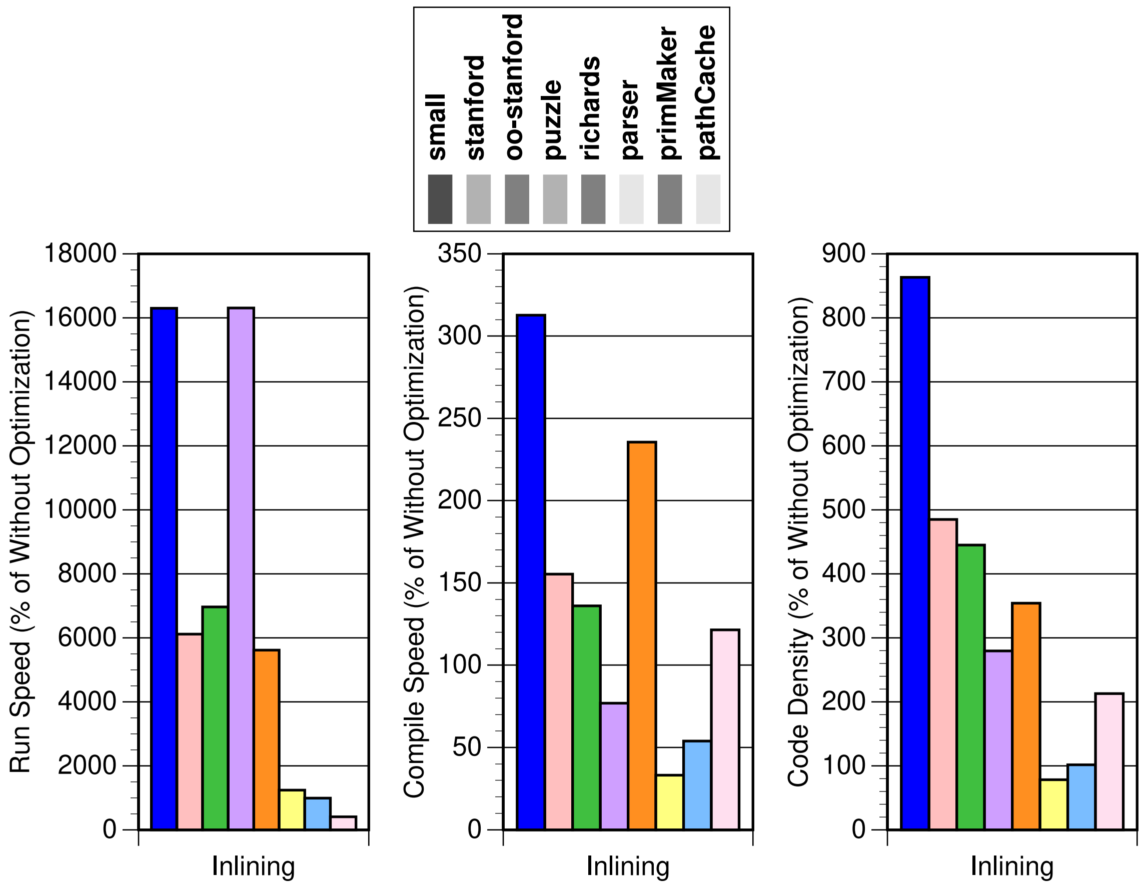
Frequently, to reduce the number of data points displayed in the charts, the results of individual benchmarks have been combined together to form a result for a suite of similar benchmarks. The average result for a benchmark suite was calculated by taking each individual benchmark's result, normalizing it to some reference result (typically either the performance of optimized C or the performance of the standard SELF configuration), and then taking the geometric mean of the normalized results for the benchmarks in the suite. Appendix C contains the original raw data for all the measurements.

B.3 Relative Effectiveness of the Techniques

This section explores the effectiveness of individual optimizations in detail, with the exception of splitting strategies which are the subject of the next section. These techniques will be covered in the order in which they were described in this dissertation. Summary information may be found in section 14.3.

B.3.1 Inlining

The SELF compiler relies on aggressive inlining to achieve good performance, as described in Chapter 7. To verify that the SELF system would be intolerably slow without inlining, we measured a version of SELF in which inlining of both messages and calls to primitives was disabled. The results are displayed in the following charts.

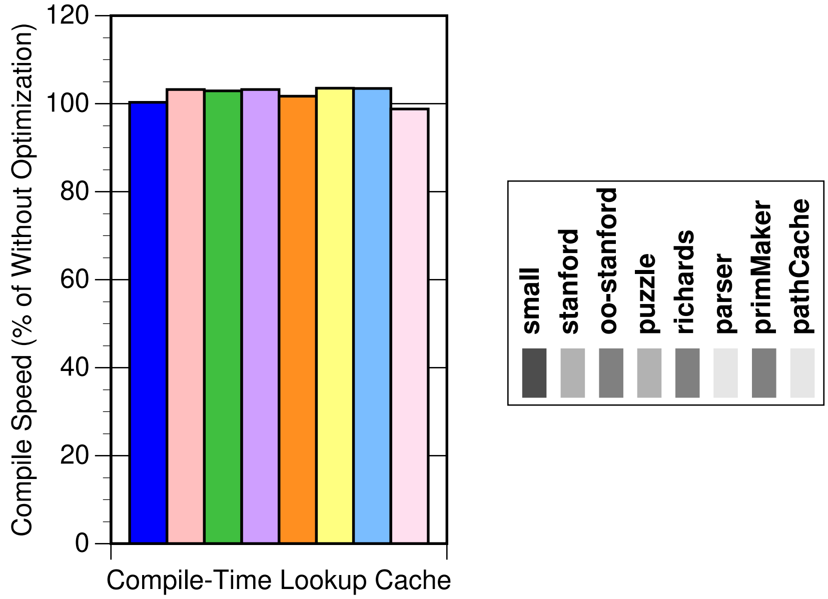


Inlining makes a huge difference in performance. The smaller numeric benchmarks run about two orders of magnitude faster with inlining, and even the larger non-numeric benchmarks run between 4 and 55 times faster with inlining. These results support our contention that aggressive inlining is the key to achieving good performance in pure object-oriented languages.

Surprisingly, inlining frequently speeds compilation and almost always saves compiled code space! This counter-intuitive result illustrates the difference between inlining in a traditional language and inlining in the SELF system. In SELF, the compiler uses inlining mostly for optimizing user-defined control structures and variable accesses, where the resulting inlined control flow graph is usually much smaller than the original un-inlined graph. These sorts of inlined constructs are already “inlined” in the traditional language environment. Inlining of larger “user-level” methods or procedures does usually increase compile time and compiled code space as has been observed in traditional environments; the SELF compiler simply spends much more of its time inlining things that shrink the control flow graph than things that expand it. Of course, a system never designed to perform inlining in the first place could compile faster than a compiler that could perform inlining but did not, so these results probably overstate the benefits of inlining for compilation speed.

B.3.2 Caching Compile-Time Message Lookups

The SELF compiler includes a message lookup cache to speed compile-time message lookups, as described in section 7.1.3. The following chart reports the effectiveness of this cache in speeding compilation.

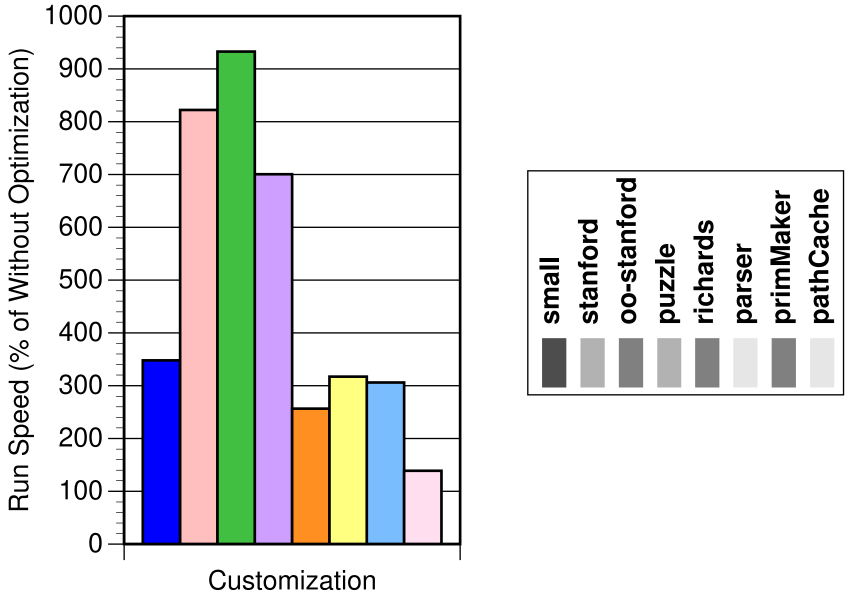


The compile-time message lookup cache speeds compilation by only a few percent. This poor showing is the result of being able to use the cache only within a single compilation and not across calls to the compiler. We are investigating system designs that would support a longer-lived compile-time message lookup cache which we believe could save a significant amount of compile time.

B.3.3 Customization

The SELF compiler uses customization to provide extra type information about **`self`** that enables much more inlining, as described in Chapter 8. Much of the implementation of the SELF system assumes that compiled methods are customized, so it is difficult to completely disable customization in order to measure its impact on the performance of the system. Fortunately, one aspect of customization can be disabled relatively easily: its contribution of the type of **`self`**. After the method prologue, the disabled compiler “forgets” the type of **`self`** by rebinding it to the unknown type. The run-time performance of this configuration should closely approximate the run-time performance of a version of the system with no customization at all, since the dominating effect of customization is this extra type information, not the duplication of methods and in-line caches. However, the compilation speed and compiled code space efficiency of the disabled configuration probably would be worse than a version of the system that did not customize at all, since the disabled configuration still can compile multiple versions of source methods, and so we cannot report accurate compile time costs and compiled code space costs for customization with this configuration.

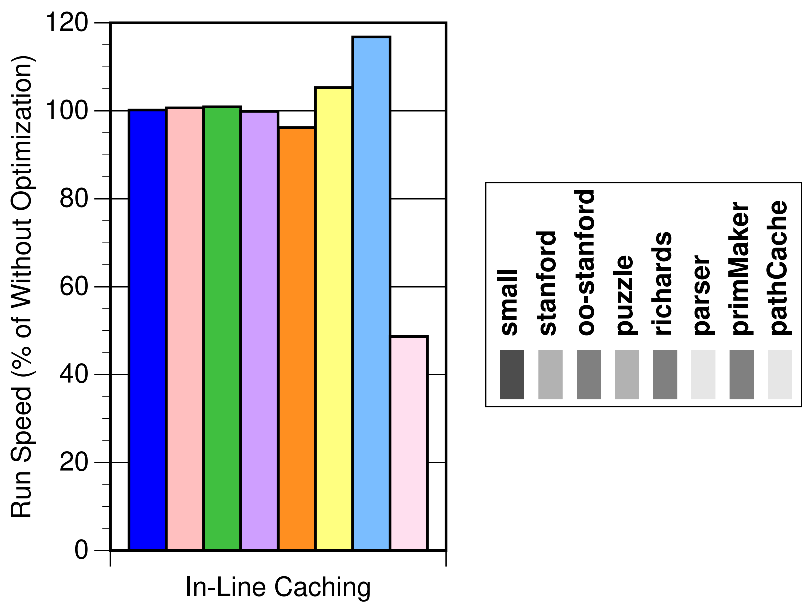
The following chart presents customization's impact on run-time execution performance.



Customization improves performance significantly, enabling these SELF benchmarks to run between 1.4 and 9 times faster. The extra type information provided by customization has been put to good use by the compiler in speeding run-time performance.

B.3.4 In-Line Caching

In-line caching speeds message sends that are neither inlined nor statically bound, as described in section 8.5. Since the SELF compiler is so effective at inlining messages, some question might remain as to whether in-line caching is still important for good performance. The following chart reports the impact of in-line caching on the execution performance of the SELF system; in-line caching affects neither compilation speed nor compiled code density.



In-line caching makes little impact on the smaller benchmarks, since most sends are inlined in these benchmarks and those that remain are frequently statically-bound to a single target method and so require no in-line caching. The **`primMaker`** and the **`parser`** benchmarks benefit more from in-line caching, but not by more than 20%. This result is rather surprising, given in-line caching's importance in the Deutsch-Schiffman Smalltalk-80 system, and it serves as confirmation of the effectiveness of inlining and static binding.

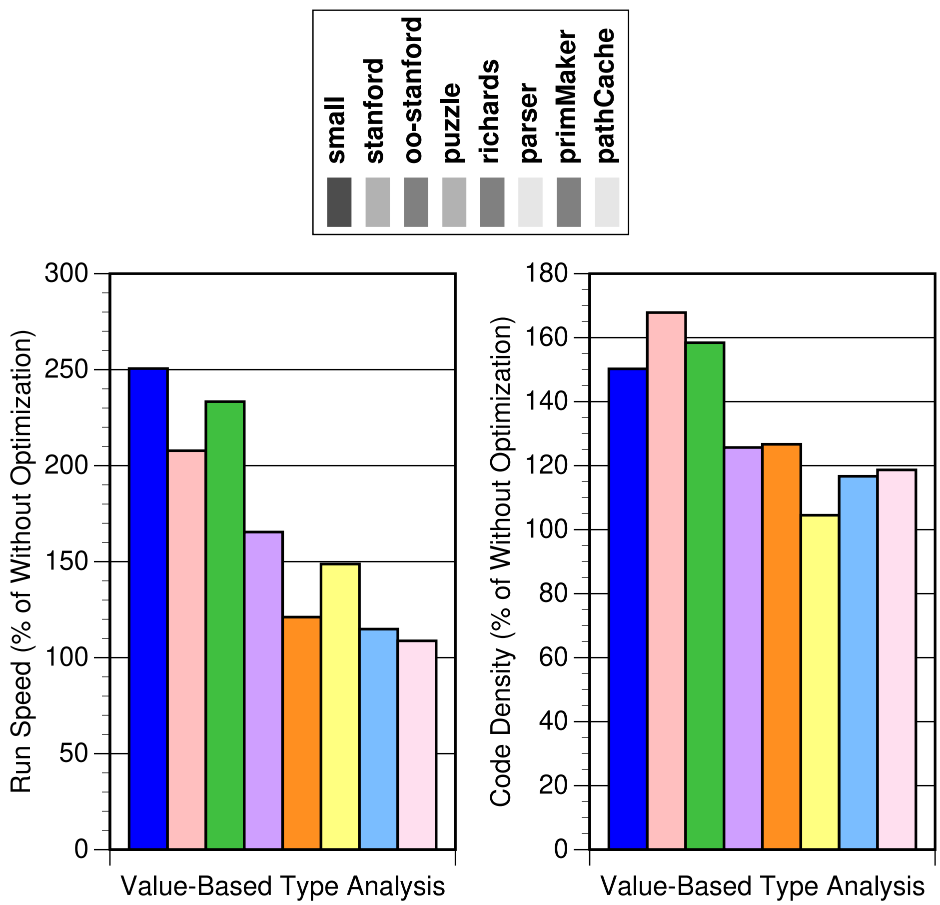
Amazingly, the **`pathCache`** benchmark runs more than twice as *slowly* with in-line caching! This unexpected result is caused by a performance bug in the run-time system (not the compiler) which slows down messages with more than 9 different receiver types. Since these measurements were made and the problem discovered, the implementation of in-line caches with many receiver types has been improved, and now in-line cached sends reportedly are almost always faster than uncached sends [Höl91].

B.3.5 Type Analysis

The SELF compiler relies on type analysis to propagate type information through the control flow graph and thus inline away more messages and avoid unnecessary type tests. Type analysis was the subject of Chapter 9 and much of Chapter 11. Like customization, type analysis is difficult to disable in the SELF compiler, since type analysis permeates the compiler's entire design. Therefore, we could not directly measure the effectiveness of type analysis using a version of SELF that performs no type analysis.

B.3.6 Value-Based Type Analysis

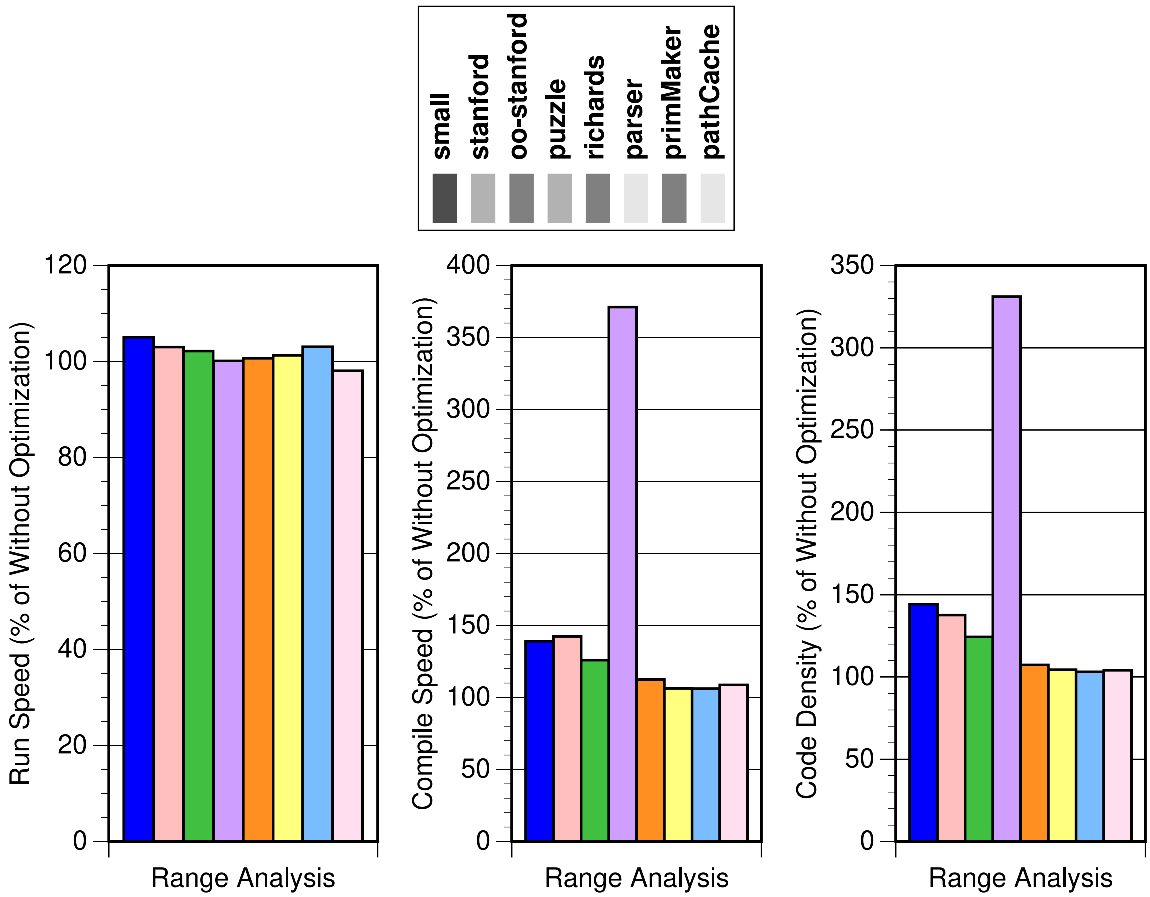
Fortunately, we can measure the effect of some parts of type analysis, such as the SELF compiler's use of value objects to link names to types and so improve the effectiveness of optimizations such as type prediction, as described in section 9.1.3. We can approximate a version of SELF without values by creating a new dummy value object whenever a name is assigned. Consequently, no two names will ever share the same value. Execution speed and compiled code space efficiency of this hobbled configuration should be close to a version of SELF without values; compilation speed would not since a compiler written without values should run faster than one written with values that have been rendered useless. The following charts report the impact of value-based type analysis on execution speed and compiled code space density.



These results show that value-based type analysis makes a significant improvement in execution performance. The smaller benchmark suites run more than twice as fast with value-based type analysis. The use of values improves the smaller, numerical benchmarks more than the larger benchmarks for two reasons. First, the numerical benchmarks rely more on type prediction for good performance (corroborated by the results in section B.3.8), and values improve the effectiveness of type prediction. Second, the smaller benchmarks send more messages to a single expression, for example an accumulator which receives a **+** message every time through a loop, and values allow the type information propagated from one name to another to be exploited when optimizing these sequences of messages. Value-based type analysis also improves the density of the compiled code by eliminating the need for repeated, redundant type tests which would take up additional compiled code space.

B.3.7 Integer Subrange Analysis

Integer subrange types support optimizations that depend on range analysis, such as eliminating unnecessary overflow checks and array bounds checks, as described in section 9.1.4.3. The effect of integer subrange types can be determined easily by using the more general integer map type wherever an integer subrange type would have been introduced. The charts below report the effectiveness of integer subrange analysis.

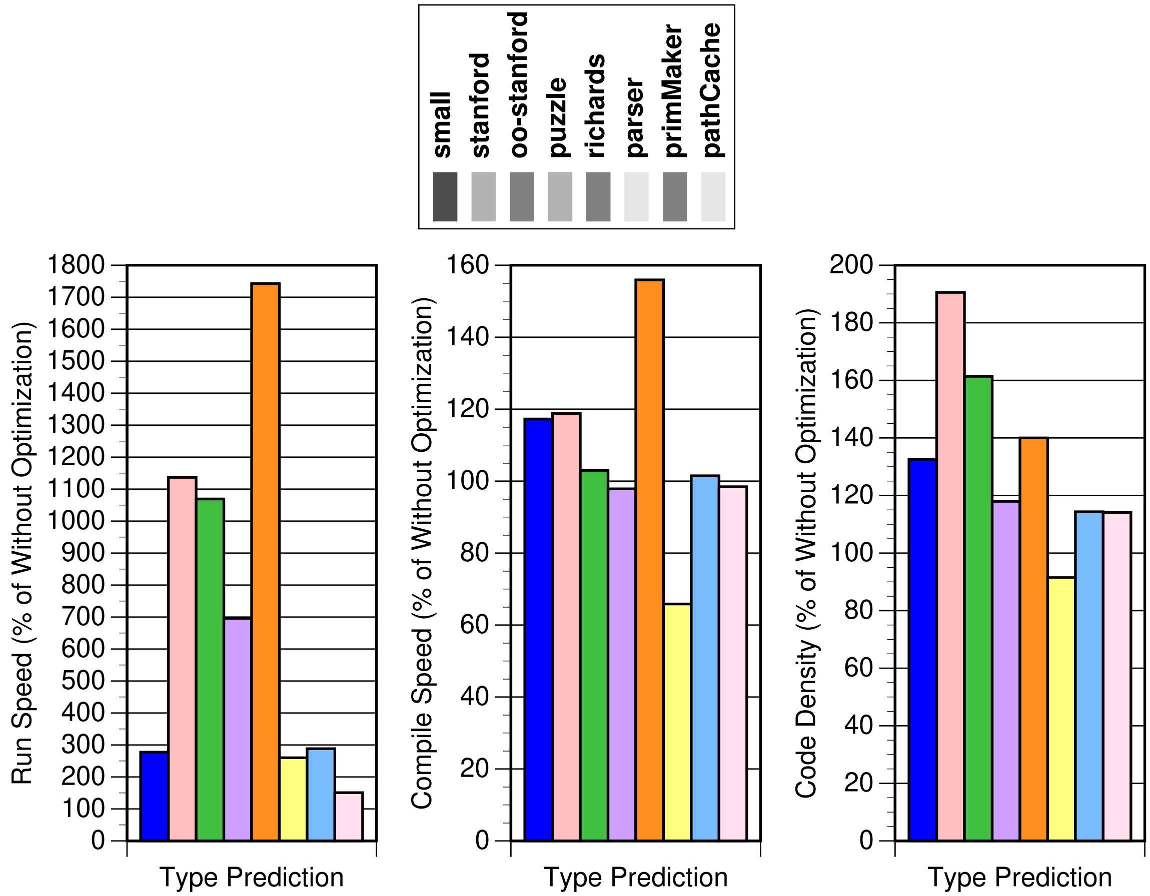


Integer subrange analysis seems to have little effect on execution performance, improving the speed of the benchmarks less than 5%. Range analysis' impact would be greater if not for two limiting factors. First, the current register allocator does not ensure that adjacent variables get allocated to the same location (as described in section 12.1.5), and so frequently a register move instruction remains after integer subrange analysis eliminates an overflow check. Second, range analysis currently is limited by requiring integer constants as upper and lower bounds. Allowing some form of symbolic bounds or constraints on unknown values might improve the effectiveness of range analysis, especially at eliminating array bounds checks for iteration over arrays of unknown size.

Integer subrange analysis frequently makes a large improvement in both compilation speed and compiled code space costs. Optimizing away possible uncommon branches via integer subrange analysis makes a big improvement in compilation speed, if not execution speed, and saves a lot of compiled code space.

B.3.8 Type Prediction

The SELF compiler uses type prediction (described in section 9.4) to increase the amount of type information available to the compiler for certain common message sends. The effectiveness of this technique can be easily measured by simply not performing type prediction. The following charts report type prediction's contribution to execution performance and its effect on compilation speed and compiled code density.



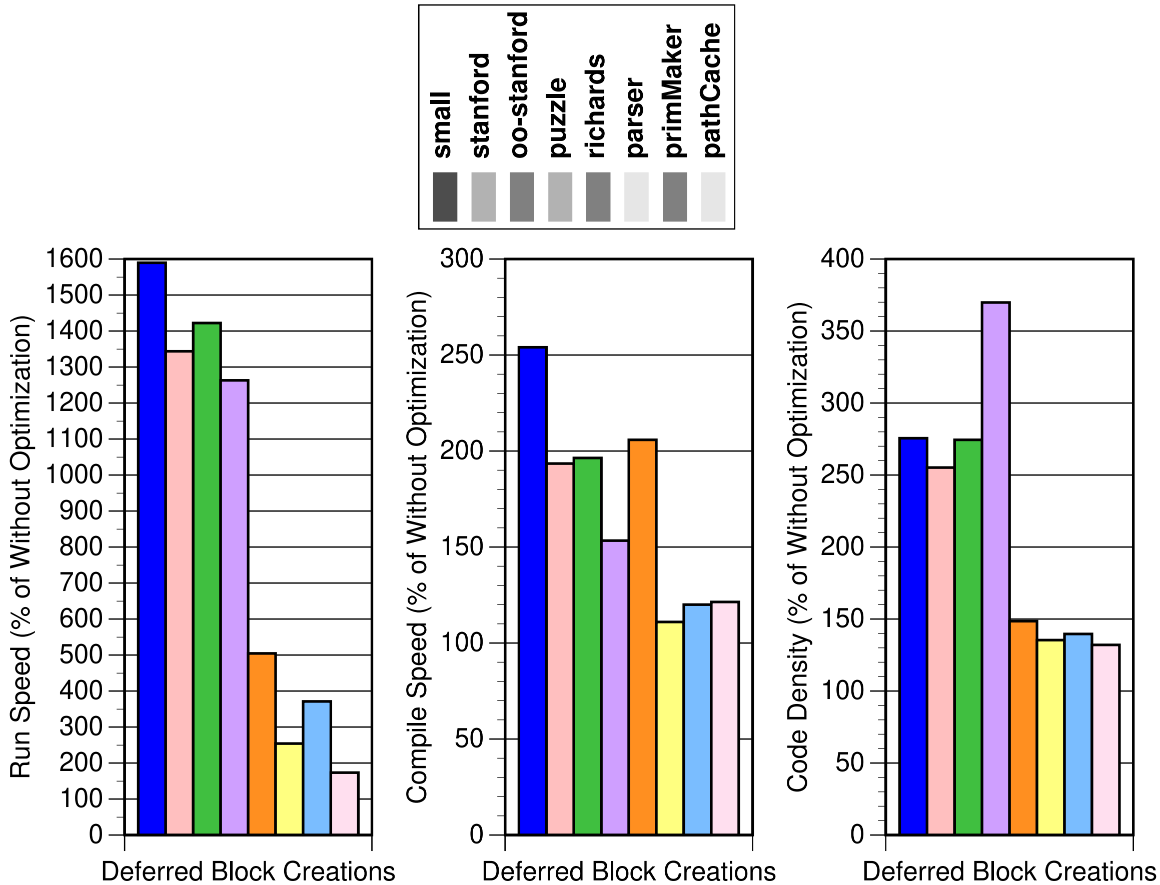
With type prediction SELF programs run much faster, between 1.5 and 17.5 times faster. Type prediction is at least as important as customization in achieving good performance on these benchmarks. Type prediction has a mixed effect on compilation speed, sometimes speeding and sometimes slowing compilation. The compiler sometimes can compile faster with type prediction because with lazy compilation of uncommon branches the common-case type-predicted version of a message send can be simpler and thus faster to compile than the original unpredicted message send. Type prediction reduces compiled code space costs for most benchmarks for much the same reason.

B.3.9 Block Analysis

Since blocks are such a key part of the SELF system, used by virtually all control structures, the SELF compiler incorporates several optimizations designed to reduce the cost of blocks. These optimizations were described in section 9.5.2.

B.3.9.1 Deferring Block Creations

The SELF compiler defers creating a block until it is first needed as a real run-time value, if at all. Disabling this optimization is easy. The charts below report the effect of deferring block creations.

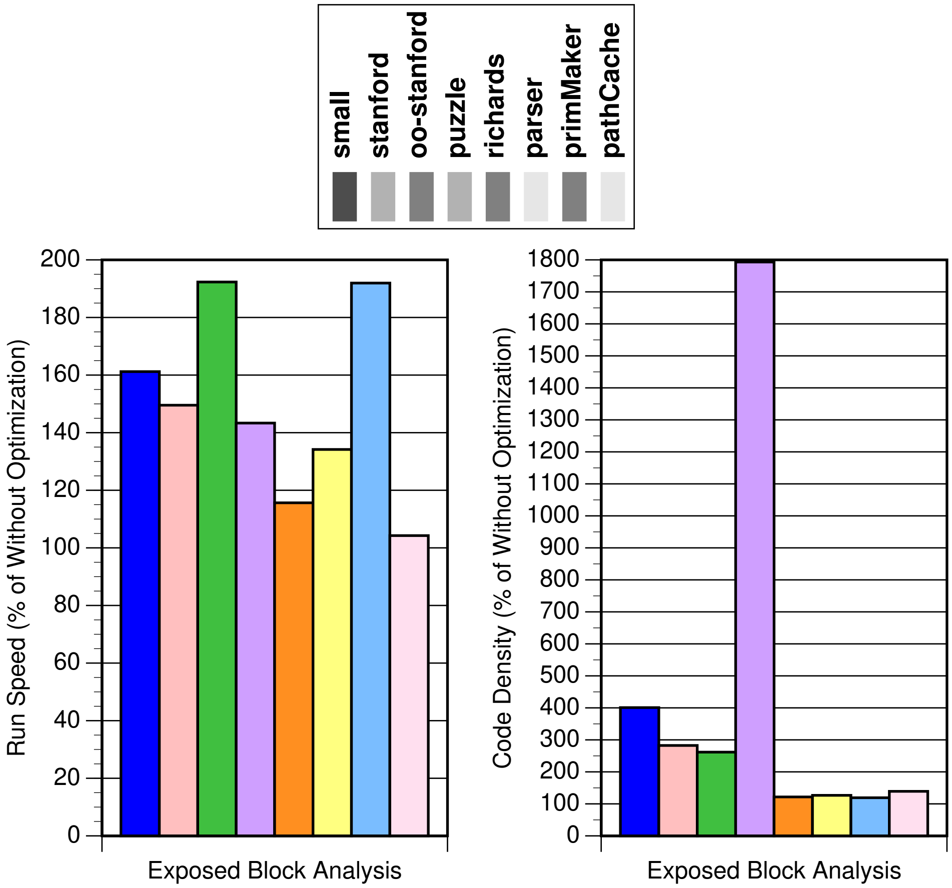


Delaying and eliminating unneeded block creations clearly boosts execution speed, frequently more than either customization or type prediction. Of course, this effect is attributable primarily to eliminating unneeded block creations entirely than simply to delaying some block creations until the latest possible moment. In any case, these measurements dramatically display the importance of optimizing block creation to get good performance in a language that relies on user-defined control structures.

Eliminating block creations saves a significant amount of compile time, speeding the compiler by more than 50% for the smaller benchmarks. Not surprisingly, eliminating the code that would have created the unnecessary blocks significantly improves compiled code space density, by more than a factor of 2.5 for the smaller benchmarks.

B.3.9.2 Exposed Block Analysis

The SELF compiler tracks which blocks have been created and passed out as run-time arguments to other methods, to reduce the number of local variables and arguments which must be considered up-level accessible. This analysis can be disabled by treating all blocks as exposed. Unfortunately, this simulation strategy does not reclaim the compile time for manipulating lists of exposed blocks, so the impact of exposed block analysis on compilation time cannot be determined. The following charts report the impact of exposed block analysis on execution speed and compiled code space efficiency.



Analyzing which blocks are exposed significantly speeds SELF programs, making some benchmarks run almost twice as fast. Exposed block analysis greatly reduces the number of local variables and arguments that must be considered visible at send points, **`_Restart`** points, and uncommon branch entry points, which in turn eliminates many unnecessary register moves. Exposed block analysis can save a lot of compiled code space, by about a factor of 3 for the smaller numerical benchmark suites. This savings comes from the extra register moves and stack accesses that exposed block analysis proves may be eliminated.

B.3.10 Common Subexpression Elimination

The SELF compiler performs common subexpression elimination, as described in sections 9.6 and 12.2. Three distinct kinds of calculations can be eliminated as redundant by the SELF compiler:

- loads and stores,
- arithmetic operations, and
- constants.

Three aspects of common subexpression elimination in the SELF compiler can improve performance. One is simply that computations are not repeated; this is the traditional benefit of common subexpression elimination. Another benefit, unique to the SELF compiler, is that any additional type information associated with the result of the original computation can be propagated to the result of the redundant computation, as described in section 9.6.2. For example, if the compiler eliminates a load instruction as redundant, then any type information known about the contents of the loaded memory cell can be propagated to the result of the eliminated load instruction. A third benefit, also unique to the SELF compiler, is that the corresponding array bounds check may be eliminated if a load or store to an array element may be eliminated. As described in section 9.6.2, this benefit exists primarily because symbolic range analysis is not implemented in the SELF compiler.

The charts on the following page display the impact of common subexpression elimination. It is easy to disable all three effects at once by simply not recording or checking for available values; results for this configuration are shown in columns labelled CSE. The individual contributions of the effects may also be measured by recording and checking for available values, but not taking advantage of a particular aspect of the information. The columns labelled CSE of Constants, CSE of Arithmetic Operations, and CSE of Memory References report the incremental effect of common subexpression elimination of only constants, arithmetic instructions, and load and store instructions, respectively. (The sum of the incremental effects of these three columns ideally should equal the incremental effect of CSE as a whole, shown in the first column.) CSE of Memory References includes the impact of three effects: eliminating memory reference instructions, propagating information about the types of the contents of memory cells, and eliminating some redundant bounds checks. The effects of these last two components are shown in columns labelled CSE of Memory Cell Type Information and CSE of Memory Cell Array Bounds Checking, respectively. (The difference between these last two columns and the CSE of Memory References column should be the incremental effect of just eliminating the memory reference instructions.) Propagated type information is ignored by introducing a new value where a memory load is replaced with an assignment node, with this value initially bound to the unknown type.

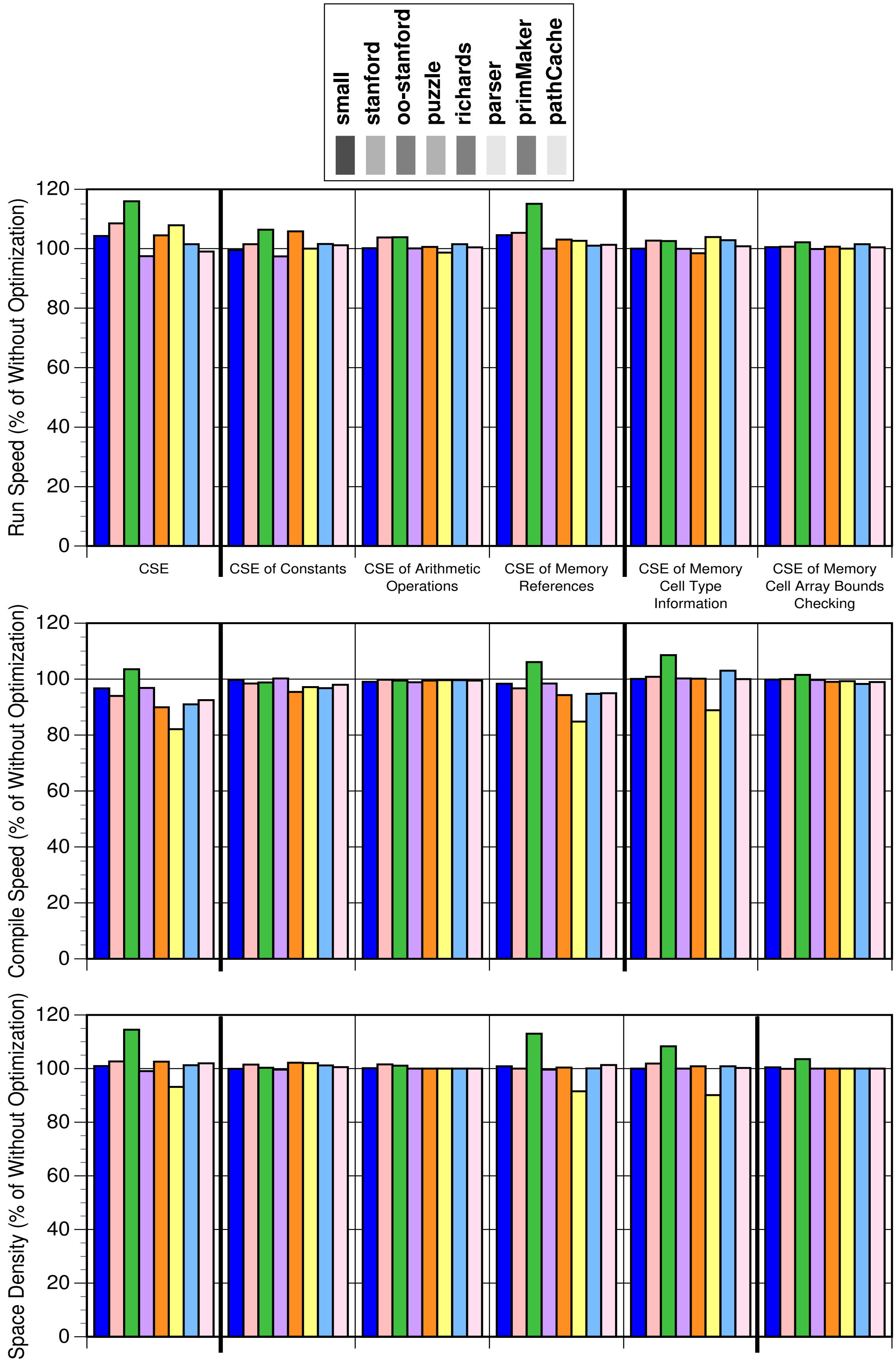
Common subexpression elimination has a relatively small effect on execution speed: less than a 15% performance improvement and occasionally a performance degradation. One cause of this poor performance is that common subexpression elimination of redundant constants is not implemented well in the current SELF system, as described in section 12.2; elimination of constants frequently replaces a two-instruction sequence (on the SPARC) that loads a 32-bit constant into a register with a one-instruction register move, instead of eliminating the loading of the constant entirely. In some cases, where the saved constant is allocated to a stack location instead of a register, performance can even slow down in the presence of common subexpression elimination of constants. Common subexpression elimination of arithmetic operations helps the Stanford integer benchmarks an average of 5%. Common subexpression elimination of memory loads and stores provides some of the best improvements, boosting the performance of the object-oriented versions of the Stanford integer benchmarks by nearly 15%.

Common subexpression elimination usually imposes a small penalty in compilation speed. The bulk of this extra cost in compilation time comes from common subexpression elimination of memory loads and stores, partially because of the extra work required to propagate the mapping from available memory cells to their contained values during type analysis and partially because of the additional inlining and other work enabled by the extra type information. Fortunately, where compilation speed drops, execution speed rises, so common subexpression elimination tends to pay for its cost in compile time with savings in execution time.

Common subexpression elimination usually saves a small amount of compiled code space, as would be expected. For the **`parser`** benchmark, however, the presence of common subexpression elimination *increases* the amount of code generated. This increase may be attributed to the extra inlining enabled by the type information available from common subexpression elimination of memory references.

B.3.11 Register Allocation

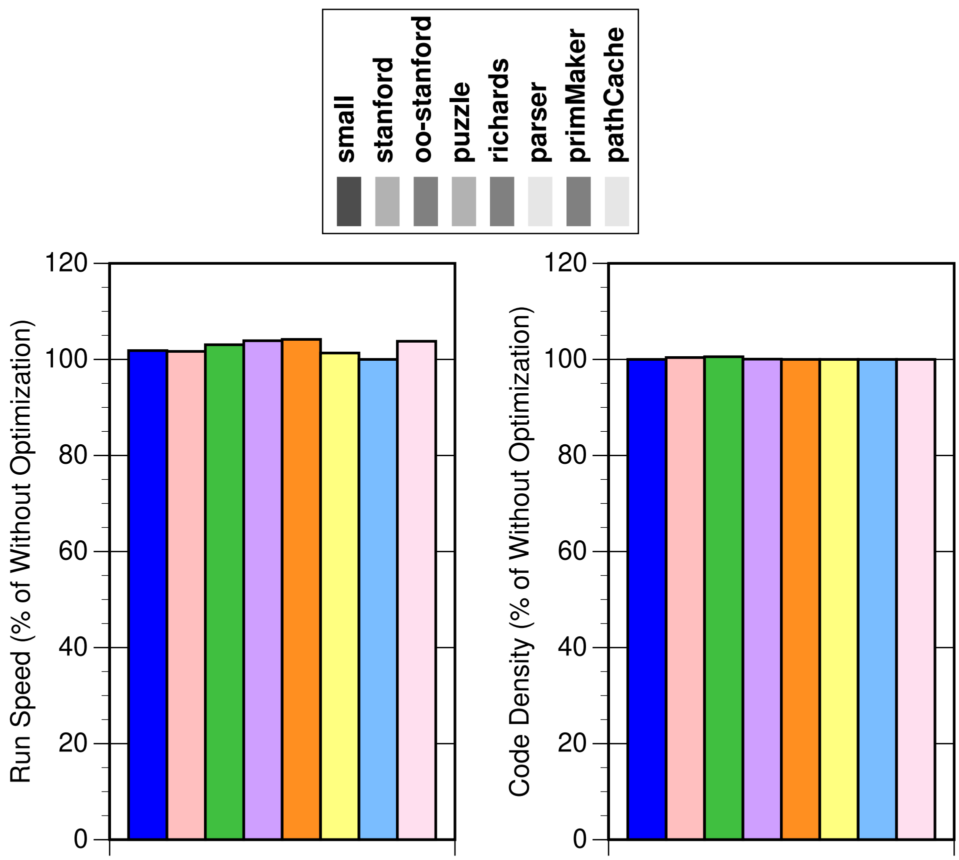
The SELF compiler maps names that are always aliases to variables, and then allocates run-time locations to these variables. This register allocation design was described in section 12.1. Unfortunately, we did not measure the effectiveness of this register allocation strategy.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Without Optimization)
Compile Speed (% of Without Optimization)
Space Density (% of Without Optimization)
0
20
40
60
80
100
120
CSE
CSE of Constants
CSE of Arithmetic Operations
CSE of Memory References
CSE of Memory Cell Type Information
CSE of Memory Cell Array Bounds Checking

B.3.12 Eliminating Unneeded Computations

The SELF compiler performs an additional pass over the control flow graph to eliminate unneeded computations, such as memory references and arithmetic operations whose results are never used, as described in section 12.3. Disabling this optimization would be easy except for two complications. Some of this pass is required for other parts of the SELF compiler to run correctly, such as eliminating unnecessary assignment nodes prior to constructing the variable lifetime conflict graph as part of register allocation. Also, other parts of the compiler expect some of this pass to be executed to get any sort of reasonable performance. For example, the initial type analysis phase expects that unnecessary loads of constants will be removed in this pass, and so the compiler feels free to insert such loads prior to message sends for variables that may later turn out to be the same constant at all sends. Since many variables are such constants, many loads are eliminated using this technique; without it some other technique would be used to eliminate these unnecessary loads.

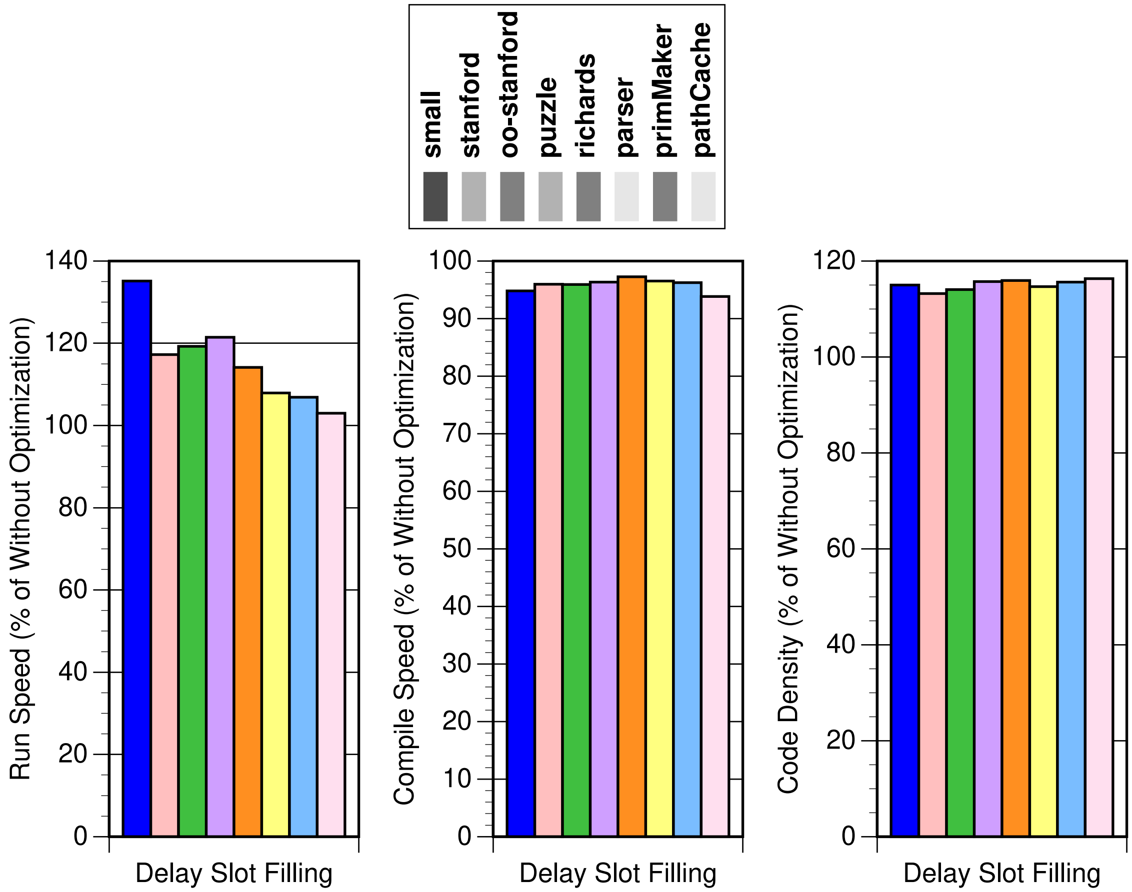
Therefore, to avoid exaggerating the importance of eliminating unneeded computations, a version of the SELF system was constructed that still eliminated assignments and loading of constants but did not eliminate unused arithmetic operations and memory loads. Compilation speed is relatively unaffected by this change, since the compiler still makes the pass to eliminate unnecessary constant loads and assignments. The following charts display the impact of eliminating unneeded arithmetic and memory loads on execution speed and compiled code space density.



Eliminating the unnecessary arithmetic and memory load instructions saves only a few percent of execution time. If eliminating unnecessary loading of constants were also disabled, the difference in execution speed would be much greater. Virtually no compiled code space is saved by this optimization, indicating that very few arithmetic or memory reference instructions were eliminated in these benchmarks.

B.3.13 Filling Delay Slots

As described in section 12.4, the SELF compiler fills delay slots on the SPARC to speed the generated code. The effect of delay slot filling on the performance of the system can be computed by constructing a version of the system than does not fill any delay slots, instead leaving a **nop** instruction in each delay slot. The following charts display the effect of delay slot filling.



Filling delay slots on the SPARC speeds up the benchmarks by between 5% and 35%, costs little in compile time, and reduces compiled code space requirements by almost 15%.* Given these execution speed and compiled code space benefits, delay slot filling seems to be worth its nominal compile time expense.

B.3.14 Summary of Effectiveness of the Techniques

Several techniques stand out as crucial to achieving good performance for SELF and similar languages. Inlining, deferred block creations, type prediction, and customization provide major improvements in performance. Value-based type analysis, exposed block analysis, splitting, lazy compilation of uncommon branches, and delay slot filling also make significant contributions to run-time performance. Other optimizations have more modest benefits.

* This number can be used to place an upper bound on the average size of a basic block in the SELF system. Assuming a 15% reduction in space, one out of every 7 instructions is a delay slot that can be filled. This gives an average basic block size for SELF of at most 6 instructions (after filling delay slots).

B.4 Splitting Strategies

Splitting is one of the most important techniques used in the SELF compiler, as well as one of the most complex to implement. Several different splitting strategies were devised and implemented in the SELF compiler. In this section we explore the effectiveness of these techniques.

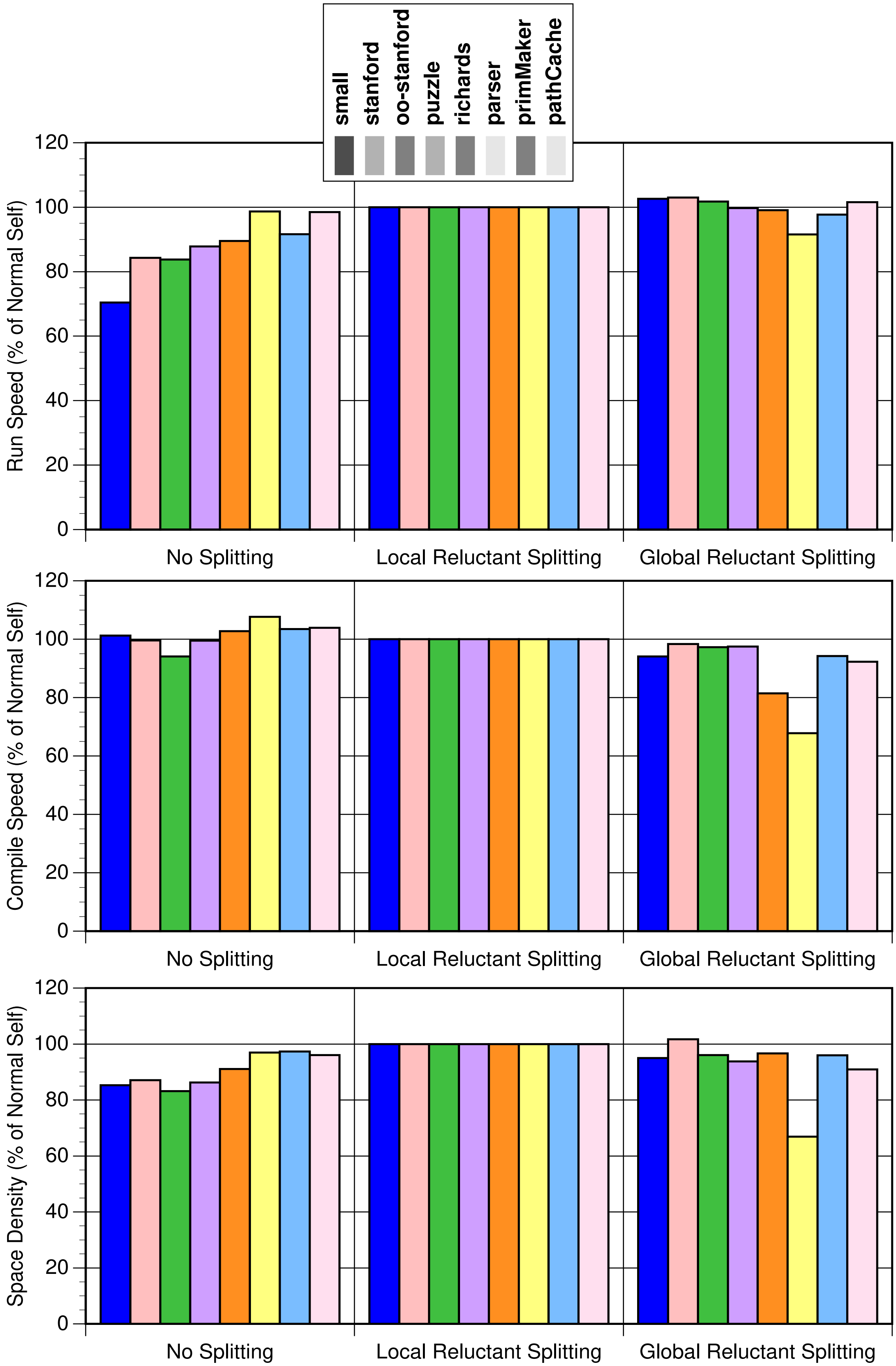
Splitting strategies can be broken down into three main approaches: no splitting, reluctant splitting (described in section 10.2), and eager splitting (described in section 10.3). Reluctant splitting itself can be divided into local reluctant splitting and global reluctant splitting, depending on how much code the compiler is willing to duplicate for a split. Eager splitting can be combined with one of three tail merging strategies: no tail merging, tail merging based on forward-computed type information, and tail merging based on reverse-computed requirements. Each of these six base splitting strategies can be combined with divided splitting (described in section 10.4) and/or lazy compilation of uncommon branches (described in section 10.5). The next few subsections explore the effectiveness of these various combinations. All measurements are reported relative to the performance of the standard SELF splitting configuration: local reluctant splitting with lazy compilation of uncommon branches but without divided splitting.

B.4.1 Reluctant Splitting Strategies

The charts on the following page illustrate the effectiveness of local reluctant splitting and global reluctant splitting in comparison to a configuration that did no splitting. These results include the effect of lazy compilation of uncommon branches. All results are relative to the performance of the standard configuration: local reluctant splitting with lazy compilation of uncommon branches.

Local reluctant splitting is clearly the best overall strategy, performing well in execution speed, compilation speed, and compiled code density. Performing no splitting at all sometimes saves some compile-time but usually incurs a significant run-time speed penalty and consumes additional compiled code space. Local reluctant splitting compares favorably to no splitting in compilation speed and code density because many of the splits performed as part of local reluctant splitting separate **`true`** results from **`false`** results after comparisons and in control structures, and performing this kind of splitting significantly *simplifies* the control flow graph (as illustrated in section 10.1), thus paying for itself through later savings in compile time and compiled code space.

Global reluctant splitting does not compensate for its larger compile time and compiled code space costs with significantly improved execution speed. In these benchmarks (and perhaps in most programs), there typically is either just a single common-case path, in which case global reluctant splitting is not needed, or a pair of common-case paths, one for a **`true`** result and one for a **`false`** result. For the **`true`** and **`false`** cases, the result of the comparison or boolean function that produced the two paths is usually consumed by the immediately following message, such as an **`ifTrue:`** message, and so local reluctant splitting is adequate to handle this kind of situation. Thus, for these benchmarks, global reluctant splitting provides unneeded extra power. As we will argue in the next subsection, global reluctant splitting would be more important if lazy compilation of uncommon branches had not been devised; lazy compilation splits off the uncommon-case paths that would have been split off, albeit less effectively, by global reluctant splitting. Even though global reluctant splitting does not seem useful in today's system with local reluctant splitting and lazy compilation, in the future multiple common-case paths may be more frequent as other implementation techniques are incorporated [HCU91], and global reluctant splitting may then begin to show significant improvement over local reluctant splitting.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
120
100
80
60
40
20
0
No Splitting
Local Reluctant Splitting
Global Reluctant Splitting

B.4.2 Lazy Compilation of Uncommon Branches

The charts on the following page report the performance of various reluctant splitting strategies both with and without lazy compilation of uncommon branches. All results are relative to the performance of the standard configuration: local reluctant splitting with lazy compilation of uncommon branches.

Lazy compilation of uncommon cases improves execution performance for almost all splitting configurations and benchmarks, as can be seen by comparing the results for each ... (Not Lazy) column to the results for the corresponding ... (Lazy) column. This speed-up may be attributed to two major factors: providing the effect of divided splitting and simplifying the register allocation problem. The largest effects occur for no splitting and local reluctant splitting; these two strategies benefit the most from the additional divided splitting effect. All strategies benefit from better register allocation enabled by lazy compilation.

The largest impact on execution speed occurs for the more numerically-oriented benchmarks. This is as expected, since in these benchmarks many messages get inlined down to primitives that introduce primitive failure uncommon branches (such as **+** messages getting inlined down to **`_IntAdd:`** primitive calls, each of which may fail with an overflow error), and many type prediction type tests get inserted (such as integer type tests before **+** messages) that each create an uncommon branch. The larger number of uncommon branch entries in these benchmarks provides more opportunities for lazy compilation.

Lazy compilation makes a dramatic improvement in both compilation speed and compiled code space efficiency, across the board for all base splitting strategies and benchmarks. As expected, lazy compilation improves performance most for the smaller, more numerical benchmarks.

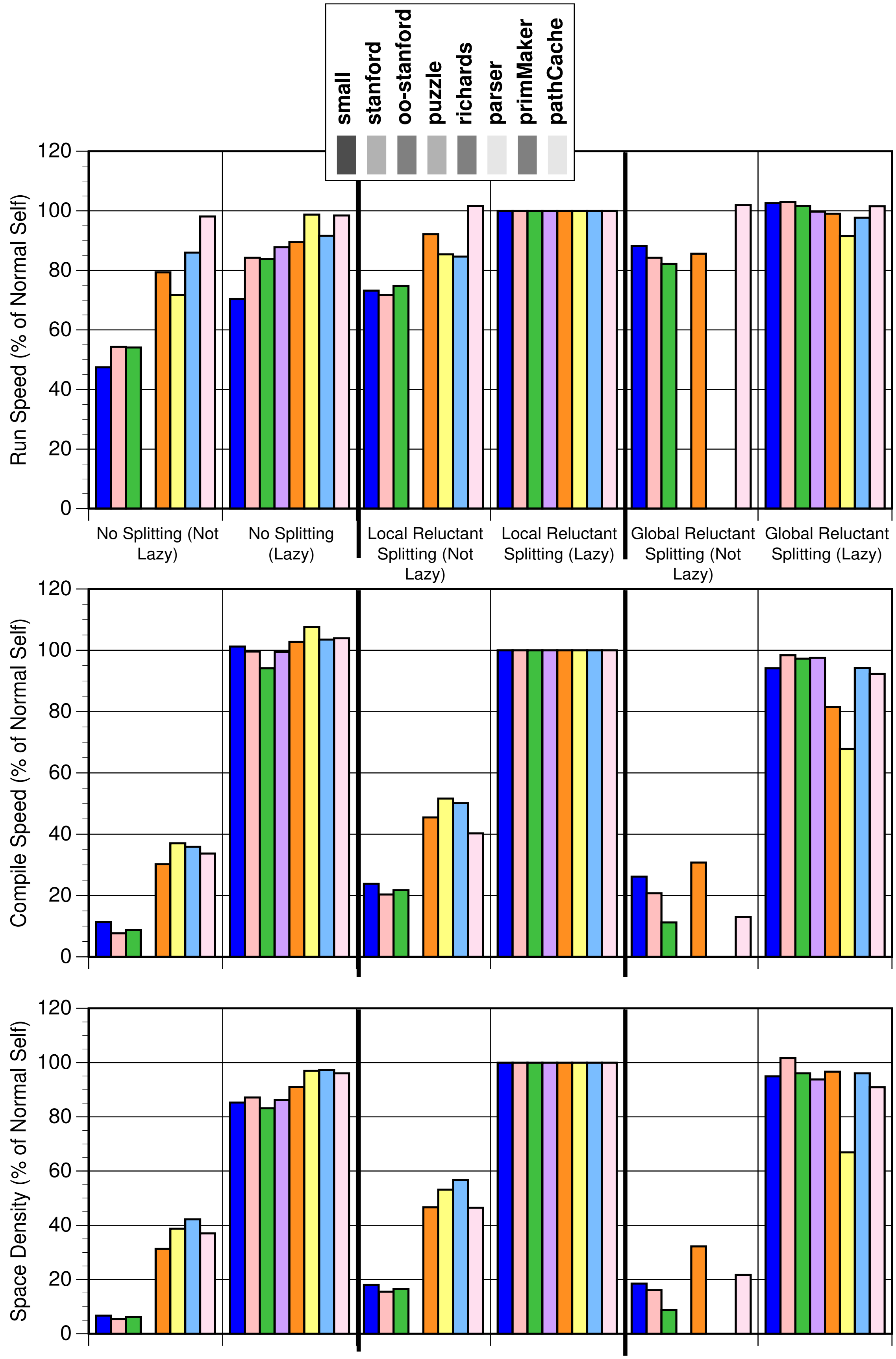
Lazy compilation of uncommon cases obviates much of the need for more sophisticated base splitting techniques such as global reluctant splitting. If neither lazy compilation nor divided splitting were implemented, then global reluctant splitting would offer performance improvements over local reluctant splitting, as can be seen by comparing the Local Reluctant Splitting (Not Lazy) results to the Global Reluctant Splitting (Not Lazy) results. However, since in the present system only one common case path normally is generated, global reluctant splitting does not provide much additional functionality.

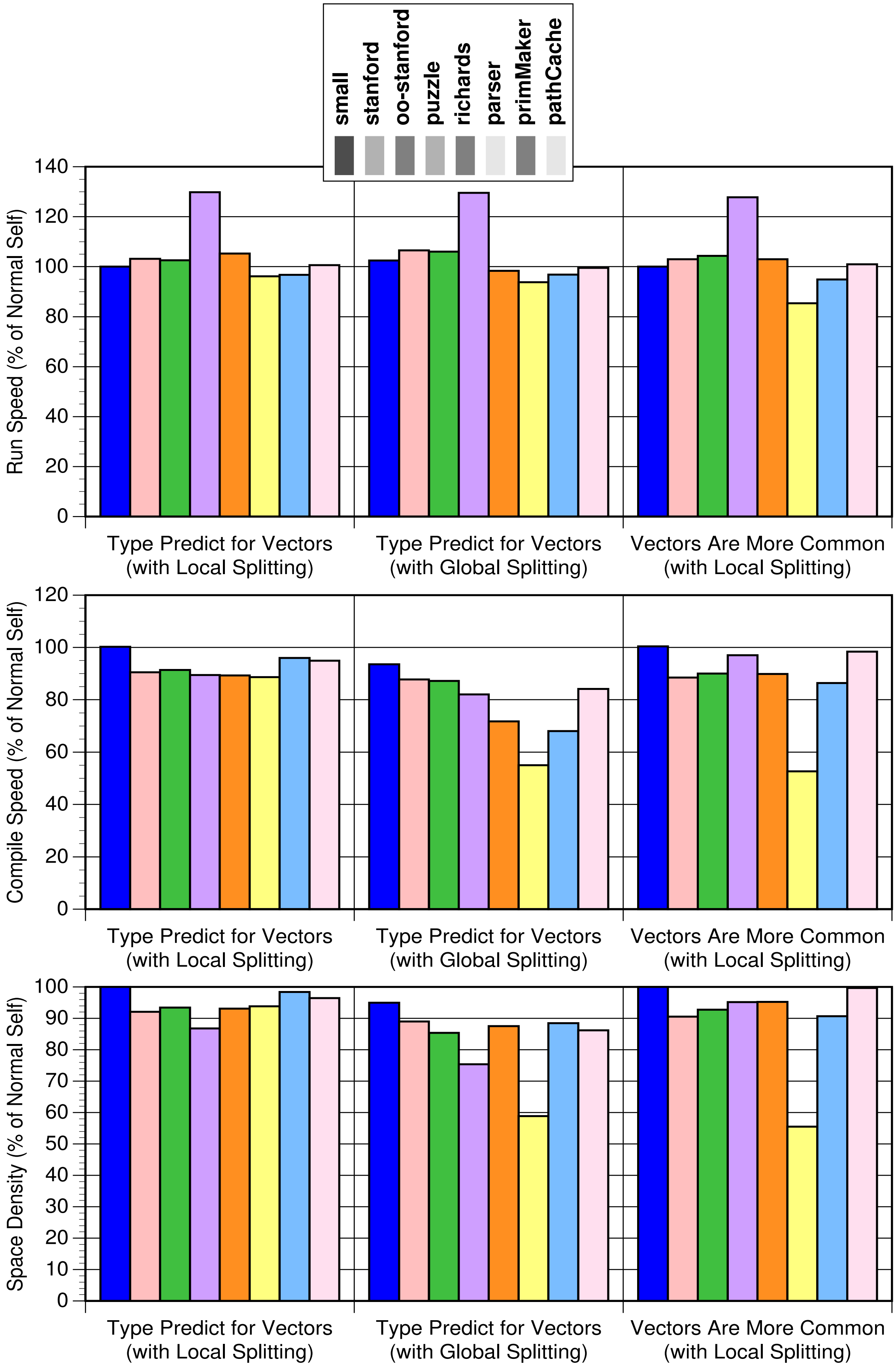
B.4.3 Vector Type Prediction

When configuring the SELF system, one choice that must be made (today at least) is the set of message names and corresponding receiver types that are to be type-predicted. For most messages, the choice is fairly obvious: predict **`true`** or **`false`** for the receiver of **`ifTrue:`** and predict integer for the receiver of **+**, for instance. Unfortunately, some choices are not so clear-cut. For example, what should the compiler predict for the **=** message? Currently, the compiler predicts that the receiver often is an integer (and consequently inserts a type-test for an integer receiver), but since many kinds of objects other than integers are compared for equality, the non-integer branch is not treated as uncommon but instead gives rise to a second common-case path.

The compiler faces a similar problem for messages like **`at:`** and **`at:Put:`**. For example, the compiler could predict that the receiver of these messages is likely to be the built-in fixed-length vector type. All the arrays manipulated by the micro-benchmarks and the Stanford integer benchmarks are of this type, so this prediction should improve the performance of these benchmarks. However, in other parts of the system, the receivers of **`at:`** and **`at:Put:`** are more likely to be dictionaries or other kinds of keyed collections, not simple fixed-length vectors. The current standard SELF configuration does no type prediction for **`at:`** and **`at:Put:`**, in consideration of the more object-oriented programs currently being run in the SELF system, but other kinds of programs (including the benchmarks) might benefit from type predicting fixed-length vector receivers for **`at:`** and **`at:Put:`**.

To determine the improvement that could be gained from vector type prediction, we measured the performance of three additional configurations. We extended the standard configuration based on local reluctant splitting to additionally type-predict for vectors; one version treats non-vector cases as uncommon, while another treats non-vector cases as giving rise to a second common-case path. Since global reluctant splitting should shine in the presence of multiple common-case paths, we also measured a configuration based on global reluctant splitting which type-predicts for vectors and treats non-vector cases as additional common-case paths. The charts on page 195 report the performance of these three vector type prediction configurations relative to that of the standard configuration based on local reluctant splitting and no vector type prediction.


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
0
20
40
60
80
100
120
No Splitting (Not Lazy)
No Splitting (Lazy)
Local Reluctant Splitting (Not Lazy)
Local Reluctant Splitting (Lazy)
Global Reluctant Splitting (Not Lazy)
Global Reluctant Splitting (Lazy)


small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
Type Predict for Vectors (with Local Splitting)
Type Predict for Vectors (with Global Splitting)
Vectors Are More Common (with Local Splitting)

Type prediction for vectors improves performance for some of the benchmarks. Most individual benchmarks are unchanged, but a few benchmarks such as **`intmm`**, **`oo-intmm`**, and **`puzzle`** improve by between 20% and 30% (raw data may be found in Appendix C). Treating non-vector cases as uncommon only makes a difference on the **`oo-intmm`** benchmark, boosting its performance another 10%. Global reluctant splitting is just as effective as treating non-vectors as uncommon, verifying that global reluctant splitting can provide improved performance in the presence of multiple common-case paths.

Unfortunately, type predicting for vectors slows down the larger, more object-oriented benchmarks in which the predictions are always wrong. When non-vector receivers are considered uncommon, mispredicting forces the compiler to generate less optimized uncommon branch extensions, further slowing down execution performance; the **`parser`** benchmark runs 15% slower in this mode than in the normal configuration. Since we expect most SELF programs to be more like **`parser`** than **`puzzle`**, the normal SELF system does not type-predict for vectors. However, in environments or applications where vector receivers were more common, vector type prediction could significantly improve performance. Developing a system in which some parts type-predict for vectors while others do not is an active area of current research [HCU91].

Type predicting for vectors uniformly slows down compilation. Usually this slow-down is less than 20%, but when mispredictions force the compiler to generate uncommon branch extension methods (when non-vector cases are considered uncommon), compile times can be twice as long as with the normal configuration. In fact, these potentially lengthy compile pauses are the primary reason we currently do not type-predict vector receivers.

Not surprisingly, vector type prediction takes up more compiled code space, but usually not more than 10% over the non-predicting configuration. When mispredictions and lazy compilation cause uncommon branch extensions to be created, however, code space consumption can double.

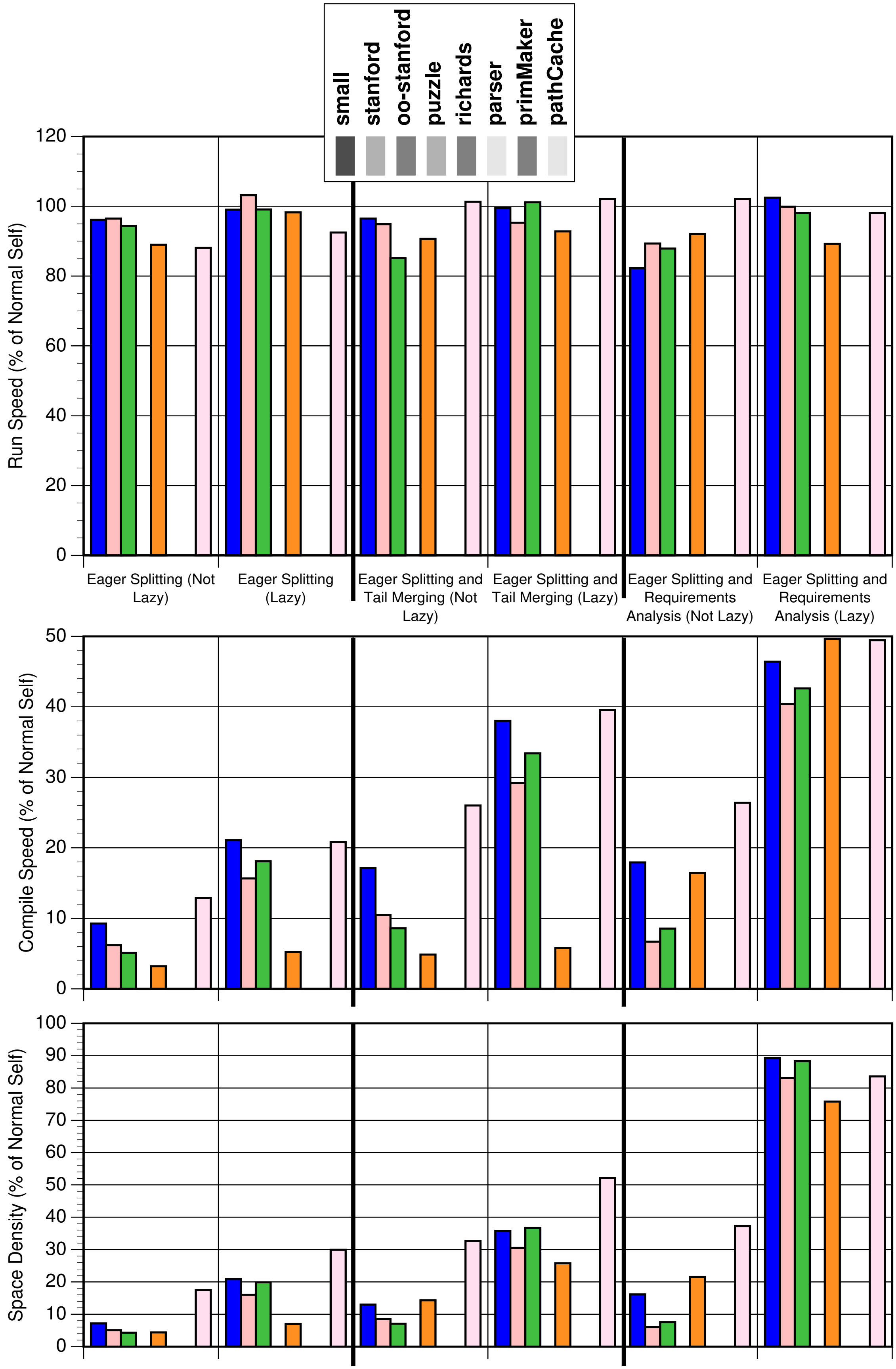
B.4.4 Eager Splitting Strategies

The charts on the following page report execution performance for the various eager splitting configurations relative to that of the standard configuration, local reluctant splitting with lazy compilation of uncommon branches.

The various tail merging strategies are designed to save compile time and compiled code space while producing the same execution speed. These expectations are borne out by the results. Different tail merging strategies achieve roughly the same execution speeds, but requirements-analysis-based tail merging, the most sophisticated tail merging technique, achieves much better compilation speeds and compiled code densities than the simpler techniques; the extra reverse pass over the control flow graph that is part of requirements analysis easily pays for itself in reduced overall compile time.

Lazy compilation slightly improves the execution performance of eager splitting. Since eager splitting already incorporates divided splitting, this potential benefit of lazy compilation is unneeded. Compilation speed improves more dramatically with lazy compilation of uncommon cases. This speed-up becomes more pronounced as the tail merging strategy becomes more sophisticated. Lazy compilation makes less of an impact for the simpler tail merging strategies since those strategies spend more of their compile time compiling common-case parts of the control flow graph, and so there is less compile time spent on uncommon branches to be eliminated. Similarly, lazy compilation dramatically improves compiled code space efficiency, especially for eager splitting with requirements-based tail merging.

Unfortunately, overall eager splitting provides no execution speed benefits compared to reluctant splitting and at best more than doubles the compilation time costs; in some cases the benchmarks could not even be compiled using eager splitting because they would have needed too much compiler temporary data space. Clearly, eager splitting is not practical as currently implemented.



small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
120
100
80
60
40
20
0
Eager Splitting (Not Lazy)
Eager Splitting (Lazy)
Eager Splitting and Tail Merging (Not Lazy)
Eager Splitting and Tail Merging (Lazy)
Eager Splitting and Requirements Analysis (Not Lazy)
Eager Splitting and Requirements Analysis (Lazy)
Compile Speed (% of Normal Self)
50
40
30
20
10
0
Space Density (% of Normal Self)
100
90
80
70
60
50
40
30
20
10
0

B.4.5 Divided Splitting

The charts on the following page report the performance of various reluctant splitting strategies both with and without divided splitting. All results are relative to the performance of the standard configuration: local reluctant splitting with lazy compilation of uncommon branches and without divided splitting.

Divided splitting makes no difference when used in conjunction with lazy compilation of uncommon branches, as can be seen by comparing the results for columns labelled ... (Lazy) to the results for columns labelled Divided ... (Lazy). This is as expected, since lazy compilation achieves the same beneficial effect on type analysis as does divided splitting but in a more efficient manner. For the strategies without lazy compilation, divided splitting speeds execution significantly, as can be seen by comparing the results for columns labelled ... (Not Lazy) to the results for columns labelled Divided ... (Not Lazy). The advantages provided by divided splitting decrease with the sophistication of the base splitting technique. Divided splitting usually slows compilation and consumes more compiled code space, as expected.

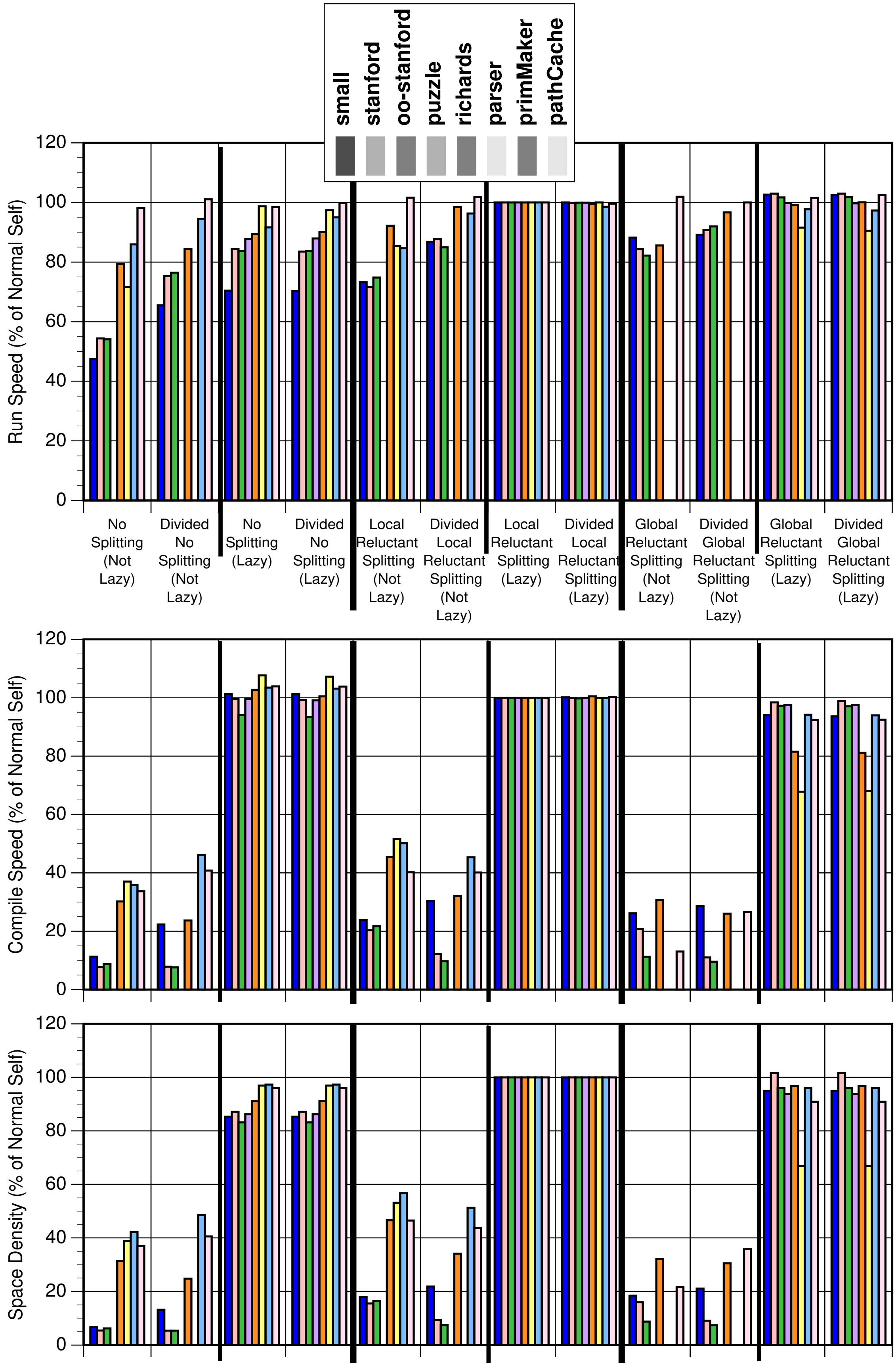
Based on these results, divided splitting provided no additional benefits over lazy compilation of uncommon cases. However, in environments where lazy compilation of uncommon cases was impossible, divided splitting can offer improved performance.

B.4.6 Summary of Splitting Strategies

Local reluctant splitting combined with lazy compilation of uncommon cases is the most effective splitting strategy, optimizing all of execution speed, compilation speed, and compiled code space efficiency. Lazy compilation provides dramatic, near order-of-magnitude improvements in compilation speed and code density, and even boosts execution speed and obviates the need for divided splitting. Global reluctant splitting provides slight improvements in performance for these benchmarks but at a noticeable decrease in compilation speed. Eager splitting as presently implemented provides no significant performance improvements and sacrifices at least half of the compilation speed of local reluctant splitting. Vector type prediction can improve performance for some benchmarks but slows down other benchmarks.

In the chart on page 200, we summarize the effectiveness of various splitting strategies and their trade-offs between execution speed and compilation speed by plotting the execution speed and compilation speed of several splitting strategies on a two-dimensional scatter chart; relative compiled code space efficiency is very nearly directly proportional to relative compilation speed so a third dimension is not necessary. To make the chart more readable, only selected configurations are included.

The cluster of strategies in the upper-right corner of the chart (in the “good” region) all use reluctant splitting and lazy compilation of uncommon branches. The other strategies produce no better execution speeds with poor compilation speeds.



small
stanford
oo-stanford
puzzle
richards
parser
primMaker
pathCache
Run Speed (% of Normal Self)
Compile Speed (% of Normal Self)
Space Density (% of Normal Self)
0
20
40
60
80
100
120
No Splitting (Not Lazy)
Divided No Splitting (Not Lazy)
No Splitting (Lazy)
Divided No Splitting (Lazy)
Local Reluctant Splitting (Not Lazy)
Divided Local Reluctant Splitting (Not Lazy)
Local Reluctant Splitting (Lazy)
Divided Local Reluctant Splitting (Lazy)
Global Reluctant Splitting (Not Lazy)
Divided Global Reluctant Splitting (Not Lazy)
Global Reluctant Splitting (Lazy)
Divided Global Reluctant Splitting (Lazy)

paste chart-page-200.ps here

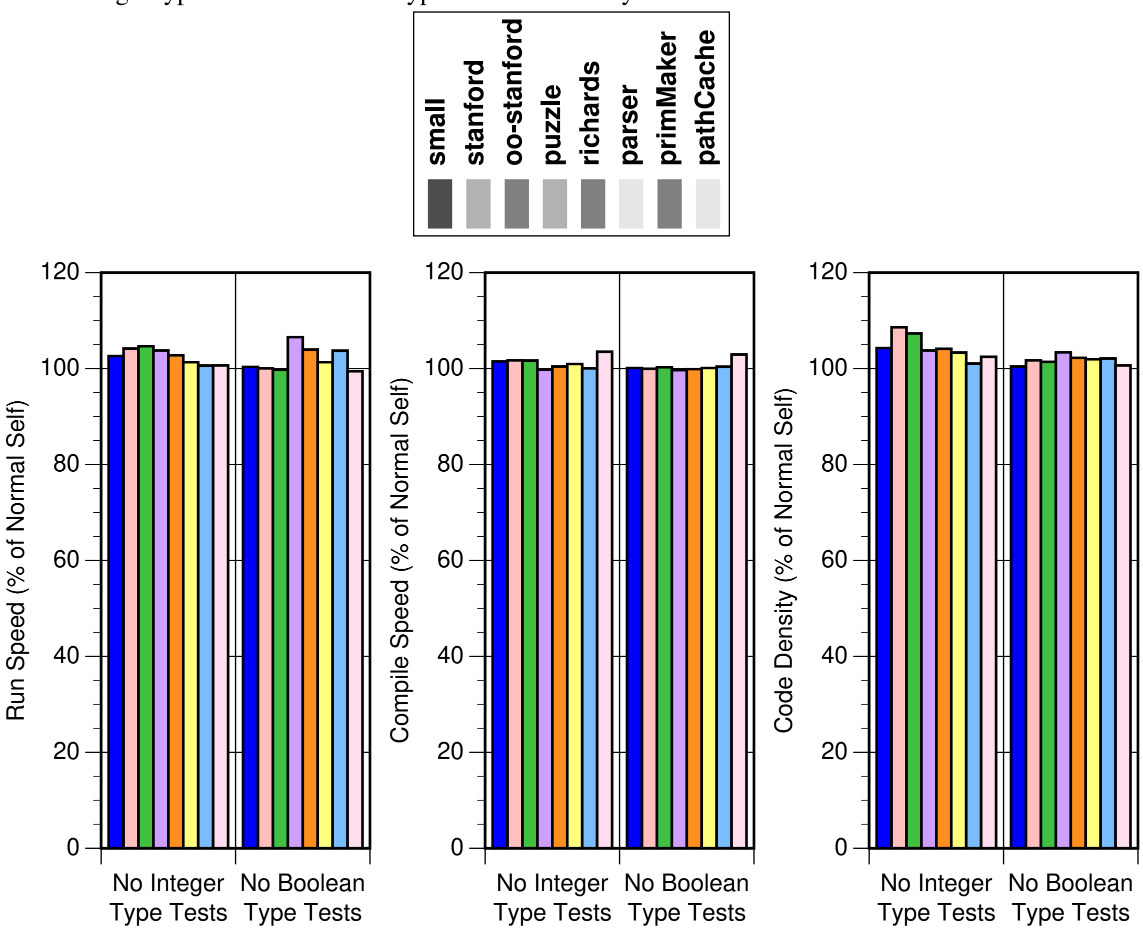
B.5 Some Remaining Sources of Overhead

This section explores in detail each of the remaining sources of overhead that we were able to measure in the SELF system that are not present in a traditional optimized language implementation. These results were summarized in section 14.4.

B.5.1 Type Tests

The SELF compiler inserts extra run-time type tests that are not present in the output of the optimizing C compiler. These tests are inserted as part of type prediction for messages like `+` and `ifTrue:` and as part of type-checking of arguments to primitives. Language features like message passing, dynamic typing, generic arithmetic, and safe primitives, present in SELF but not in C, incur this extra overhead.

We can measure the cost of these run-time type tests by constructing a version of SELF that does not generate any type checks but instead assumes that the type tests would always succeed. This configuration obviously is unsafe, but none of the benchmarks happen to fail any type tests and so do not break under this configuration. The charts below report the costs in execution speed, compilation speed, and compiled code space for run-time type tests and checks; we report the costs of integer type tests and boolean type tests individually.

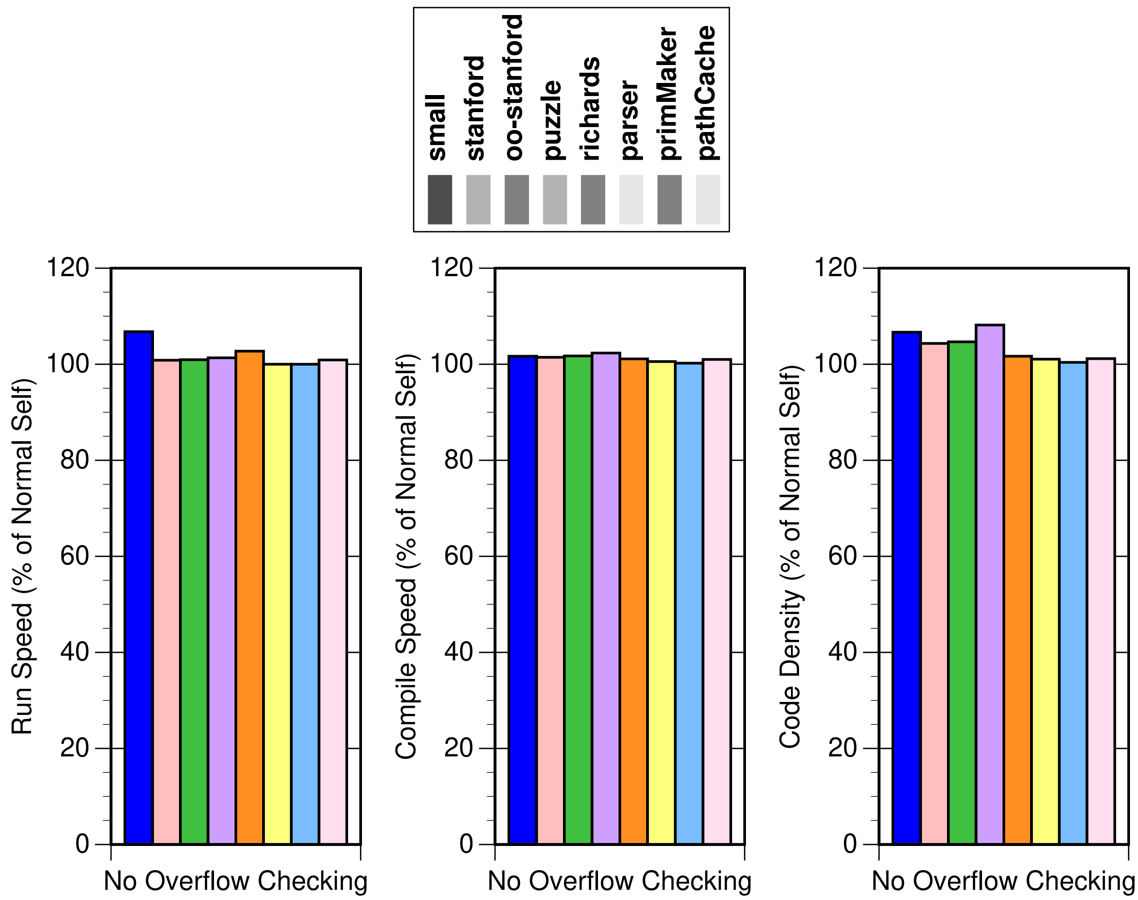


Type tests increase execution time by no more than 10% for these benchmarks, take up little extra compile time, and add no more than 10% additional compiled code space. Type analysis, splitting, and lazy compilation of uncommon cases are largely responsible for the relatively low number of type tests that remain in compiled SELF code.

B.5.2 Overflow Checking

The SELF compiler sometimes generates an overflow check after an integer arithmetic instruction to check for primitive failure of the corresponding integer arithmetic primitives. By handling this primitive failure, SELF programs can (and do) support generic arithmetic. Generic arithmetic traditionally has been an expensive language feature; section 14.2 showed that the performance of T programs that use generic arithmetic can be half that of T programs that avoid generic arithmetic. Much of the added cost of generic arithmetic involves the extra type tests associated with checking for integer arguments to generic arithmetic operations; this overhead was measured in the previous section. The remaining cost of generic arithmetic is incurred by the extra overflow checks which are not generated by the integer-specific arithmetic supported by traditional languages.

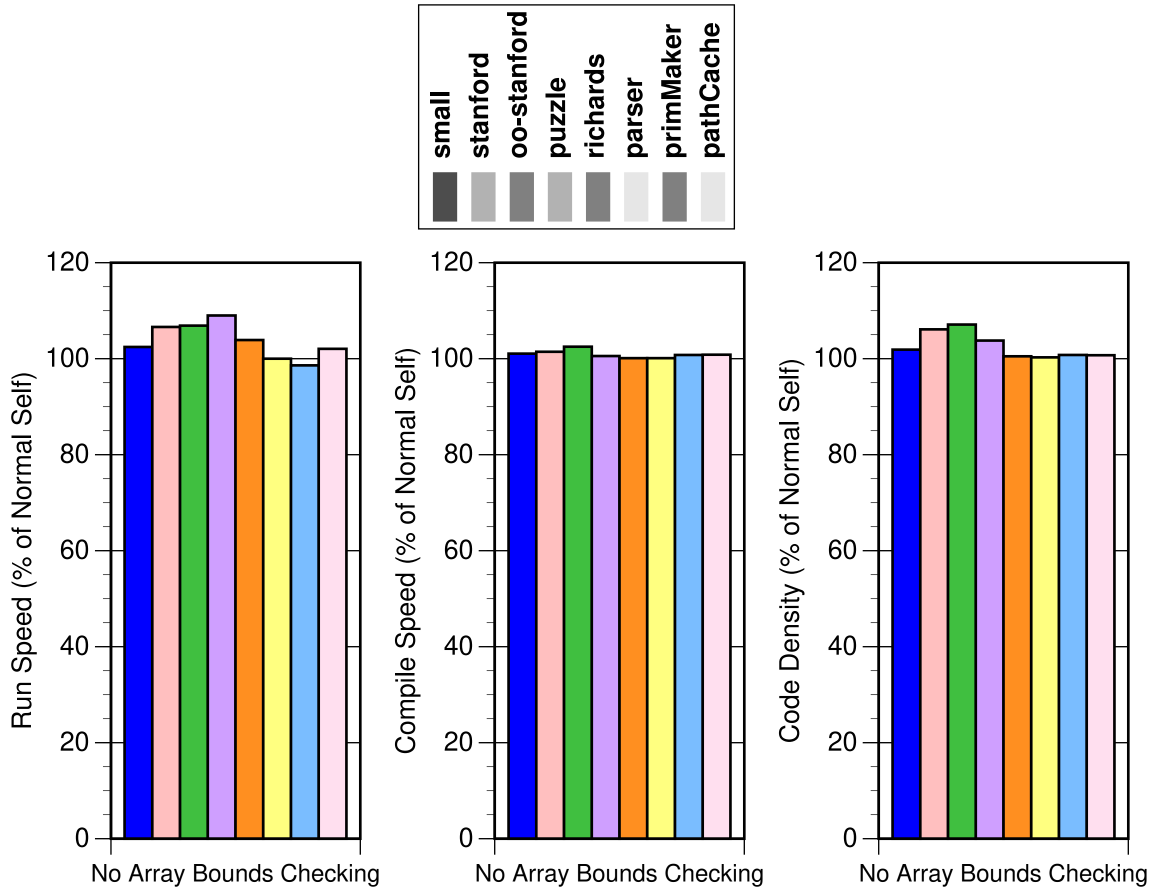
We can measure the cost of overflow checking in SELF simply by not checking for overflows. Again, this configuration is unsafe, but none of the benchmarks overflow any arithmetic operations. The charts below report the cost of overflow checking.



Overflow checking slows execution by less than 5% for all but the smallest benchmarks. However, since the register allocator frequently leaves an extra register move behind even when overflow checks are removed, the cost of overflow checking may be understated by these results by up to a factor of two. Overflow checking imposes negligible compile-time cost and a small space cost.

B.5.3 Array Bounds Checking

SELF array accessing primitives always verify that accesses lie within the bounds of the array. We can measure the cost of array bounds checking in SELF by not checking for access out-of-bounds in the generated code. The following charts display the cost of array bounds checking.



Array bounds checking imposes a modest run-time performance cost, between 5% and 10% overhead for those benchmarks manipulating arrays. We think that much of this overhead could be eliminated by applying more sophisticated integer subrange analysis using symbolic bounds, but the complexity of this technique may not be worth the apparently modest improvement in execution time. Very little compile time is used for generating code to check for out-of-bounds array accesses. Some compiled code space is required to support array bounds checking in the SELF implementation, but this space overhead is less than 10%. The cost would be much higher if the compiler did not compile uncommon cases lazily.

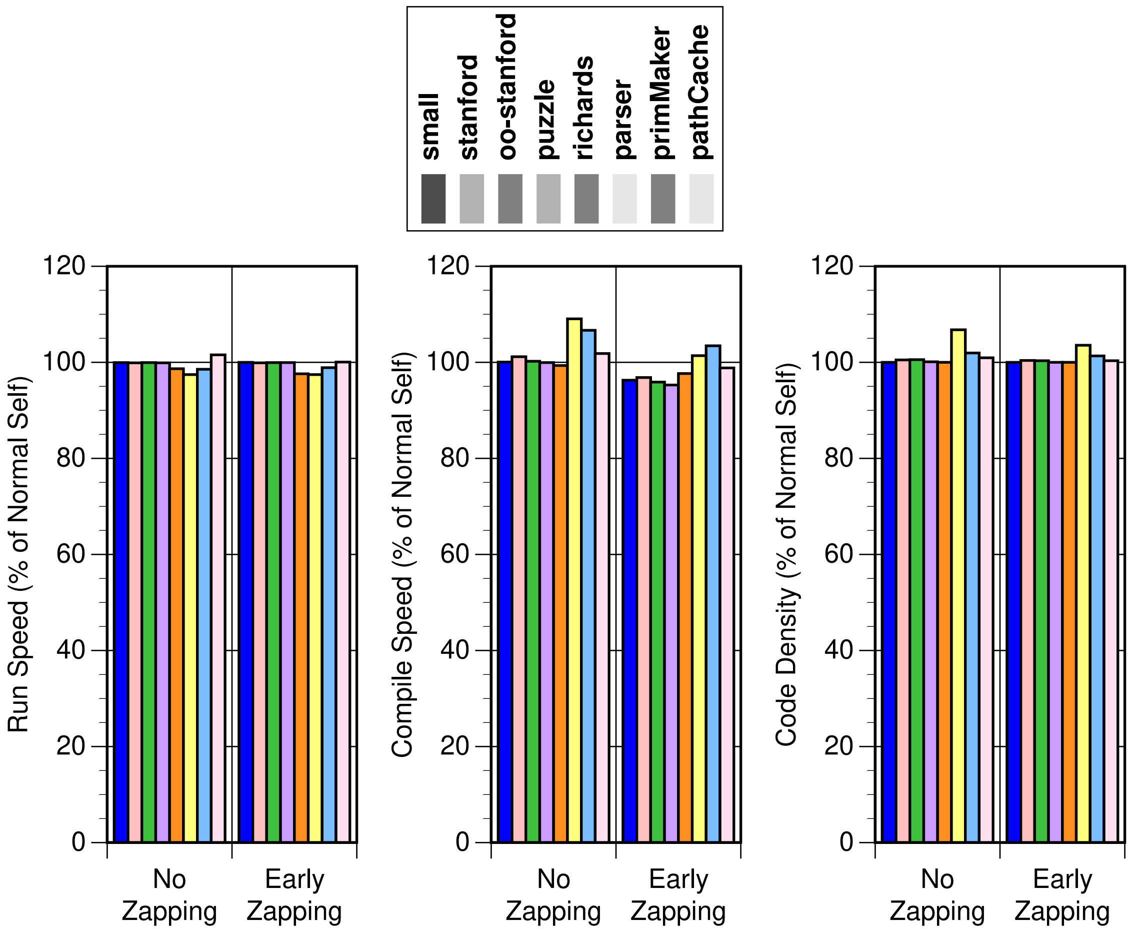
B.5.4 Block Zapping

The current SELF implementation does not allow a block to be invoked after its lexically-enclosing activation record has returned. To enforce this restriction, the compiler generates extra code that "zaps" blocks once they become uninvokable, as described in section 6.3.2. This zapping cost could become fairly expensive. Block zapping involves both extra run-time code needed to zero the frame pointer of any block that might have been created and extra registers or stack locations to hold the blocks until they are used by the zap code. Since block **`value`** methods themselves do not require explicit run-time code to test for a zero frame pointer, instead relying on the machine's addressing and protection hardware to trap references to illegal addresses, this zapping architecture imposes no run-time overhead on block invocation.

To gauge the cost of this design, we measured three configurations of the SELF system, each with a different rule for zapping blocks:

- no zapping,
- early zapping (block lifetimes extend to the end of the message in which they are initially an argument), and
- late zapping (block lifetimes extend to the end of the scope in which they are created); this is the standard configuration.

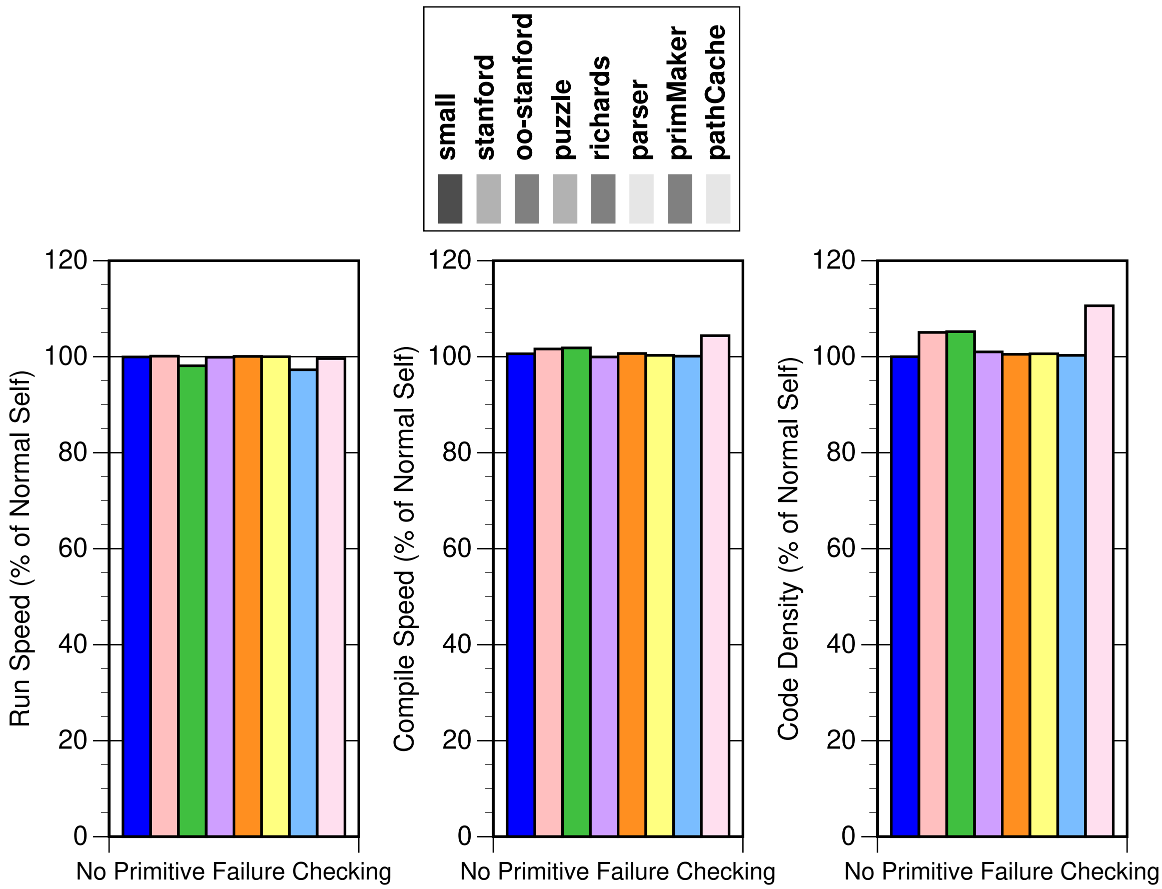
Early zapping was proposed as an alternative to late zapping that might be less expensive, particularly in terms of register usage. The following charts report the performance of no zapping and early zapping relative to late zapping, the standard configuration.



Surprisingly, block zapping has a negligible impact on execution speed of these benchmarks. The slight slow-downs for some of the benchmarks may be caused by unlucky interactions with other parts of the system; performance should only improve with these alternative zapping implementations. As expected, the SELF compiler is faster when it does not bother generating any zap code. Early zapping slows down the compiler relative to late zapping; the data structures and algorithms used by the compiler to generate the early zapping code are more complex than those for late zapping. Both no zapping and early zapping have slightly better compiled code space efficiencies than late zapping, but by a few percent at most. Based on these results, the safety and greater flexibility of late zapping make it the preferable block zapping strategy. Some future implementation of SELF might support true non-LIFO blocks (fully upward closures) and dispense entirely with the need for block zapping, simplifying this part of the compiler's implementation in the process.

B.5.5 Primitive Failure Checking

If the SELF compiler implements a primitive by calling a routine in the virtual machine (rather than generating special inlined code for the primitive), the compiler generates code after the primitive call that checks for the primitive's failure by testing for a special return value. This run-time overhead could be avoided by an alternate calling convention for such primitives that used different return offsets for successful returns and failing returns. To determine whether this change would be useful, we need to know the cost of the current design. We can compute this cost simply by not checking for primitive failures, assuming that all such primitives succeed. The following charts report the cost of external primitive failure checking as currently implemented.

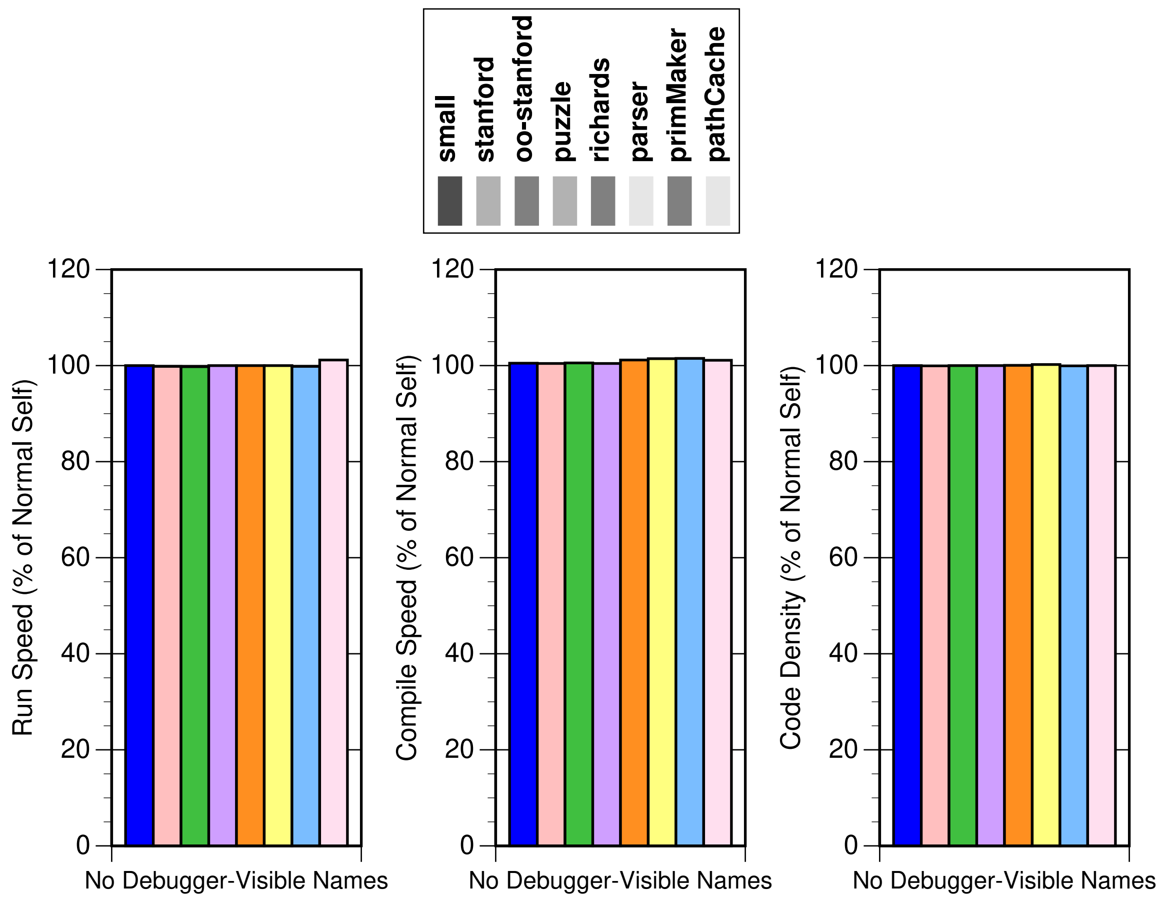


Primitive failure checking for non-inlined primitives has no effect on execution performance for these benchmarks. The slight slow-down for the **`oo-stanford`** and **`primMaker`** benchmarks may be unfortunate interactions with the register allocator, since removing the run-time check should only improve performance. Primitive failure checking imposes a slight cost in compile time and increases space costs by up to 10%. These costs do not seem significant enough to justify optimizing the return sequence of external primitives.

B.5.6 Debugger-Visible Names

In section 5.1 we argued that a language implementation should support debugging of the entire program at the level of the source code, with all optimizations and other implementation artifacts hidden from the programmer. This requirement restricts the kinds of optimizations that can be performed, thus possibly degrading performance over a system that did not support source-level debugging.

Some of these costs cannot be measured easily in our system, such as the cost of not performing tail call elimination or of not reordering code. Fortunately, at least one of the costs attributable to the need for debugging can be measured. In the SELF system, the programmer can get a complete view of the state of a suspended process, including the values of all data slots in activation records (i.e., the contents of local variables and arguments of source-level stack frames), as described in section 13.1. This requires that the compiler ensure that the contents of a variable is always up-to-date and available as long as the debugger might access it, even if the compiled code has no more need of the variable. This imposes a cost in terms of registers that are used and cannot be reallocated to other expressions, possibly causing some expressions to be spilled out to the stack. We measured the cost of this variable lifetime extension by configuring the compiler to ignore the effect of the debugger when computing the lifetime of variables, freeing up registers as soon as the compiled code has no more use of the variable, and report the effects in the charts below.

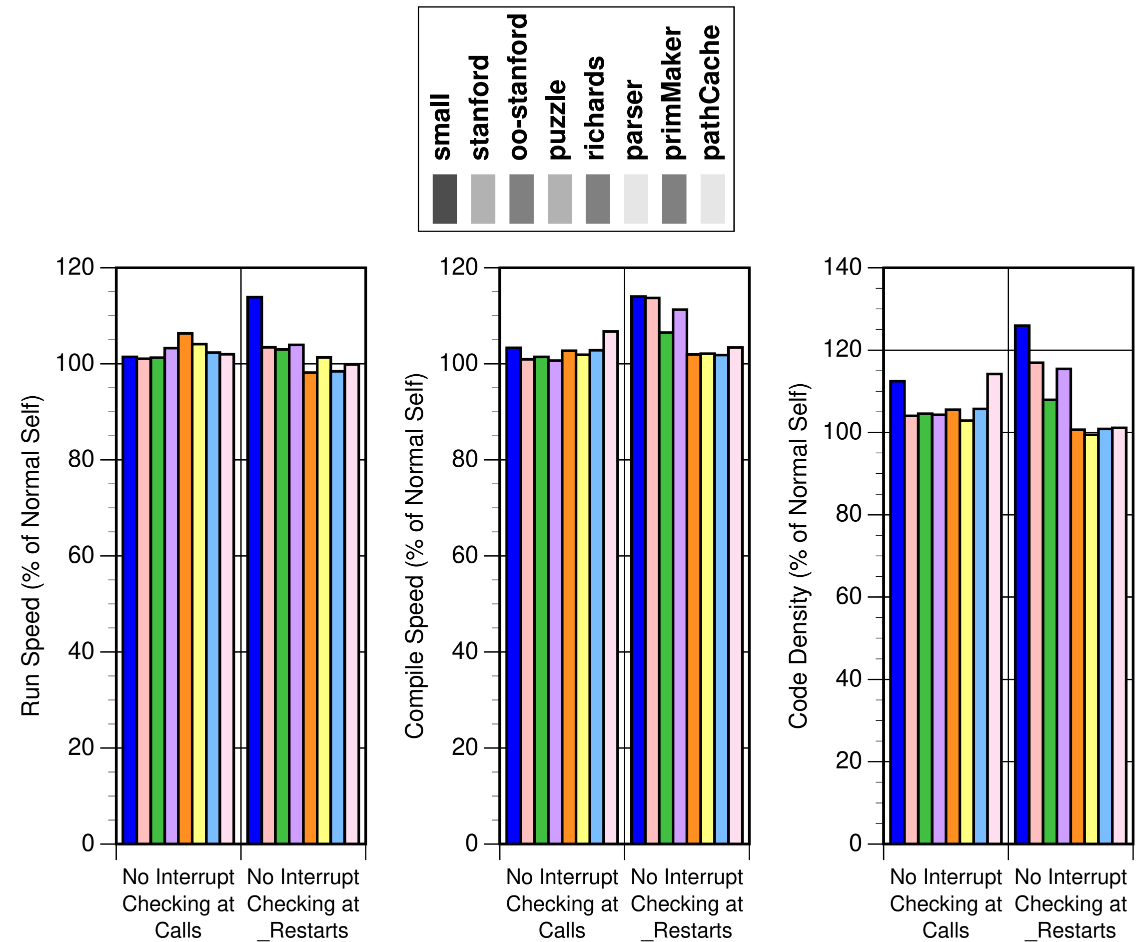


Supporting the debugger's ability to view all source-level visible names whether or not the executing code requires a name imposes no execution-time cost for these benchmarks. This is partially attributable to the presence of hardware register windows on the SPARC (since leaving an unused variable in a register is free while saving and restoring an unused variable across calls is not), partially attributable to the presence of interruption points and uncommon branch entry points that force many names to be maintained anyway, and partially attributable to the infrequency of variable names becoming unused well prior to the end of their scope, perhaps because methods in SELF are typically much shorter than procedures in a traditional language. Ensuring that names stay live throughout their visible lifetimes takes a negligible amount of extra compile time and a negligible amount of compiled code space. Clearly, this aspect of the programming environment can be supported at no cost.

B.5.7 Interrupt Checking and Stack Overflow Checking

The SELF run-time system handles interrupts via polling, with the compiler generating code at method entry and **_Restart** points (loop tails) to check for interrupts, as described in section 6.3. The cost associated with these checks might be avoided by an alternative interrupt architecture that does no polling but instead backpatches instructions ahead of the current program counter to call the interrupt handler at exactly the same point that the polling code would have detected the interrupt. Also, since the interrupt check at each method entry doubles as a check for stack overflow, this use of polling would need to be eliminated by read-protecting the page after the maximum stack size and using the page protection hardware to detect stack overflow.

Since the costs current associated with interrupt polling could probably be avoided by a more sophisticated run-time system design, it is important to determine how costly is the polling overhead in the current SELF implementation. We can measure the cost of polling simply by not generating any polling code and ignoring interrupts. The following charts report the costs for the two sources of polling overhead.



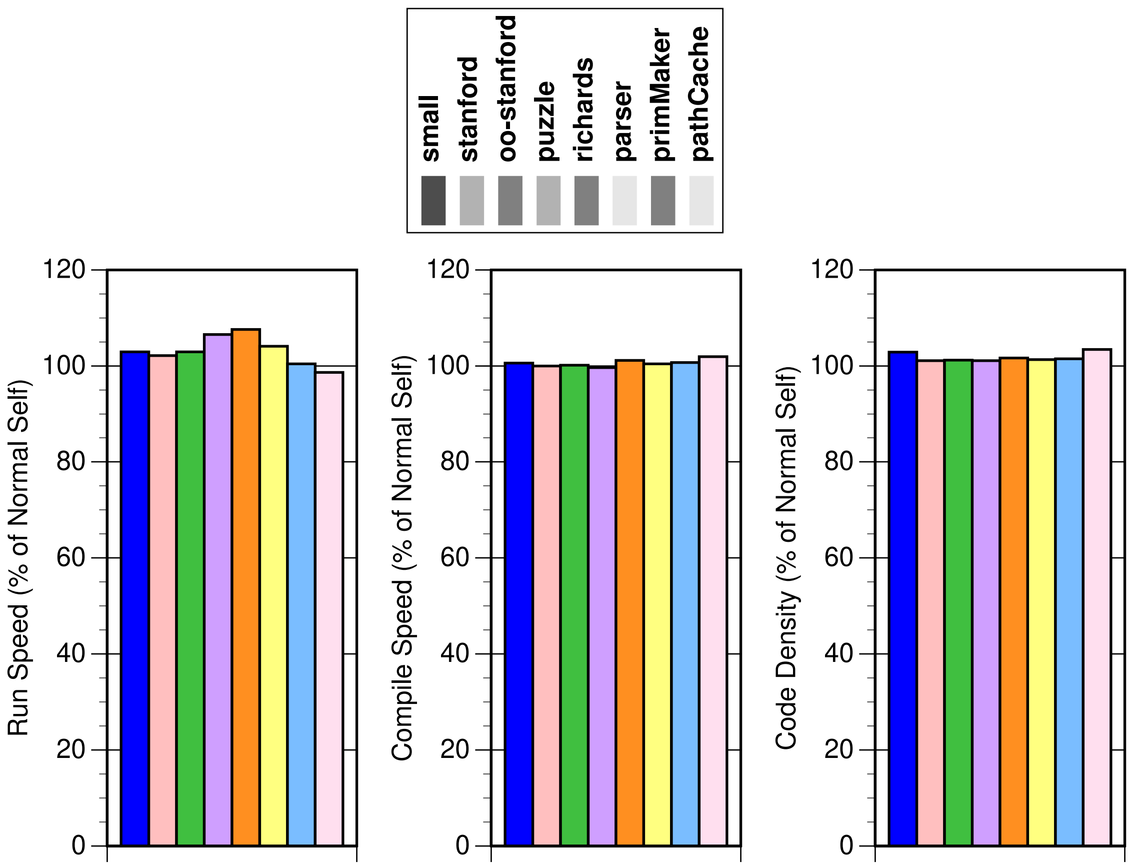
Interrupt checking imposes a moderate cost in execution time for most benchmarks. The more numerical benchmarks slow down by between 4% and 13% from interrupt checking at **_Restart**'s and a few percent from interrupt checking at method entries, while the larger, more object-oriented benchmarks slow down by a few percent from interrupt checking at method entries and virtually none from interrupt checking at **_Restart**'s. This difference reflects the fact that the smaller benchmarks contain more tight loops while the larger benchmarks contain more calls.

Generating code to handle interrupts imposes a fairly substantial compile-time cost, at least when compared to the other measured sources of overhead. Interrupt checking at **_Restart**'s is more costly than at calls, since it involves ensuring that all debugger-visible names are properly set up so that the debugger could display the virtual source-level call stack if invoked at the interrupt point. Supporting interrupts via polling imposes fairly significant compiled code space costs, with up to 35% extra compiled code generated for the smaller benchmarks. Again, interrupt checking at

_Restart's is more expensive than at calls since it may require extra code to support the debugger. It is unlikely, however, than either of these compile-time costs could be avoided by an alternative run-time mechanism for handling interrupts.

B.5.8 LRU Compiled Method Reclamation Support

Compiled methods are stored in a fixed-sized cache. Some compiled methods must be flushed from the cache to make room for new compiled methods when the cache is full. The system attempts to flush compiled methods that are unlikely to be used soon afterwards by keeping track of which compiled methods have been invoked recently, and flushing those compiled methods which have been used least recently. The compiler supports this *least-recently-used* (*LRU*) replacement strategy by generating extra code in the compiled method prologue to mark the compiled method as recently used, as described in section 8.2.3. This overhead would not exist in an implementation that either never threw away compiled code or replaced compiled methods in a different way (such as using *first-in first-out* (*FIFO*) replacement or LRU replacement with usage information computed via periodic interrupts). We measured the overhead for the current LRU implementation by disabling the marking code in the method prologue. The following charts display the costs of the current LRU compiled method reclamation support.



No LRU Compiled Method Reclamation Support

LRU support imposes a run-time cost of up to 8% over a system without this support. This cost is directly proportional to the call frequency of the program being measured. LRU reclamation support takes little additional compile time and little additional compiled code space.

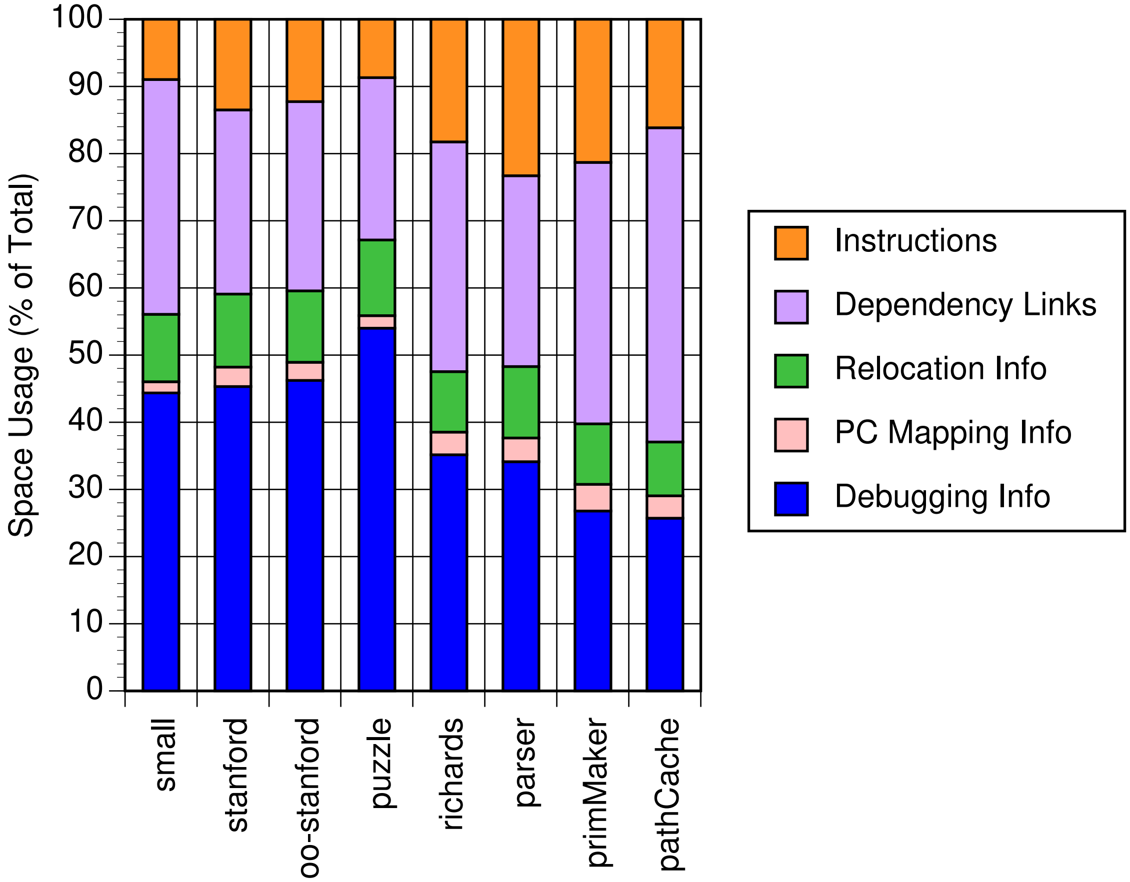
B.5.9 Summary of Remaining Sources of Overhead

Of the remaining sources of overhead we were able to measure, no single source of overhead imposes a significant execution speed cost. Only array bounds checking, type testing, overflow checking, interrupt checking, and LRU compiled method reclamation have a non-trivial execution time cost, and none incurs more than 10% overhead. Much of the remaining gap in performance between SELF and optimized C remains unaccounted for, however.

B.6 Additional Space Costs

The previous compiled code space efficiency measurements compared the number of machine instructions generated by the SELF compiler to the number of machine instructions generated by other configurations of SELF or by other language implementations. The SELF compiler generates additional information with compiled methods that takes up more space. These additional pieces of information include descriptions of inlined scopes and tables mapping between physical program counters and source-level byte code position. This information is used to reconstruct the virtual call stack from the physical call stack during debugging (as described in section 13.1), to support lazy compilation of uncommon branches, and to compile block methods that perform up-level accesses to lexically-enclosing stack frames. The compiler also generates dependency links, used for selective invalidation of compiled methods after programming changes (as described in section 13.2). Finally, the compiler generates information identifying the locations of all tagged object references embedded in compiled code and scope debugging information, to enable the system to update these pointers after a scavenge or garbage collection.

The following chart breaks down the space consumed by the output of the SELF compiler into the above categories of information, under the standard configuration.



This chart illustrates that machine instructions take only a relatively small fraction of the space consumed by compiler-generated data, around 10% of the space for the more numerical benchmarks and 20% of the space for the larger object-oriented programs. Scope debugging information and dependency links take up the lion's share of the space used by generated information, between 60% and 80% of the total generated space. Relocation information requires a roughly constant 10% of the total space. Program counter/byte code mappings are relatively concise, taking up less than 5% of the total space used.

Fortunately, not all of this information needs to remain in main memory at all times. Much of it can be paged out to virtual memory and brought into main memory only when needed. Compiled instructions need to be in main memory (for those methods in active use). Location information needs to be in main memory, since it will be accessed relatively frequently during scavenges.* Scope and p.c./byte code debugging information is needed only when compiling nested blocks and uncommon branch extensions and when debugging, so for methods not being debugged the scope information can be paged out relatively quickly after "warming up" the compiled code cache. Dependencies are only needed when programming and flushing invalid methods, so this space can be paged out when not in program development mode. Thus, for working, debugged programs most of the extra data generated by the compiler can be paged out of main memory, minimizing real memory requirements. Ultimately only the machine instructions and locations need be in main memory (and the latter only for garbage collections), thus keeping real memory space costs down to a fraction of the total virtual memory space costs.

* If all tagged object pointers embedded in a compiled method refer to tenured objects in old-space (objects not scanned as part of scavenging), this location information will only be needed during a full garbage collection and so normally can be paged out. As an optimization, the system could automatically tenure all objects reachable from compiled methods to allow the location information to be paged out immediately and to speed scavenges.

Appendix C Raw Benchmark Data

This appendix includes the raw data for all the performance measurements reported in Chapter 14 and Appendix B. For each row, from left to right, the tables report the running time for the benchmark (in seconds), the compile time for the benchmark (in seconds), and the size in bytes of the compiled instructions generated by the compiler for the benchmark. Smalltalk-80 compile time and compiled code space measurements are unavailable. Other blank rows correspond to configurations where the compiler consumed too much internal memory (over 30MB) when compiling the benchmark. Note that the Local Reluctant Splitting (Lazy) and Late Block Zapping configurations are identical to the Normal SELF configuration.

C.1 recur

language/configuration	run	compile	space
Optimized C	0.293	1.1	88
Smalltalk-80	1.459		
T (normal)	3.316	0.73	1400
T (integer only)	0.55	0.35	404
Normal SELF	0.495	0.485	616
No Splitting (Not Lazy)	0.715	2.745	6840
No Splitting (Lazy)	0.666	0.484	792
Local Reluctant Splitting (Not Lazy)	0.495	0.475	616
Local Reluctant Splitting (Lazy)	0.495	0.485	616
Global Reluctant Splitting (Not Lazy)	0.496	0.464	616
Global Reluctant Splitting (Lazy)	0.495	0.475	616
Divided No Splitting (Not Lazy)	0.697	1.583	3920
Divided No Splitting (Lazy)	0.666	0.474	792
Divided Local Reluctant Splitting (Not Lazy)	0.496	0.474	616
Divided Local Reluctant Splitting (Lazy)	0.495	0.475	616
Divided Global Reluctant Splitting (Not Lazy)	0.496	0.474	616
Divided Global Reluctant Splitting (Lazy)	0.496	0.484	616
Eager Splitting (Not Lazy)	0.496	0.844	1000
Eager Splitting (Lazy)	0.495	0.845	1000
Eager Splitting and Tail Merging (Not Lazy)	0.496	0.484	628
Eager Splitting and Tail Merging (Lazy)	0.495	0.495	628
Eager Splitting and Requirements Analysis (Not Lazy)	0.496	0.724	616
Eager Splitting and Requirements Analysis (Lazy)	0.496	0.724	616
Type Predict for Vectors (with Local Splitting)	0.497	0.473	616
Type Predict for Vectors (with Global Splitting)	0.497	0.473	616
Vectors Are More Common (with Local Splitting)	0.496	0.464	616
Without Inlining	22.513	2.137	616
Without In-Line Caching	0.495	0.475	616
Without Compile-Time Lookup Caching	0.495	0.465	724
Without Customization	0.535	0.875	616
Without Value-Based Type Analysis	0.768	0.512	6552
Without Range Analysis	0.516	0.614	616
Without Type Prediction	0.497	0.463	616
Without Deferred Block Creation	4.997	3.833	616
Without Exposed Block Analysis	0.498	0.912	976
Without CSE	0.506	0.464	1636
Without CSE of Constants	0.496	0.474	1364
Without CSE of Arithmetic Operations	0.507	0.473	832
Without CSE of Memory References	0.495	0.465	624
Without CSE of Memory Cell Type Information	0.496	0.474	616
Without CSE of Memory Cell Array Bounds Checking	0.497	0.473	624
Without Eliminating Unneeded Computations	0.504	0.486	616
Without Delay Slot Filling	0.607	0.443	616
No Integer Type Tests	0.496	0.464	616
No Boolean Type Tests	0.497	0.463	616
No Overflow Checking	0.496	0.474	720
No Array Bounds Checking	0.496	0.474	616
No Block Zapping	0.497	0.463	592
Early Block Zapping	0.497	0.483	616
Late Block Zapping	0.495	0.485	616
No Primitive Failure Checking	0.497	0.473	520
No Debugger-Visible Names	0.497	0.463	584
No Interrupt Checking at Calls	0.457	0.443	616
No Interrupt Checking at _Restarts	0.496	0.454	616
No LRU Compiled Method Reclamation Support	0.416	0.474	616
Fast	0.415	0.435	616
Fastest	0.416	0.414	616

C.2 sumTo

language/configuration	run	compile	space
Optimized C	0.245	1.1	96
Smalltalk-80	2.366		
T (normal)	4.682	0.81	1440
T (integer only)	0.303	0.4	348
Normal SELF	0.67	0.65	632
No Splitting (Not Lazy)	2.013	4.167	8512
No Splitting (Lazy)	1.099	0.631	760
Local Reluctant Splitting (Not Lazy)	1.586	2.944	5104
Local Reluctant Splitting (Lazy)	0.67	0.65	632
Global Reluctant Splitting (Not Lazy)	0.915	10.385	19524
Global Reluctant Splitting (Lazy)	0.671	0.649	632
Divided No Splitting (Not Lazy)	1.221	3.199	6500
Divided No Splitting (Lazy)	1.099	0.641	760
Divided Local Reluctant Splitting (Not Lazy)	0.854	2.696	4788
Divided Local Reluctant Splitting (Lazy)	0.671	0.659	632
Divided Global Reluctant Splitting (Not Lazy)	0.854	3.136	4900
Divided Global Reluctant Splitting (Lazy)	0.671	0.649	632
Eager Splitting (Not Lazy)	0.733	6.107	9356
Eager Splitting (Lazy)	0.732	1.368	1120
Eager Splitting and Tail Merging (Not Lazy)	0.732	3.008	4788
Eager Splitting and Tail Merging (Lazy)	0.732	0.748	660
Eager Splitting and Requirements Analysis (Not Lazy)	0.978	3.652	4864
Eager Splitting and Requirements Analysis (Lazy)	0.672	1.228	632
Type Predict for Vectors (with Local Splitting)	0.671	0.649	632
Type Predict for Vectors (with Global Splitting)	0.672	0.648	632
Vectors Are More Common (with Local Splitting)	0.671	0.649	632
Without Inlining	254.721	1.519	632
Without In-Line Caching	0.671	0.639	632
Without Compile-Time Lookup Caching	0.672	0.648	720
Without Customization	0.672	1.038	632
Without Value-Based Type Analysis	2.627	0.733	5380
Without Range Analysis	0.671	0.789	632
Without Type Prediction	0.673	0.647	632
Without Deferred Block Creation	18.107	1.943	632
Without Exposed Block Analysis	2.02	2.32	1044
Without CSE	0.671	0.639	1996
Without CSE of Constants	0.672	0.648	3928
Without CSE of Arithmetic Operations	0.671	0.639	812
Without CSE of Memory References	0.671	0.639	632
Without CSE of Memory Cell Type Information	0.671	0.649	632
Without CSE of Memory Cell Array Bounds Checking	0.671	0.649	632
Without Eliminating Unneeded Computations	0.694	0.626	632
Without Delay Slot Filling	1.099	0.601	632
No Integer Type Tests	0.671	0.639	632
No Boolean Type Tests	0.672	0.648	632
No Overflow Checking	0.55	0.62	716
No Array Bounds Checking	0.672	0.648	608
No Block Zapping	0.672	0.648	616
Early Block Zapping	0.672	0.678	632
Late Block Zapping	0.67	0.65	632
No Primitive Failure Checking	0.671	0.649	568
No Debugger-Visible Names	0.672	0.638	488
No Interrupt Checking at Calls	0.671	0.619	632
No Interrupt Checking at _Restarts	0.489	0.571	552
No LRU Compiled Method Reclamation Support	0.672	0.648	632
Fast	0.488	0.542	632
Fastest	0.427	0.493	632

C.3 sumFromTo

language/configuration	run	compile	space
Optimized C	0.245	1.1	96
Smalltalk-80	2.364		
T (normal)	4.627	0.81	1432
T (integer only)	0.305	0.37	344
Normal SELF	0.672	0.648	664
No Splitting (Not Lazy)	2.013	4.277	8720
No Splitting (Lazy)	1.098	0.642	792
Local Reluctant Splitting (Not Lazy)	1.585	3.075	5256
Local Reluctant Splitting (Lazy)	0.672	0.648	664
Global Reluctant Splitting (Not Lazy)	0.916	9.084	19656
Global Reluctant Splitting (Lazy)	0.671	0.649	664
Divided No Splitting (Not Lazy)	1.221	3.289	6704
Divided No Splitting (Lazy)	1.099	0.641	792
Divided Local Reluctant Splitting (Not Lazy)	0.854	2.806	4956
Divided Local Reluctant Splitting (Lazy)	0.672	0.648	664
Divided Global Reluctant Splitting (Not Lazy)	0.855	3.045	5068
Divided Global Reluctant Splitting (Lazy)	0.671	0.649	664
Eager Splitting (Not Lazy)	0.733	5.667	9632
Eager Splitting (Lazy)	0.732	1.368	1184
Eager Splitting and Tail Merging (Not Lazy)	0.733	2.907	4940
Eager Splitting and Tail Merging (Lazy)	0.733	0.747	700
Eager Splitting and Requirements Analysis (Not Lazy)	0.978	3.432	5072
Eager Splitting and Requirements Analysis (Lazy)	0.671	1.229	672
Type Predict for Vectors (with Local Splitting)	0.671	0.649	664
Type Predict for Vectors (with Global Splitting)	0.672	0.648	664
Vectors Are More Common (with Local Splitting)	0.672	0.648	664
Without Inlining	254.801	1.509	664
Without In-Line Caching	0.672	0.648	664
Without Compile-Time Lookup Caching	0.672	0.648	748
Without Customization	0.672	1.058	664
Without Value-Based Type Analysis	2.75	0.58	5516
Without Range Analysis	0.671	0.779	1172
Without Type Prediction	46.915	1.465	664
Without Deferred Block Creation	18.11	1.98	664
Without Exposed Block Analysis	2.021	2.349	836
Without CSE	0.672	0.628	2036
Without CSE of Constants	0.671	0.639	4016
Without CSE of Arithmetic Operations	0.671	0.629	852
Without CSE of Memory References	0.672	0.638	664
Without CSE of Memory Cell Type Information	0.672	0.648	664
Without CSE of Memory Cell Array Bounds Checking	0.672	0.638	664
Without Eliminating Unneeded Computations	0.674	0.666	664
Without Delay Slot Filling	1.099	0.611	664
No Integer Type Tests	0.671	0.639	664
No Boolean Type Tests	0.672	0.648	664
No Overflow Checking	0.549	0.631	752
No Array Bounds Checking	0.672	0.638	616
No Block Zapping	0.671	0.649	648
Early Block Zapping	0.672	0.678	664
Late Block Zapping	0.672	0.648	664
No Primitive Failure Checking	0.672	0.648	600
No Debugger-Visible Names	0.671	0.639	520
No Interrupt Checking at Calls	0.671	0.639	664
No Interrupt Checking at _Restarts	0.488	0.572	584
No LRU Compiled Method Reclamation Support	0.672	0.638	664
Fast	0.488	0.542	664
Fastest	0.428	0.502	664

C.4 fastSumTo

language/configuration	run	compile	space
Optimized C	0.245	1.1	80
Smalltalk-80	2.368		
T (normal)	4.194	0.76	1344
T (integer only)	0.432	0.34	260
Normal SELF	0.732	0.528	368
No Splitting (Not Lazy)	2.012	20.508	28024
No Splitting (Lazy)	1.219	0.531	460
Local Reluctant Splitting (Not Lazy)	0.975	9.455	11552
Local Reluctant Splitting (Lazy)	0.732	0.528	368
Global Reluctant Splitting (Not Lazy)	0.854	2.256	3148
Global Reluctant Splitting (Lazy)	0.732	0.538	368
Divided No Splitting (Not Lazy)	1.28	2.75	4188
Divided No Splitting (Lazy)	1.22	0.53	460
Divided Local Reluctant Splitting (Not Lazy)	0.915	2.485	3348
Divided Local Reluctant Splitting (Lazy)	0.731	0.529	368
Divided Global Reluctant Splitting (Not Lazy)	0.854	2.466	3376
Divided Global Reluctant Splitting (Lazy)	0.732	0.538	368
Eager Splitting (Not Lazy)	0.793	26.337	30636
Eager Splitting (Lazy)	0.731	23.289	26268
Eager Splitting and Tail Merging (Not Lazy)	0.793	13.037	15368
Eager Splitting and Tail Merging (Lazy)	0.732	11.398	13184
Eager Splitting and Requirements Analysis (Not Lazy)	1.038	3.182	3392
Eager Splitting and Requirements Analysis (Lazy)	0.732	1.348	368
Type Predict for Vectors (with Local Splitting)	0.732	0.538	368
Type Predict for Vectors (with Global Splitting)	0.732	0.538	368
Vectors Are More Common (with Local Splitting)	0.732	0.538	368
Without Inlining	254.883	1.487	368
Without In-Line Caching	0.733	0.527	368
Without Compile-Time Lookup Caching	0.732	0.528	380
Without Customization	0.732	0.908	368
Without Value-Based Type Analysis	2.075	0.575	5332
Without Range Analysis	0.793	1.257	368
Without Type Prediction	0.733	0.527	368
Without Deferred Block Creation	18.211	1.669	368
Without Exposed Block Analysis	0.796	4.774	556
Without CSE	0.732	0.518	1636
Without CSE of Constants	0.732	0.538	5232
Without CSE of Arithmetic Operations	0.732	0.528	864
Without CSE of Memory References	0.732	0.528	368
Without CSE of Memory Cell Type Information	0.732	0.538	368
Without CSE of Memory Cell Array Bounds Checking	0.731	0.539	368
Without Eliminating Unneeded Computations	0.817	0.443	368
Without Delay Slot Filling	1.098	0.502	368
No Integer Type Tests	0.732	0.528	368
No Boolean Type Tests	0.732	0.528	368
No Overflow Checking	0.671	0.509	420
No Array Bounds Checking	0.732	0.538	368
No Block Zapping	0.732	0.538	360
Early Block Zapping	0.732	0.548	368
Late Block Zapping	0.732	0.528	368
No Primitive Failure Checking	0.732	0.528	336
No Debugger-Visible Names	0.732	0.538	200
No Interrupt Checking at Calls	0.732	0.528	368
No Interrupt Checking at _Restarts	0.61	0.39	328
No LRU Compiled Method Reclamation Support	0.732	0.538	368
Fast	0.609	0.381	368
Fastest	0.549	0.351	368

C.5 nestedLoop

language/configuration	run	compile	space
Optimized C	0.25	1.1	88
Smalltalk-80	2.027		
T (normal)	4.469	1.02	1796
T (integer only)	1.285	0.49	492
Normal SELF	0.618	0.762	604
No Splitting (Not Lazy)	1.794	16.846	26756
No Splitting (Lazy)	1.115	0.755	752
Local Reluctant Splitting (Not Lazy)	0.869	6.151	8092
Local Reluctant Splitting (Lazy)	0.618	0.762	604
Global Reluctant Splitting (Not Lazy)	0.741	2.619	3536
Global Reluctant Splitting (Lazy)	0.618	0.762	604
Divided No Splitting (Not Lazy)	1.242	3.548	6016
Divided No Splitting (Lazy)	1.116	0.764	752
Divided Local Reluctant Splitting (Not Lazy)	0.805	2.665	3644
Divided Local Reluctant Splitting (Lazy)	0.618	0.762	604
Divided Global Reluctant Splitting (Not Lazy)	0.804	2.716	3720
Divided Global Reluctant Splitting (Lazy)	0.619	0.781	604
Eager Splitting (Not Lazy)	0.682	18.218	24736
Eager Splitting (Lazy)	0.679	15.131	20344
Eager Splitting and Tail Merging (Not Lazy)	0.682	9.148	12524
Eager Splitting and Tail Merging (Lazy)	0.679	7.631	10328
Eager Splitting and Requirements Analysis (Not Lazy)	0.928	3.452	3680
Eager Splitting and Requirements Analysis (Lazy)	0.618	1.572	604
Type Predict for Vectors (with Local Splitting)	0.619	0.761	604
Type Predict for Vectors (with Global Splitting)	0.619	0.771	604
Vectors Are More Common (with Local Splitting)	0.619	0.761	604
Without Inlining	310.865	1.965	604
Without In-Line Caching	0.619	0.761	604
Without Compile-Time Lookup Caching	0.619	0.771	664
Without Customization	0.618	1.162	604
Without Value-Based Type Analysis	1.856	0.814	5988
Without Range Analysis	0.68	1.59	604
Without Type Prediction	0.62	0.76	604
Without Deferred Block Creation	18.87	2.44	604
Without Exposed Block Analysis	3.252	5.998	884
Without CSE	0.619	0.741	2440
Without CSE of Constants	0.619	0.761	8136
Without CSE of Arithmetic Operations	0.619	0.751	1240
Without CSE of Memory References	0.618	0.762	604
Without CSE of Memory Cell Type Information	0.618	0.752	604
Without CSE of Memory Cell Array Bounds Checking	0.618	0.762	604
Without Eliminating Unneeded Computations	0.622	0.778	604
Without Delay Slot Filling	0.989	0.721	604
No Integer Type Tests	0.619	0.771	604
No Boolean Type Tests	0.619	0.771	604
No Overflow Checking	0.557	0.753	696
No Array Bounds Checking	0.619	0.761	604
No Block Zapping	0.619	0.741	588
Early Block Zapping	0.619	0.791	604
Late Block Zapping	0.618	0.762	604
No Primitive Failure Checking	0.618	0.762	540
No Debugger-Visible Names	0.619	0.761	404
No Interrupt Checking at Calls	0.618	0.752	604
No Interrupt Checking at _Restarts	0.494	0.616	564
No LRU Compiled Method Reclamation Support	0.619	0.771	604
Fast	0.494	0.576	604
Fastest	0.434	0.546	604

C.6 atAllPut

language/configuration	run	compile	space
Optimized C	0.065	1.1	88
Smalltalk-80	1.046		
T (normal)	0.322	0.53	992
T (integer only)	0.095	0.36	368
Normal SELF	0.13	0.29	404
No Splitting (Not Lazy)	0.266	2.204	4644
No Splitting (Lazy)	0.173	0.287	440
Local Reluctant Splitting (Not Lazy)	0.143	0.367	532
Local Reluctant Splitting (Lazy)	0.13	0.29	404
Global Reluctant Splitting (Not Lazy)	0.112	0.698	1040
Global Reluctant Splitting (Lazy)	0.099	0.511	636
Divided No Splitting (Not Lazy)	0.186	0.654	1300
Divided No Splitting (Lazy)	0.173	0.287	440
Divided Local Reluctant Splitting (Not Lazy)	0.143	0.727	1264
Divided Local Reluctant Splitting (Lazy)	0.131	0.289	404
Divided Global Reluctant Splitting (Not Lazy)	0.112	1.088	1748
Divided Global Reluctant Splitting (Lazy)	0.1	0.51	636
Eager Splitting (Not Lazy)	0.118	2.552	3660
Eager Splitting (Lazy)	0.106	1.314	1148
Eager Splitting and Tail Merging (Not Lazy)	0.111	1.309	1924
Eager Splitting and Tail Merging (Lazy)	0.099	0.711	680
Eager Splitting and Requirements Analysis (Not Lazy)	0.136	1.814	2020
Eager Splitting and Requirements Analysis (Lazy)	0.1	1.2	656
Type Predict for Vectors (with Local Splitting)	0.13	0.29	404
Type Predict for Vectors (with Global Splitting)	0.1	0.52	636
Vectors Are More Common (with Local Splitting)	0.13	0.29	404
Without Inlining	27.572	1.168	636
Without In-Line Caching	0.131	0.289	404
Without Compile-Time Lookup Caching	0.13	0.29	1024
Without Customization	1.381	0.849	404
Without Value-Based Type Analysis	0.396	0.334	5312
Without Range Analysis	0.136	0.424	404
Without Type Prediction	0.131	0.299	404
Without Deferred Block Creation	2.365	0.505	404
Without Exposed Block Analysis	0.13	0.71	608
Without CSE	0.148	0.282	1076
Without CSE of Constants	0.13	0.29	1008
Without CSE of Arithmetic Operations	0.13	0.29	564
Without CSE of Memory References	0.149	0.281	408
Without CSE of Memory Cell Type Information	0.13	0.29	404
Without CSE of Memory Cell Array Bounds Checking	0.131	0.289	404
Without Eliminating Unneeded Computations	0.13	0.29	408
Without Delay Slot Filling	0.167	0.283	404
No Integer Type Tests	0.13	0.29	404
No Boolean Type Tests	0.131	0.289	404
No Overflow Checking	0.131	0.299	464
No Array Bounds Checking	0.118	0.282	404
No Block Zapping	0.13	0.29	388
Early Block Zapping	0.13	0.3	404
Late Block Zapping	0.13	0.29	404
No Primitive Failure Checking	0.131	0.289	340
No Debugger-Visible Names	0.13	0.3	328
No Interrupt Checking at Calls	0.13	0.28	404
No Interrupt Checking at _Restarts	0.118	0.252	404
No LRU Compiled Method Reclamation Support	0.131	0.289	380
Fast	0.117	0.233	404
Fastest	0.106	0.224	404

C.7 sumAll

language/configuration	run	compile	space
Optimized C	0.055	1.1	88
Smalltalk-80	0.698		
T (normal)	0.505	0.71	1196
T (integer only)	0.098	0.33	372
Normal SELF	0.13	0.49	580
No Splitting (Not Lazy)	0.269	4.921	9448
No Splitting (Lazy)	0.172	0.498	652
Local Reluctant Splitting (Not Lazy)	0.16	2.29	3512
Local Reluctant Splitting (Lazy)	0.13	0.49	580
Global Reluctant Splitting (Not Lazy)	0.147	2.403	3756
Global Reluctant Splitting (Lazy)	0.129	0.501	580
Divided No Splitting (Not Lazy)	0.184	2.516	4688
Divided No Splitting (Lazy)	0.172	0.488	652
Divided Local Reluctant Splitting (Not Lazy)	0.147	2.693	4492
Divided Local Reluctant Splitting (Lazy)	0.129	0.501	580
Divided Global Reluctant Splitting (Not Lazy)	0.147	2.703	4504
Divided Global Reluctant Splitting (Lazy)	0.129	0.501	580
Eager Splitting (Not Lazy)	0.134	5.466	8752
Eager Splitting (Lazy)	0.129	1.101	1020
Eager Splitting and Tail Merging (Not Lazy)	0.135	2.755	4504
Eager Splitting and Tail Merging (Lazy)	0.129	0.591	620
Eager Splitting and Requirements Analysis (Not Lazy)	0.16	3.29	4588
Eager Splitting and Requirements Analysis (Lazy)	0.129	1.021	580
Type Predict for Vectors (with Local Splitting)	0.129	0.491	580
Type Predict for Vectors (with Global Splitting)	0.128	0.492	580
Vectors Are More Common (with Local Splitting)	0.129	0.491	580
Without Inlining	36.196	1.274	580
Without In-Line Caching	0.13	0.5	580
Without Compile-Time Lookup Caching	0.129	0.511	1860
Without Customization	10.94	1.11	580
Without Value-Based Type Analysis	0.453	0.587	5928
Without Range Analysis	0.135	0.505	580
Without Type Prediction	0.129	0.491	580
Without Deferred Block Creation	2.762	0.888	580
Without Exposed Block Analysis	0.283	2.157	1012
Without CSE	0.147	0.473	1600
Without CSE of Constants	0.128	0.502	3660
Without CSE of Arithmetic Operations	0.129	0.491	644
Without CSE of Memory References	0.147	0.483	588
Without CSE of Memory Cell Type Information	0.13	0.5	580
Without CSE of Memory Cell Array Bounds Checking	0.129	0.491	580
Without Eliminating Unneeded Computations	0.129	0.501	588
Without Delay Slot Filling	0.177	0.473	580
No Integer Type Tests	0.117	0.483	580
No Boolean Type Tests	0.129	0.491	580
No Overflow Checking	0.122	0.488	668
No Array Bounds Checking	0.117	0.473	532
No Block Zapping	0.129	0.501	564
Early Block Zapping	0.129	0.521	580
Late Block Zapping	0.13	0.49	580
No Primitive Failure Checking	0.129	0.491	516
No Debugger-Visible Names	0.129	0.491	472
No Interrupt Checking at Calls	0.129	0.491	580
No Interrupt Checking at _Restarts	0.117	0.423	540
No LRU Compiled Method Reclamation Support	0.129	0.491	532
Fast	0.117	0.423	580
Fastest	0.086	0.354	580

C.8 incrementAll

language/configuration	run	compile	space
Optimized C	0.113	1.1	96
Smalltalk-80	1.754		
T (normal)	0.538	0.7	1188
T (integer only)	0.152	0.32	376
Normal SELF	0.173	0.347	440
No Splitting (Not Lazy)	0.369	4.061	8220
No Splitting (Lazy)	0.216	0.334	476
Local Reluctant Splitting (Not Lazy)	0.218	0.912	1428
Local Reluctant Splitting (Lazy)	0.173	0.347	440
Global Reluctant Splitting (Not Lazy)	0.218	0.932	1464
Global Reluctant Splitting (Lazy)	0.172	0.348	440
Divided No Splitting (Not Lazy)	0.228	4.132	8412
Divided No Splitting (Lazy)	0.216	0.334	476
Divided Local Reluctant Splitting (Not Lazy)	0.191	0.949	1504
Divided Local Reluctant Splitting (Lazy)	0.173	0.347	440
Divided Global Reluctant Splitting (Not Lazy)	0.192	0.958	1500
Divided Global Reluctant Splitting (Lazy)	0.172	0.348	440
Eager Splitting (Not Lazy)	0.179	1.841	2752
Eager Splitting (Lazy)	0.173	0.697	712
Eager Splitting and Tail Merging (Not Lazy)	0.18	0.97	1484
Eager Splitting and Tail Merging (Lazy)	0.173	0.387	464
Eager Splitting and Requirements Analysis (Not Lazy)	0.203	5.327	8700
Eager Splitting and Requirements Analysis (Lazy)	0.173	0.657	440
Type Predict for Vectors (with Local Splitting)	0.173	0.347	440
Type Predict for Vectors (with Global Splitting)	0.173	0.347	440
Vectors Are More Common (with Local Splitting)	0.173	0.347	440
Without Inlining	34.257	1.463	440
Without In-Line Caching	0.173	0.347	440
Without Compile-Time Lookup Caching	0.173	0.347	2048
Without Customization	8.968	1.202	440
Without Value-Based Type Analysis	0.588	0.402	6148
Without Range Analysis	0.18	0.52	604
Without Type Prediction	0.306	0.394	440
Without Deferred Block Creation	3.6	0.72	440
Without Exposed Block Analysis	0.173	0.837	736
Without CSE	0.215	0.345	1268
Without CSE of Constants	0.173	0.347	1116
Without CSE of Arithmetic Operations	0.173	0.347	644
Without CSE of Memory References	0.216	0.344	472
Without CSE of Memory Cell Type Information	0.173	0.347	440
Without CSE of Memory Cell Array Bounds Checking	0.182	0.358	440
Without Eliminating Unneeded Computations	0.173	0.337	472
Without Delay Slot Filling	0.22	0.33	440
No Integer Type Tests	0.164	0.336	464
No Boolean Type Tests	0.172	0.348	440
No Overflow Checking	0.168	0.332	508
No Array Bounds Checking	0.161	0.339	416
No Block Zapping	0.173	0.347	424
Early Block Zapping	0.173	0.367	440
Late Block Zapping	0.173	0.347	440
No Primitive Failure Checking	0.173	0.347	376
No Debugger-Visible Names	0.173	0.347	364
No Interrupt Checking at Calls	0.172	0.338	440
No Interrupt Checking at _Restarts	0.16	0.31	420
No LRU Compiled Method Reclamation Support	0.173	0.337	416
Fast	0.16	0.28	440
Fastest	0.133	0.257	440

C.9 tak

language/configuration	run	compile	space
Optimized C	0.065	1.1	152
Smalltalk-80	0.309		
T (normal)	0.55	0.88	1592
T (integer only)	0.133	0.43	532
Normal SELF	0.146	0.234	700
No Splitting (Not Lazy)	0.238	1.062	3576
No Splitting (Lazy)	0.176	0.234	772
Local Reluctant Splitting (Not Lazy)	0.166	1.194	3428
Local Reluctant Splitting (Lazy)	0.146	0.234	700
Global Reluctant Splitting (Not Lazy)	0.162	1.498	4364
Global Reluctant Splitting (Lazy)	0.147	0.233	700
Divided No Splitting (Not Lazy)	0.189	2.021	6644
Divided No Splitting (Lazy)	0.177	0.233	772
Divided Local Reluctant Splitting (Not Lazy)	0.162	2.188	6332
Divided Local Reluctant Splitting (Lazy)	0.147	0.233	700
Divided Global Reluctant Splitting (Not Lazy)	0.163	2.037	5820
Divided Global Reluctant Splitting (Lazy)	0.147	0.233	700
Eager Splitting (Not Lazy)	0.154	3.346	9600
Eager Splitting (Lazy)	0.146	0.484	1280
Eager Splitting and Tail Merging (Not Lazy)	0.154	2.216	6324
Eager Splitting and Tail Merging (Lazy)	0.146	0.294	712
Eager Splitting and Requirements Analysis (Not Lazy)	0.171	2.349	6432
Eager Splitting and Requirements Analysis (Lazy)	0.146	0.384	700
Type Predict for Vectors (with Local Splitting)	0.146	0.234	700
Type Predict for Vectors (with Global Splitting)	0.146	0.244	700
Vectors Are More Common (with Local Splitting)	0.146	0.234	700
Without Inlining	7.079	1.001	700
Without In-Line Caching	0.147	0.243	700
Without Compile-Time Lookup Caching	0.147	0.243	712
Without Customization	0.147	0.613	700
Without Value-Based Type Analysis	0.2	0.28	2600
Without Range Analysis	0.146	0.244	1864
Without Type Prediction	5.771	0.419	700
Without Deferred Block Creation	1.233	0.317	700
Without Exposed Block Analysis	0.146	0.244	928
Without CSE	0.147	0.223	1088
Without CSE of Constants	0.146	0.234	700
Without CSE of Arithmetic Operations	0.146	0.234	700
Without CSE of Memory References	0.147	0.233	700
Without CSE of Memory Cell Type Information	0.146	0.234	700
Without CSE of Memory Cell Array Bounds Checking	0.146	0.234	700
Without Eliminating Unneeded Computations	0.147	0.233	700
Without Delay Slot Filling	0.176	0.224	700
No Integer Type Tests	0.129	0.231	700
No Boolean Type Tests	0.146	0.244	700
No Overflow Checking	0.143	0.237	816
No Array Bounds Checking	0.146	0.234	604
No Block Zapping	0.147	0.243	684
Early Block Zapping	0.146	0.244	700
Late Block Zapping	0.146	0.234	700
No Primitive Failure Checking	0.146	0.234	636
No Debugger-Visible Names	0.146	0.234	700
No Interrupt Checking at Calls	0.139	0.221	700
No Interrupt Checking at _Restarts	0.146	0.234	640
No LRU Compiled Method Reclamation Support	0.13	0.23	700
Fast	0.131	0.239	700
Fastest	0.11	0.2	700

C.10 takl

language/configuration	run	compile	space
Optimized C	2.66	1.7	448
Smalltalk-80	8.279		
T (normal)	0.782	0.97	1584
T (integer only)	0.776	0.65	1044
Normal SELF	4.239	0.741	2872
No Splitting (Not Lazy)	4.776	1.064	4292
No Splitting (Lazy)	4.744	0.686	2996
Local Reluctant Splitting (Not Lazy)	4.248	1.162	4160
Local Reluctant Splitting (Lazy)	4.239	0.741	2872
Global Reluctant Splitting (Not Lazy)	4.418	1.122	4612
Global Reluctant Splitting (Lazy)	4.238	0.812	3220
Divided No Splitting (Not Lazy)	4.752	1.168	4772
Divided No Splitting (Lazy)	4.745	0.695	2996
Divided Local Reluctant Splitting (Not Lazy)	4.248	1.252	4612
Divided Local Reluctant Splitting (Lazy)	4.238	0.722	2872
Divided Global Reluctant Splitting (Not Lazy)	4.291	1.309	5064
Divided Global Reluctant Splitting (Lazy)	4.239	0.811	3220
Eager Splitting (Not Lazy)	4.135	2.475	9104
Eager Splitting (Lazy)	4.177	1.403	5932
Eager Splitting and Tail Merging (Not Lazy)	4.162	1.418	5284
Eager Splitting and Tail Merging (Lazy)	4.212	0.788	3376
Eager Splitting and Requirements Analysis (Not Lazy)	4.134	1.796	5484
Eager Splitting and Requirements Analysis (Lazy)	4.123	1.287	3276
Type Predict for Vectors (with Local Splitting)	4.239	0.731	2872
Type Predict for Vectors (with Global Splitting)	4.242	0.818	3220
Vectors Are More Common (with Local Splitting)	4.241	0.729	2872
Without Inlining	82.578	2.612	3220
Without In-Line Caching	4.248	0.722	2872
Without Compile-Time Lookup Caching	4.248	0.722	4212
Without Customization	8.346	1.294	2872
Without Value-Based Type Analysis	4.828	0.792	6996
Without Range Analysis	4.241	0.729	3416
Without Type Prediction	9.352	0.678	2872
Without Deferred Block Creation	9.873	1.087	2872
Without Exposed Block Analysis	4.26	0.73	3344
Without CSE	4.104	0.686	3324
Without CSE of Constants	4.104	0.716	2872
Without CSE of Arithmetic Operations	4.243	0.727	2872
Without CSE of Memory References	4.241	0.709	2868
Without CSE of Memory Cell Type Information	4.238	0.742	2868
Without CSE of Memory Cell Array Bounds Checking	4.242	0.728	2872
Without Eliminating Unneeded Computations	4.338	0.662	2872
Without Delay Slot Filling	4.447	0.703	2872
No Integer Type Tests	4.25	0.71	2872
No Boolean Type Tests	4.235	0.725	2872
No Overflow Checking	4.24	0.72	3376
No Array Bounds Checking	4.242	0.738	2720
No Block Zapping	4.242	0.748	2808
Early Block Zapping	4.242	0.768	2872
Late Block Zapping	4.239	0.741	2872
No Primitive Failure Checking	4.24	0.72	2632
No Debugger-Visible Names	4.243	0.727	2888
No Interrupt Checking at Calls	4.177	0.693	2836
No Interrupt Checking at _Restarts	4.242	0.738	2852
No LRU Compiled Method Reclamation Support	4.11	0.73	2872
Fast	4.107	0.693	2868
Fastest	4.107	0.813	2872

C.11 sieve

language/configuration	run	compile	space
Optimized C	0.165	1.5	224
Smalltalk-80	2.953		
T (normal)	1.146	1.63	2756
T (integer only)	0.423	0.67	668
Normal SELF	0.49	0.97	856
No Splitting (Not Lazy)	1.03	20.79	41440
No Splitting (Lazy)	0.722	0.988	1128
Local Reluctant Splitting (Not Lazy)	0.742	18.218	26684
Local Reluctant Splitting (Lazy)	0.49	0.97	856
Global Reluctant Splitting (Not Lazy)	0.524	2.576	3908
Global Reluctant Splitting (Lazy)	0.489	0.961	856
Divided No Splitting (Not Lazy)	0.807	3.773	7300
Divided No Splitting (Lazy)	0.727	1.003	1128
Divided Local Reluctant Splitting (Not Lazy)	0.576	2.754	4140
Divided Local Reluctant Splitting (Lazy)	0.489	0.971	856
Divided Global Reluctant Splitting (Not Lazy)	0.575	2.795	4208
Divided Global Reluctant Splitting (Lazy)	0.489	0.981	856
Eager Splitting (Not Lazy)	0.532	51.768	72776
Eager Splitting (Lazy)	0.525	48.445	68312
Eager Splitting and Tail Merging (Not Lazy)	0.532	33.078	47328
Eager Splitting and Tail Merging (Lazy)	0.525	31.585	45096
Eager Splitting and Requirements Analysis (Not Lazy)	0.539	8.241	9696
Eager Splitting and Requirements Analysis (Lazy)	0.503	3.467	1588
Type Predict for Vectors (with Local Splitting)	0.489	0.961	856
Type Predict for Vectors (with Global Splitting)	0.49	0.97	856
Vectors Are More Common (with Local Splitting)	0.49	0.97	856
Without Inlining	83.298	2.342	856
Without In-Line Caching	0.487	0.983	856
Without Compile-Time Lookup Caching	0.486	0.984	3152
Without Customization	4.428	2.672	856
Without Value-Based Type Analysis	1.002	1.108	9248
Without Range Analysis	0.601	1.719	2476
Without Type Prediction	3.381	1.359	856
Without Deferred Block Creation	9.089	2.901	856
Without Exposed Block Analysis	0.824	3.966	1524
Without CSE	0.484	0.936	3376
Without CSE of Constants	0.485	0.965	5864
Without CSE of Arithmetic Operations	0.49	0.96	1984
Without CSE of Memory References	0.49	0.96	852
Without CSE of Memory Cell Type Information	0.489	0.971	852
Without CSE of Memory Cell Array Bounds Checking	0.49	0.96	856
Without Eliminating Unneeded Computations	0.492	0.988	856
Without Delay Slot Filling	0.605	0.925	856
No Integer Type Tests	0.487	0.973	856
No Boolean Type Tests	0.473	0.977	856
No Overflow Checking	0.475	0.965	964
No Array Bounds Checking	0.49	0.97	856
No Block Zapping	0.49	0.95	840
Early Block Zapping	0.49	1.0	856
Late Block Zapping	0.49	0.97	856
No Primitive Failure Checking	0.489	0.961	792
No Debugger-Visible Names	0.489	0.961	592
No Interrupt Checking at Calls	0.49	0.96	828
No Interrupt Checking at _Restarts	0.438	0.792	816
No LRU Compiled Method Reclamation Support	0.49	0.98	856
Fast	0.438	0.752	856
Fastest	0.422	0.718	856

C.12 perm

language/configuration	run	compile	space
Optimized C	0.11	2.8	2400
Smalltalk-80	1.456		
T (normal)	1.16	2.1	3600
T (integer only)	0.28	1	1300
Normal SELF	0.203	1.207	1320
No Splitting (Not Lazy)	0.44	22.8	39876
No Splitting (Lazy)	0.243	1.467	1884
Local Reluctant Splitting (Not Lazy)	0.288	4.642	7332
Local Reluctant Splitting (Lazy)	0.203	1.207	1320
Global Reluctant Splitting (Not Lazy)	0.211	3.459	5448
Global Reluctant Splitting (Lazy)	0.178	1.272	1368
Divided No Splitting (Not Lazy)	0.278	9.662	20312
Divided No Splitting (Lazy)	0.244	1.496	1884
Divided Local Reluctant Splitting (Not Lazy)	0.234	7.276	13672
Divided Local Reluctant Splitting (Lazy)	0.203	1.217	1320
Divided Global Reluctant Splitting (Not Lazy)	0.21	14.0	21952
Divided Global Reluctant Splitting (Lazy)	0.178	1.272	1368
Eager Splitting (Not Lazy)	0.203	27.787	39232
Eager Splitting (Lazy)	0.179	20.801	25156
Eager Splitting and Tail Merging (Not Lazy)	0.204	15.126	22704
Eager Splitting and Tail Merging (Lazy)	0.311	9.759	12016
Eager Splitting and Requirements Analysis (Not Lazy)	0.224	9.076	14684
Eager Splitting and Requirements Analysis (Lazy)	0.178	2.582	1372
Type Predict for Vectors (with Local Splitting)	0.203	1.217	1320
Type Predict for Vectors (with Global Splitting)	0.178	1.282	1368
Vectors Are More Common (with Local Splitting)	0.204	1.206	1320
Without Inlining	22.427	2.523	1368
Without In-Line Caching	0.204	1.216	1320
Without Compile-Time Lookup Caching	0.204	1.226	3504
Without Customization	2.911	2.059	1320
Without Value-Based Type Analysis	0.562	1.468	11440
Without Range Analysis	0.215	2.075	2784
Without Type Prediction	3.795	1.535	1320
Without Deferred Block Creation	4.252	3.198	1320
Without Exposed Block Analysis	0.625	6.195	2356
Without CSE	0.221	1.129	5368
Without CSE of Constants	0.219	1.201	6580
Without CSE of Arithmetic Operations	0.206	1.204	2104
Without CSE of Memory References	0.203	1.127	1176
Without CSE of Memory Cell Type Information	0.203	1.207	1360
Without CSE of Memory Cell Array Bounds Checking	0.204	1.206	1328
Without Eliminating Unneeded Computations	0.205	1.385	1128
Without Delay Slot Filling	0.237	1.153	1320
No Integer Type Tests	0.194	1.216	1320
No Boolean Type Tests	0.204	1.216	1320
No Overflow Checking	0.2	1.21	1492
No Array Bounds Checking	0.197	1.193	1272
No Block Zapping	0.203	1.187	1304
Early Block Zapping	0.203	1.257	1268
Late Block Zapping	0.203	1.207	1320
No Primitive Failure Checking	0.203	1.207	1264
No Debugger-Visible Names	0.203	1.207	1212
No Interrupt Checking at Calls	0.198	1.202	1320
No Interrupt Checking at _Restarts	0.201	1.089	1280
No LRU Compiled Method Reclamation Support	0.193	1.217	1264
Fast	0.19	1.03	1316
Fastest	0.17	0.97	1320

C.13 towers

language/configuration	run	compile	space
Optimized C	0.191	3.7	3100
Smalltalk-80	1.927		
T (normal)	0.73	3.4	6500
T (integer only)	0.36	2.3	5900
Normal SELF	0.399	2.061	4024
No Splitting (Not Lazy)	0.583	11.507	30260
No Splitting (Lazy)	0.426	2.084	4440
Local Reluctant Splitting (Not Lazy)	0.565	5.915	16124
Local Reluctant Splitting (Lazy)	0.399	2.061	4024
Global Reluctant Splitting (Not Lazy)	0.484	18.016	34256
Global Reluctant Splitting (Lazy)	0.382	2.178	4176
Divided No Splitting (Not Lazy)	0.449	10.541	31008
Divided No Splitting (Lazy)	0.43	2.08	4440
Divided Local Reluctant Splitting (Not Lazy)	0.436	9.474	25760
Divided Local Reluctant Splitting (Lazy)	0.399	2.061	4024
Divided Global Reluctant Splitting (Not Lazy)	0.415	16.615	37748
Divided Global Reluctant Splitting (Lazy)	0.382	2.178	4176
Eager Splitting (Not Lazy)	0.394	20.786	50072
Eager Splitting (Lazy)	0.38	8.86	15128
Eager Splitting and Tail Merging (Not Lazy)	0.396	13.834	33236
Eager Splitting and Tail Merging (Lazy)	0.381	5.929	9996
Eager Splitting and Requirements Analysis (Not Lazy)	0.417	15.883	31456
Eager Splitting and Requirements Analysis (Lazy)	0.42	7.35	6224
Type Predict for Vectors (with Local Splitting)	0.4	2.06	4024
Type Predict for Vectors (with Global Splitting)	0.382	2.168	4176
Vectors Are More Common (with Local Splitting)	0.399	2.061	4024
Without Inlining	9.633	3.397	4176
Without In-Line Caching	0.399	2.061	4024
Without Compile-Time Lookup Caching	0.399	2.111	7280
Without Customization	4.163	3.037	4024
Without Value-Based Type Analysis	0.635	2.385	16032
Without Range Analysis	0.403	2.537	5788
Without Type Prediction	5.188	2.222	4024
Without Deferred Block Creation	3.998	4.092	4024
Without Exposed Block Analysis	0.399	3.391	6016
Without CSE	0.426	1.844	8548
Without CSE of Constants	0.407	2.023	6492
Without CSE of Arithmetic Operations	0.4	2.05	4816
Without CSE of Memory References	0.418	1.892	3984
Without CSE of Memory Cell Type Information	0.405	2.065	4080
Without CSE of Memory Cell Array Bounds Checking	0.399	2.051	4048
Without Eliminating Unneeded Computations	0.404	2.076	3904
Without Delay Slot Filling	0.454	1.966	4128
No Integer Type Tests	0.374	2.016	4024
No Boolean Type Tests	0.399	2.071	4032
No Overflow Checking	0.396	2.044	4596
No Array Bounds Checking	0.386	2.034	3560
No Block Zapping	0.399	2.021	3992
Early Block Zapping	0.399	2.101	3628
Late Block Zapping	0.399	2.061	4024
No Primitive Failure Checking	0.402	1.998	3912
No Debugger-Visible Names	0.399	2.041	3908
No Interrupt Checking at Calls	0.393	2.037	4024
No Interrupt Checking at _Restarts	0.399	1.971	3904
No LRU Compiled Method Reclamation Support	0.388	2.062	3796
Fast	0.387	1.863	4020
Fastest	0.344	1.696	4024

C.14 queens

language/configuration	run	compile	space
Optimized C	0.092	3.1	2500
Smalltalk-80	0.8219		
T (normal)	0.64	3.4	5200
T (integer only)	0.24	1.6	1700
Normal SELF	0.174	3.116	4860
No Splitting (Not Lazy)	0.41	32.38	78500
No Splitting (Lazy)	0.229	2.941	5092
Local Reluctant Splitting (Not Lazy)	0.294	31.956	63244
Local Reluctant Splitting (Lazy)	0.174	3.116	4860
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	0.171	2.289	3576
Divided No Splitting (Not Lazy)	0.273	45.077	96248
Divided No Splitting (Lazy)	0.23	2.96	5092
Divided Local Reluctant Splitting (Not Lazy)	0.214	48.726	90740
Divided Local Reluctant Splitting (Lazy)	0.174	3.126	4860
Divided Global Reluctant Splitting (Not Lazy)	0.209	41.031	71936
Divided Global Reluctant Splitting (Lazy)	0.172	2.298	3576
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)	0.16	10.11	13160
Eager Splitting and Tail Merging (Not Lazy)	0.198	47.982	92256
Eager Splitting and Tail Merging (Lazy)	0.159	5.321	6932
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	0.173	3.137	4860
Type Predict for Vectors (with Global Splitting)	0.172	2.298	3576
Vectors Are More Common (with Local Splitting)	0.174	3.126	4860
Without Inlining	19.996	3.594	3576
Without In-Line Caching	0.175	3.125	4860
Without Compile-Time Lookup Caching	0.174	3.276	10780
Without Customization	1.628	4.992	4860
Without Value-Based Type Analysis	0.49	4.17	16060
Without Range Analysis	0.174	3.256	8572
Without Type Prediction	4.307	3.483	4860
Without Deferred Block Creation	3.212	5.808	4860
Without Exposed Block Analysis	0.291	6.879	8672
Without CSE	0.179	2.841	11344
Without CSE of Constants	0.178	3.022	13184
Without CSE of Arithmetic Operations	0.177	3.113	5268
Without CSE of Memory References	0.173	3.017	5120
Without CSE of Memory Cell Type Information	0.174	3.126	5180
Without CSE of Memory Cell Array Bounds Checking	0.174	3.116	5076
Without Eliminating Unneeded Computations	0.174	3.466	4748
Without Delay Slot Filling	0.213	3.027	4860
No Integer Type Tests	0.174	3.076	4748
No Boolean Type Tests	0.167	3.043	4860
No Overflow Checking	0.168	3.022	5388
No Array Bounds Checking	0.164	3.016	4660
No Block Zapping	0.173	3.107	4836
Early Block Zapping	0.174	3.206	4808
Late Block Zapping	0.174	3.116	4860
No Primitive Failure Checking	0.175	3.105	4764
No Debugger-Visible Names	0.174	3.096	2788
No Interrupt Checking at Calls	0.173	3.117	4464
No Interrupt Checking at _Restarts	0.163	2.027	4364
No LRU Compiled Method Reclamation Support	0.173	3.127	4260
Fast	0.162	1.918	4860
Fastest	0.137	1.743	4860

C.15 intmm

language/configuration	run	compile	space
Optimized C	0.278	2.9	2500
Smalltalk-80	2.788		
T (normal)	2.5	3	5400
T (integer only)	0.9	1.5	2100
Normal SELF	0.723	1.987	2496
No Splitting (Not Lazy)	0.94	52.76	85436
No Splitting (Lazy)	0.77	1.99	2824
Local Reluctant Splitting (Not Lazy)	0.788	18.732	29640
Local Reluctant Splitting (Lazy)	0.723	1.987	2496
Global Reluctant Splitting (Not Lazy)	0.752	14.328	32248
Global Reluctant Splitting (Lazy)	0.725	1.995	2508
Divided No Splitting (Not Lazy)	0.795	42.055	72972
Divided No Splitting (Lazy)	0.786	1.984	2824
Divided Local Reluctant Splitting (Not Lazy)	0.746	31.034	48436
Divided Local Reluctant Splitting (Lazy)	0.724	1.966	2496
Divided Global Reluctant Splitting (Not Lazy)	0.749	28.351	46752
Divided Global Reluctant Splitting (Lazy)	0.724	1.996	2508
Eager Splitting (Not Lazy)	0.738	72.432	108444
Eager Splitting (Lazy)	0.73	45.75	62632
Eager Splitting and Tail Merging (Not Lazy)	0.744	35.706	54696
Eager Splitting and Tail Merging (Lazy)	0.728	22.732	31548
Eager Splitting and Requirements Analysis (Not Lazy)	0.757	45.983	60576
Eager Splitting and Requirements Analysis (Lazy)	0.724	4.096	2516
Type Predict for Vectors (with Local Splitting)	0.582	3.928	4444
Type Predict for Vectors (with Global Splitting)	0.571	4.679	6372
Vectors Are More Common (with Local Splitting)	0.582	4.648	4996
Without Inlining	25.486	2.064	6372
Without In-Line Caching	0.725	1.985	4996
Without Compile-Time Lookup Caching	0.724	2.016	6620
Without Customization	3.578	5.502	4444
Without Value-Based Type Analysis	1.131	2.259	9360
Without Range Analysis	0.762	4.158	3436
Without Type Prediction	0.864	2.106	2496
Without Deferred Block Creation	4.866	4.024	2496
Without Exposed Block Analysis	0.934	8.926	3588
Without CSE	0.725	1.925	6608
Without CSE of Constants	0.725	1.975	12500
Without CSE of Arithmetic Operations	0.724	1.956	4856
Without CSE of Memory References	0.724	1.946	2508
Without CSE of Memory Cell Type Information	0.724	1.986	2496
Without CSE of Memory Cell Array Bounds Checking	0.724	1.966	2496
Without Eliminating Unneeded Computations	0.729	2.001	2504
Without Delay Slot Filling	0.819	1.901	2496
No Integer Type Tests	0.699	1.951	2496
No Boolean Type Tests	0.725	1.995	2496
No Overflow Checking	0.72	1.93	2876
No Array Bounds Checking	0.682	1.968	2328
No Block Zapping	0.724	2.006	2448
Early Block Zapping	0.725	2.085	2356
Late Block Zapping	0.723	1.987	2496
No Primitive Failure Checking	0.711	1.929	2304
No Debugger-Visible Names	0.725	1.955	2292
No Interrupt Checking at Calls	0.707	1.943	2496
No Interrupt Checking at _Restarts	0.716	1.814	2356
No LRU Compiled Method Reclamation Support	0.69	1.98	2416
Fast	0.681	1.719	2496
Fastest	0.596	1.564	2496

C.16 quick

language/configuration	run	compile	space
Optimized C	0.13	3	2800
Smalltalk-80	1.276		
T (normal)	1.545	3.4	5800
T (integer only)	0.65	1.7	2700
Normal SELF	0.289	2.371	3604
No Splitting (Not Lazy)	0.568	20.922	48408
No Splitting (Lazy)	0.356	2.184	3980
Local Reluctant Splitting (Not Lazy)	0.428	15.962	33740
Local Reluctant Splitting (Lazy)	0.289	2.371	3604
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	0.283	2.917	3796
Divided No Splitting (Not Lazy)	0.462	40.998	91292
Divided No Splitting (Lazy)	0.357	2.193	3980
Divided Local Reluctant Splitting (Not Lazy)	0.365	38.435	75676
Divided Local Reluctant Splitting (Lazy)	0.29	2.37	3604
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	0.282	2.778	3796
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)			
Eager Splitting and Tail Merging (Not Lazy)			
Eager Splitting and Tail Merging (Lazy)			
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	0.29	2.36	3604
Type Predict for Vectors (with Global Splitting)	0.282	2.768	3796
Vectors Are More Common (with Local Splitting)	0.29	2.37	3604
Without Inlining	29.753	3.797	3796
Without In-Line Caching	0.294	2.376	3604
Without Compile-Time Lookup Caching	0.294	2.486	9280
Without Customization	2.506	4.284	3604
Without Value-Based Type Analysis	0.657	3.463	17148
Without Range Analysis	0.304	2.736	8716
Without Type Prediction	4.006	3.404	3604
Without Deferred Block Creation	4.449	4.551	3608
Without Exposed Block Analysis	0.449	5.261	7692
Without CSE	0.281	2.199	8888
Without CSE of Constants	0.276	2.304	9808
Without CSE of Arithmetic Operations	0.29	2.34	4400
Without CSE of Memory References	0.294	2.316	3640
Without CSE of Memory Cell Type Information	0.291	2.389	3476
Without CSE of Memory Cell Array Bounds Checking	0.291	2.369	3604
Without Eliminating Unneeded Computations	0.296	2.394	3748
Without Delay Slot Filling	0.341	2.269	3676
No Integer Type Tests	0.279	2.311	3604
No Boolean Type Tests	0.294	2.376	3624
No Overflow Checking	0.289	2.331	4032
No Array Bounds Checking	0.262	2.318	3212
No Block Zapping	0.29	2.31	3576
Early Block Zapping	0.29	2.41	3428
Late Block Zapping	0.289	2.371	3604
No Primitive Failure Checking	0.287	2.333	3496
No Debugger-Visible Names	0.29	2.38	2948
No Interrupt Checking at Calls	0.289	2.351	3604
No Interrupt Checking at _Restarts	0.262	2.088	3508
No LRU Compiled Method Reclamation Support	0.288	2.362	3312
Fast	0.259	1.961	3556
Fastest	0.218	1.782	3568

C.17 bubble

language/configuration	run	compile	space
Optimized C	0.195	2.9	2700
Smalltalk-80	2.83		
T (normal)	1	2.7	4700
T (integer only)	0.34	1.3	1900
Normal SELF	0.271	1.669	2428
No Splitting (Not Lazy)	0.849	49.191	113356
No Splitting (Lazy)	0.375	1.615	2752
Local Reluctant Splitting (Not Lazy)	0.486	5.364	10780
Local Reluctant Splitting (Lazy)	0.271	1.669	2428
Global Reluctant Splitting (Not Lazy)	0.439	6.651	12952
Global Reluctant Splitting (Lazy)	0.272	1.808	2572
Divided No Splitting (Not Lazy)	0.407	44.173	105364
Divided No Splitting (Lazy)	0.375	1.615	2752
Divided Local Reluctant Splitting (Not Lazy)	0.325	8.025	15544
Divided Local Reluctant Splitting (Lazy)	0.272	1.668	2428
Divided Global Reluctant Splitting (Not Lazy)	0.325	9.595	18012
Divided Global Reluctant Splitting (Lazy)	0.272	1.818	2572
Eager Splitting (Not Lazy)	0.291	19.319	38568
Eager Splitting (Lazy)	0.268	6.222	9616
Eager Splitting and Tail Merging (Not Lazy)	0.292	10.088	19984
Eager Splitting and Tail Merging (Lazy)	0.268	3.602	5400
Eager Splitting and Requirements Analysis (Not Lazy)	0.357	61.083	114656
Eager Splitting and Requirements Analysis (Lazy)	0.275	3.835	2852
Type Predict for Vectors (with Local Splitting)	0.271	1.669	2428
Type Predict for Vectors (with Global Splitting)	0.272	1.808	2572
Vectors Are More Common (with Local Splitting)	0.272	1.668	2428
Without Inlining	37.954	3.416	2572
Without In-Line Caching	0.272	1.678	2428
Without Compile-Time Lookup Caching	0.272	1.748	6836
Without Customization	6.37	3.69	2428
Without Value-Based Type Analysis	0.865	2.095	14764
Without Range Analysis	0.28	2.67	5376
Without Type Prediction	5.832	2.128	2428
Without Deferred Block Creation	8.735	4.985	2428
Without Exposed Block Analysis	0.404	4.296	4384
Without CSE	0.409	1.621	6496
Without CSE of Constants	0.279	1.651	7060
Without CSE of Arithmetic Operations	0.338	1.692	3648
Without CSE of Memory References	0.366	1.664	2856
Without CSE of Memory Cell Type Information	0.319	1.711	2444
Without CSE of Memory Cell Array Bounds Checking	0.28	1.69	2564
Without Eliminating Unneeded Computations	0.282	1.688	2748
Without Delay Slot Filling	0.353	1.597	2576
No Integer Type Tests	0.249	1.621	2476
No Boolean Type Tests	0.272	1.688	2464
No Overflow Checking	0.271	1.659	2716
No Array Bounds Checking	0.229	1.641	2188
No Block Zapping	0.273	1.607	2404
Early Block Zapping	0.271	1.709	2292
Late Block Zapping	0.271	1.669	2428
No Primitive Failure Checking	0.272	1.638	2348
No Debugger-Visible Names	0.272	1.668	2144
No Interrupt Checking at Calls	0.272	1.658	2428
No Interrupt Checking at _Restarts	0.257	1.543	2368
No LRU Compiled Method Reclamation Support	0.272	1.678	2276
Fast	0.256	1.444	2380
Fastest	0.191	1.289	2388

C.18 tree

language/configuration	run	compile	space
Optimized C	0.869	3.9	3300
Smalltalk-80	1.658		
T (normal)	1.25	3.5	5800
T (integer only)	0.96	2.4	3600
Normal SELF	1.114	1.566	3260
No Splitting (Not Lazy)	1.333	13.057	29736
No Splitting (Lazy)	1.205	1.555	3672
Local Reluctant Splitting (Not Lazy)	1.164	4.676	10624
Local Reluctant Splitting (Lazy)	1.114	1.566	3260
Global Reluctant Splitting (Not Lazy)	1.232	5.618	12792
Global Reluctant Splitting (Lazy)	1.116	1.604	3272
Divided No Splitting (Not Lazy)	1.274	14.456	33260
Divided No Splitting (Lazy)	1.233	1.557	3672
Divided Local Reluctant Splitting (Not Lazy)	1.16	7.4	16184
Divided Local Reluctant Splitting (Lazy)	1.114	1.576	3260
Divided Global Reluctant Splitting (Not Lazy)	1.242	8.438	18344
Divided Global Reluctant Splitting (Lazy)	1.114	1.606	3272
Eager Splitting (Not Lazy)	1.231	17.169	37792
Eager Splitting (Lazy)	1.199	5.131	10092
Eager Splitting and Tail Merging (Not Lazy)	1.215	8.385	18148
Eager Splitting and Tail Merging (Lazy)	1.117	2.593	4436
Eager Splitting and Requirements Analysis (Not Lazy)	1.231	23.489	41184
Eager Splitting and Requirements Analysis (Lazy)	1.196	4.024	4328
Type Predict for Vectors (with Local Splitting)	1.114	1.576	3260
Type Predict for Vectors (with Global Splitting)	1.116	1.594	3272
Vectors Are More Common (with Local Splitting)	1.115	1.575	3260
Without Inlining	22.771	2.529	3272
Without In-Line Caching	1.124	1.576	3260
Without Compile-Time Lookup Caching	1.127	1.603	8024
Without Customization	2.004	3.096	3260
Without Value-Based Type Analysis	1.324	1.846	16632
Without Range Analysis	1.12	2.18	7548
Without Type Prediction	12.748	1.762	3260
Without Deferred Block Creation	6.801	1.399	3260
Without Exposed Block Analysis	1.213	2.767	4628
Without CSE	1.12	1.52	6512
Without CSE of Constants	1.116	1.544	5408
Without CSE of Arithmetic Operations	1.115	1.575	4116
Without CSE of Memory References	1.119	1.561	3524
Without CSE of Memory Cell Type Information	1.116	1.594	3344
Without CSE of Memory Cell Array Bounds Checking	1.115	1.575	3260
Without Eliminating Unneeded Computations	1.137	1.583	3412
Without Delay Slot Filling	1.197	1.513	3344
No Integer Type Tests	1.092	1.538	3260
No Boolean Type Tests	1.124	1.566	3280
No Overflow Checking	1.115	1.545	3768
No Array Bounds Checking	1.114	1.576	2988
No Block Zapping	1.116	1.584	3220
Early Block Zapping	1.115	1.645	3176
Late Block Zapping	1.114	1.566	3260
No Primitive Failure Checking	1.113	1.547	3124
No Debugger-Visible Names	1.116	1.564	3212
No Interrupt Checking at Calls	1.106	1.544	3160
No Interrupt Checking at _Restarts	1.115	1.525	3200
No LRU Compiled Method Reclamation Support	1.095	1.555	3260
Fast	1.094	1.456	3260
Fastest	1.142	1.268	3260

C.19 oo-perm

language/configuration	run	compile	space
Optimized C	0.11	2.8	2400
Smalltalk-80	1.381		
T (normal)	1.16	2.1	3600
T (integer only)	0.28	1	1300
Normal SELF	0.2	1.82	2040
No Splitting (Not Lazy)	0.419	30.081	54536
No Splitting (Lazy)	0.24	2.44	3168
Local Reluctant Splitting (Not Lazy)	0.285	6.365	10400
Local Reluctant Splitting (Lazy)	0.2	1.82	2040
Global Reluctant Splitting (Not Lazy)	0.217	5.483	8280
Global Reluctant Splitting (Lazy)	0.182	1.948	2136
Divided No Splitting (Not Lazy)	0.279	18.181	37672
Divided No Splitting (Lazy)	0.241	2.459	3168
Divided Local Reluctant Splitting (Not Lazy)	0.235	13.495	24372
Divided Local Reluctant Splitting (Lazy)	0.201	1.829	2040
Divided Global Reluctant Splitting (Not Lazy)	0.216	23.724	39156
Divided Global Reluctant Splitting (Lazy)	0.182	1.958	2136
Eager Splitting (Not Lazy)	0.21	41.19	60072
Eager Splitting (Lazy)	0.185	25.415	29152
Eager Splitting and Tail Merging (Not Lazy)	0.331	21.779	33284
Eager Splitting and Tail Merging (Lazy)	0.184	10.936	13024
Eager Splitting and Requirements Analysis (Not Lazy)	0.241	16.519	26340
Eager Splitting and Requirements Analysis (Lazy)	0.183	4.067	2132
Type Predict for Vectors (with Local Splitting)	0.201	1.809	2040
Type Predict for Vectors (with Global Splitting)	0.183	1.947	2136
Vectors Are More Common (with Local Splitting)	0.201	1.819	2040
Without Inlining	22.891	2.929	2136
Without In-Line Caching	0.201	1.829	2040
Without Compile-Time Lookup Caching	0.201	1.849	4236
Without Customization	2.891	2.409	2040
Without Value-Based Type Analysis	0.729	2.461	12664
Without Range Analysis	0.204	2.676	3924
Without Type Prediction	3.769	1.961	2040
Without Deferred Block Creation	4.256	5.334	2040
Without Exposed Block Analysis	0.625	7.605	4308
Without CSE	0.248	1.732	8908
Without CSE of Constants	0.322	1.818	8344
Without CSE of Arithmetic Operations	0.203	1.807	2828
Without CSE of Memory References	0.243	1.767	2064
Without CSE of Memory Cell Type Information	0.201	1.819	2040
Without CSE of Memory Cell Array Bounds Checking	0.21	1.87	2056
Without Eliminating Unneeded Computations	0.21	2.38	2048
Without Delay Slot Filling	0.239	1.751	2040
No Integer Type Tests	0.192	1.808	2184
No Boolean Type Tests	0.202	1.818	2040
No Overflow Checking	0.198	1.812	2304
No Array Bounds Checking	0.181	1.739	1944
No Block Zapping	0.201	1.819	2016
Early Block Zapping	0.201	1.899	1988
Late Block Zapping	0.2	1.82	2040
No Primitive Failure Checking	0.201	1.819	1952
No Debugger-Visible Names	0.201	1.819	1892
No Interrupt Checking at Calls	0.196	1.804	2040
No Interrupt Checking at _Restarts	0.197	1.663	1980
No LRU Compiled Method Reclamation Support	0.19	1.82	1760
Fast	0.187	1.573	2036
Fastest	0.154	1.456	2040

C.20 oo-towers

language/configuration	run	compile	space
Optimized C	0.191	3.7	3100
Smalltalk-80	1.032		
T (normal)	0.73	3.4	6500
T (integer only)	0.36	2.3	5900
Normal SELF	0.223	1.077	2588
No Splitting (Not Lazy)	0.325	5.135	14612
No Splitting (Lazy)	0.247	1.173	2952
Local Reluctant Splitting (Not Lazy)	0.269	3.811	10624
Local Reluctant Splitting (Lazy)	0.223	1.077	2588
Global Reluctant Splitting (Not Lazy)	0.263	6.837	16204
Global Reluctant Splitting (Lazy)	0.231	1.089	2612
Divided No Splitting (Not Lazy)	0.258	6.322	19244
Divided No Splitting (Lazy)	0.246	1.194	2952
Divided Local Reluctant Splitting (Not Lazy)	0.24	5.8	16420
Divided Local Reluctant Splitting (Lazy)	0.223	1.087	2588
Divided Global Reluctant Splitting (Not Lazy)	0.249	6.521	17064
Divided Global Reluctant Splitting (Lazy)	0.231	1.089	2612
Eager Splitting (Not Lazy)	0.244	9.816	26940
Eager Splitting (Lazy)	0.236	2.434	5484
Eager Splitting and Tail Merging (Not Lazy)	0.24	6.18	17180
Eager Splitting and Tail Merging (Lazy)	0.25	1.44	3240
Eager Splitting and Requirements Analysis (Not Lazy)	0.258	7.192	18456
Eager Splitting and Requirements Analysis (Lazy)	0.23	1.72	2660
Type Predict for Vectors (with Local Splitting)	0.223	1.077	2588
Type Predict for Vectors (with Global Splitting)	0.231	1.089	2612
Vectors Are More Common (with Local Splitting)	0.223	1.067	2588
Without Inlining	3.578	2.472	2612
Without In-Line Caching	0.223	1.077	2588
Without Compile-Time Lookup Caching	0.224	1.076	4500
Without Customization	1.52	1.55	2588
Without Value-Based Type Analysis	0.324	1.316	10780
Without Range Analysis	0.223	1.237	3812
Without Type Prediction	1.943	1.227	2588
Without Deferred Block Creation	1.625	1.685	2588
Without Exposed Block Analysis	0.224	1.536	3536
Without CSE	0.231	0.979	4656
Without CSE of Constants	0.225	1.055	3416
Without CSE of Arithmetic Operations	0.224	1.076	2872
Without CSE of Memory References	0.227	1.003	2464
Without CSE of Memory Cell Type Information	0.223	1.067	2624
Without CSE of Memory Cell Array Bounds Checking	0.223	1.077	2612
Without Eliminating Unneeded Computations	0.225	1.085	2400
Without Delay Slot Filling	0.253	1.027	2588
No Integer Type Tests	0.209	1.051	2588
No Boolean Type Tests	0.224	1.076	2592
No Overflow Checking	0.221	1.049	3000
No Array Bounds Checking	0.218	1.072	2340
No Block Zapping	0.223	1.077	2556
Early Block Zapping	0.223	1.107	2352
Late Block Zapping	0.223	1.077	2588
No Primitive Failure Checking	0.263	1.027	2476
No Debugger-Visible Names	0.223	1.067	2572
No Interrupt Checking at Calls	0.217	1.053	2588
No Interrupt Checking at _Restarts	0.223	1.047	2476
No LRU Compiled Method Reclamation Support	0.211	1.069	2524
Fast	0.211	0.999	2584
Fastest	0.191	0.889	2588

C.21 oo-queens

language/configuration	run	compile	space
Optimized C	0.092	3.1	2500
Smalltalk-80	0.72		
T (normal)	0.64	3.4	5200
T (integer only)	0.24	1.6	1700
Normal SELF	0.147	3.973	4524
No Splitting (Not Lazy)	0.316	38.254	71652
No Splitting (Lazy)	0.187	3.803	4896
Local Reluctant Splitting (Not Lazy)	0.179	16.311	25100
Local Reluctant Splitting (Lazy)	0.147	3.973	4524
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	0.141	3.119	3696
Divided No Splitting (Not Lazy)	0.204	50.326	86244
Divided No Splitting (Lazy)	0.186	3.814	4896
Divided Local Reluctant Splitting (Not Lazy)	0.168	52.092	78468
Divided Local Reluctant Splitting (Lazy)	0.147	4.003	4524
Divided Global Reluctant Splitting (Not Lazy)	0.164	37.206	57100
Divided Global Reluctant Splitting (Lazy)	0.141	3.159	3696
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)	0.132	13.338	13312
Eager Splitting and Tail Merging (Not Lazy)	0.172	53.808	78940
Eager Splitting and Tail Merging (Lazy)	0.133	8.097	8572
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	0.147	3.973	4524
Type Predict for Vectors (with Global Splitting)	0.141	3.109	3696
Vectors Are More Common (with Local Splitting)	0.147	3.973	4524
Without Inlining	21.28	2.67	3696
Without In-Line Caching	0.147	3.993	4524
Without Compile-Time Lookup Caching	0.147	4.213	9244
Without Customization	1.529	5.271	4524
Without Value-Based Type Analysis	0.314	2.396	11964
Without Range Analysis	0.156	4.664	3500
Without Type Prediction	3.418	2.152	4524
Without Deferred Block Creation	3.153	7.427	4524
Without Exposed Block Analysis	0.345	8.135	3496
Without CSE	0.144	3.646	13136
Without CSE of Constants	0.145	3.935	11964
Without CSE of Arithmetic Operations	0.151	3.999	5484
Without CSE of Memory References	0.145	3.765	4496
Without CSE of Memory Cell Type Information	0.147	3.973	4668
Without CSE of Memory Cell Array Bounds Checking	0.145	3.865	4692
Without Eliminating Unneeded Computations	0.148	4.572	4232
Without Delay Slot Filling	0.18	3.83	4524
No Integer Type Tests	0.147	3.923	4232
No Boolean Type Tests	0.137	3.873	4524
No Overflow Checking	0.143	3.837	5108
No Array Bounds Checking	0.142	3.838	4356
No Block Zapping	0.147	4.013	4500
Early Block Zapping	0.147	4.263	4472
Late Block Zapping	0.147	3.973	4524
No Primitive Failure Checking	0.147	3.983	4428
No Debugger-Visible Names	0.148	3.952	4176
No Interrupt Checking at Calls	0.146	3.944	4148
No Interrupt Checking at _Restarts	0.141	3.889	4100
No LRU Compiled Method Reclamation Support	0.145	3.975	4164
Fast	0.139	3.681	4524
Fastest	0.12	3.21	4524

C.22 oo-intmm

language/configuration	run	compile	space
Optimized C	0.278	2.9	2500
Smalltalk-80	4.638		
T (normal)	2.5	3	5400
T (integer only)	0.9	1.5	2100
Normal SELF	0.691	2.239	2728
No Splitting (Not Lazy)	0.878	56.312	85744
No Splitting (Lazy)	0.72	2.83	3680
Local Reluctant Splitting (Not Lazy)	0.752	20.318	29908
Local Reluctant Splitting (Lazy)	0.691	2.239	2728
Global Reluctant Splitting (Not Lazy)	0.729	11.801	20104
Global Reluctant Splitting (Lazy)	0.69	2.25	2740
Divided No Splitting (Not Lazy)	0.746	53.534	83716
Divided No Splitting (Lazy)	0.729	2.851	3680
Divided Local Reluctant Splitting (Not Lazy)	0.902	35.178	51796
Divided Local Reluctant Splitting (Lazy)	0.69	2.23	2728
Divided Global Reluctant Splitting (Not Lazy)	0.717	35.473	56084
Divided Global Reluctant Splitting (Lazy)	0.69	2.24	2740
Eager Splitting (Not Lazy)	0.716	80.774	113524
Eager Splitting (Lazy)	0.695	49.565	63048
Eager Splitting and Tail Merging (Not Lazy)	0.71	39.82	57368
Eager Splitting and Tail Merging (Lazy)	0.696	24.614	31852
Eager Splitting and Requirements Analysis (Not Lazy)	0.738	47.472	64480
Eager Splitting and Requirements Analysis (Lazy)	0.694	4.936	2748
Type Predict for Vectors (with Local Splitting)	0.577	4.223	4380
Type Predict for Vectors (with Global Splitting)	0.518	4.862	6268
Vectors Are More Common (with Local Splitting)	0.508	4.672	4608
Without Inlining	27.52	2.34	6268
Without In-Line Caching	0.691	2.239	4608
Without Compile-Time Lookup Caching	0.691	2.259	5724
Without Customization	6.772	2.218	4380
Without Value-Based Type Analysis	1.203	2.517	10840
Without Range Analysis	0.702	4.638	3524
Without Type Prediction	0.798	2.352	2728
Without Deferred Block Creation	4.252	5.128	2728
Without Exposed Block Analysis	1.185	9.775	3864
Without CSE	0.704	2.126	8224
Without CSE of Constants	0.69	2.23	12884
Without CSE of Arithmetic Operations	0.69	2.21	5024
Without CSE of Memory References	0.703	2.167	2748
Without CSE of Memory Cell Type Information	0.689	2.231	2728
Without CSE of Memory Cell Array Bounds Checking	0.691	2.219	2728
Without Eliminating Unneeded Computations	0.697	2.253	2744
Without Delay Slot Filling	0.785	2.135	2728
No Integer Type Tests	0.665	2.205	2728
No Boolean Type Tests	0.691	2.239	2728
No Overflow Checking	0.687	2.183	3164
No Array Bounds Checking	0.64	2.18	2584
No Block Zapping	0.69	2.27	2672
Early Block Zapping	0.691	2.349	2588
Late Block Zapping	0.691	2.239	2728
No Primitive Failure Checking	0.677	2.173	2504
No Debugger-Visible Names	0.692	2.198	2524
No Interrupt Checking at Calls	0.679	2.171	2728
No Interrupt Checking at _Restarts	0.691	2.039	2588
No LRU Compiled Method Reclamation Support	0.656	2.214	2568
Fast	0.648	1.942	2728
Fastest	0.555	1.775	2728

C.23 oo-quick

language/configuration	run	compile	space
Optimized C	0.13	3	2800
Smalltalk-80	2.705		
T (normal)	1.545	3.4	5800
T (integer only)	0.65	1.7	2700
Normal SELF	0.257	2.933	3964
No Splitting (Not Lazy)	0.53	21.77	48916
No Splitting (Lazy)	0.318	2.712	4312
Local Reluctant Splitting (Not Lazy)	0.429	19.131	38632
Local Reluctant Splitting (Lazy)	0.257	2.933	3964
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	0.252	3.328	4556
Divided No Splitting (Not Lazy)	0.373	46.837	100160
Divided No Splitting (Lazy)	0.316	2.734	4312
Divided Local Reluctant Splitting (Not Lazy)	0.324	52.336	98508
Divided Local Reluctant Splitting (Lazy)	0.257	2.923	3964
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	0.252	3.328	4556
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)			
Eager Splitting and Tail Merging (Not Lazy)			
Eager Splitting and Tail Merging (Lazy)			
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	0.257	2.933	3964
Type Predict for Vectors (with Global Splitting)	0.251	3.339	4556
Vectors Are More Common (with Local Splitting)	0.258	2.942	3964
Without Inlining	30.654	3.906	4556
Without In-Line Caching	0.271	2.919	3964
Without Compile-Time Lookup Caching	0.27	3.09	8756
Without Customization	2.448	4.032	3964
Without Value-Based Type Analysis	0.734	4.326	17436
Without Range Analysis	0.262	2.838	9332
Without Type Prediction	3.884	3.986	3964
Without Deferred Block Creation	4.439	5.671	3964
Without Exposed Block Analysis	0.36	5.45	9036
Without CSE	0.289	3.381	10396
Without CSE of Constants	0.248	2.872	9632
Without CSE of Arithmetic Operations	0.257	2.883	4156
Without CSE of Memory References	0.297	3.473	5228
Without CSE of Memory Cell Type Information	0.26	3.46	3868
Without CSE of Memory Cell Array Bounds Checking	0.26	3.02	3964
Without Eliminating Unneeded Computations	0.28	2.96	5300
Without Delay Slot Filling	0.32	2.81	4656
No Integer Type Tests	0.244	2.876	4252
No Boolean Type Tests	0.277	2.933	4004
No Overflow Checking	0.255	2.895	4476
No Array Bounds Checking	0.24	2.83	3612
No Block Zapping	0.257	2.883	3932
Early Block Zapping	0.257	3.013	3736
Late Block Zapping	0.257	2.933	3964
No Primitive Failure Checking	0.254	2.866	3852
No Debugger-Visible Names	0.257	2.923	3300
No Interrupt Checking at Calls	0.256	2.914	3964
No Interrupt Checking at _Restarts	0.237	2.653	3836
No LRU Compiled Method Reclamation Support	0.255	2.925	3568
Fast	0.234	2.516	3916
Fastest	0.199	2.261	3924

C.24 oo-bubble

language/configuration	run	compile	space
Optimized C	0.195	2.9	2700
Smalltalk-80	2.587		
T (normal)	1	2.7	4700
T (integer only)	0.34	1.3	1900
Normal SELF	0.235	2.195	2644
No Splitting (Not Lazy)	0.773	63.097	98548
No Splitting (Lazy)	0.331	2.119	2924
Local Reluctant Splitting (Not Lazy)	0.435	8.785	14836
Local Reluctant Splitting (Lazy)	0.235	2.195	2644
Global Reluctant Splitting (Not Lazy)	0.394	58.526	104520
Global Reluctant Splitting (Lazy)	0.235	2.475	3188
Divided No Splitting (Not Lazy)	0.368	57.582	96232
Divided No Splitting (Lazy)	0.332	2.128	2924
Divided Local Reluctant Splitting (Not Lazy)	0.289	21.091	35112
Divided Local Reluctant Splitting (Lazy)	0.235	2.195	2644
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	0.235	2.475	3188
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)	0.258	7.832	7816
Eager Splitting and Tail Merging (Not Lazy)	0.258	28.252	44788
Eager Splitting and Tail Merging (Lazy)	0.231	4.449	4108
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)	0.238	6.842	3452
Type Predict for Vectors (with Local Splitting)	0.235	2.185	2644
Type Predict for Vectors (with Global Splitting)	0.234	2.466	3188
Vectors Are More Common (with Local Splitting)	0.235	2.205	2644
Without Inlining	48.937	4.183	3188
Without In-Line Caching	0.235	2.195	2644
Without Compile-Time Lookup Caching	0.235	2.275	5816
Without Customization	5.831	3.259	2644
Without Value-Based Type Analysis	1.162	3.018	15400
Without Range Analysis	0.243	2.617	5556
Without Type Prediction	5.71	2.6	2644
Without Deferred Block Creation	9.001	6.209	2644
Without Exposed Block Analysis	0.501	6.759	5312
Without CSE	0.444	2.736	7540
Without CSE of Constants	0.235	2.165	8528
Without CSE of Arithmetic Operations	0.293	2.177	3084
Without CSE of Memory References	0.424	2.786	4060
Without CSE of Memory Cell Type Information	0.275	2.735	2640
Without CSE of Memory Cell Array Bounds Checking	0.259	2.311	2704
Without Eliminating Unneeded Computations	0.245	2.215	4032
Without Delay Slot Filling	0.319	2.101	3360
No Integer Type Tests	0.211	2.149	2964
No Boolean Type Tests	0.235	2.195	2692
No Overflow Checking	0.234	2.166	2980
No Array Bounds Checking	0.197	2.143	2476
No Block Zapping	0.235	2.155	2612
Early Block Zapping	0.234	2.276	2452
Late Block Zapping	0.235	2.195	2644
No Primitive Failure Checking	0.235	2.145	2532
No Debugger-Visible Names	0.235	2.185	2404
No Interrupt Checking at Calls	0.234	2.166	2644
No Interrupt Checking at _Restarts	0.219	2.011	2544
No LRU Compiled Method Reclamation Support	0.235	2.205	2524
Fast	0.219	1.881	2596
Fastest	0.157	1.693	2604

C.25 oo-tree

language/configuration	run	compile	space
Optimized C	0.869	3.9	3300
Smalltalk-80	1.101		
T (normal)	1.25	3.5	5800
T (integer only)	0.96	2.4	3600
Normal SELF	0.874	2.476	4016
No Splitting (Not Lazy)	1.143	15.377	32460
No Splitting (Lazy)	0.984	2.406	4696
Local Reluctant Splitting (Not Lazy)	0.95	9.01	17260
Local Reluctant Splitting (Lazy)	0.874	2.476	4016
Global Reluctant Splitting (Not Lazy)	1.03	51.22	106636
Global Reluctant Splitting (Lazy)	0.876	2.754	4428
Divided No Splitting (Not Lazy)	1.041	21.419	45016
Divided No Splitting (Lazy)	0.984	2.406	4696
Divided Local Reluctant Splitting (Not Lazy)	0.937	21.693	38604
Divided Local Reluctant Splitting (Lazy)	0.875	2.485	4016
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	0.875	2.745	4428
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)	0.951	8.459	10776
Eager Splitting and Tail Merging (Not Lazy)	0.977	29.053	48396
Eager Splitting and Tail Merging (Lazy)	0.883	4.887	5984
Eager Splitting and Requirements Analysis (Not Lazy)	0.983	35.827	55792
Eager Splitting and Requirements Analysis (Lazy)	1.0	7.18	5304
Type Predict for Vectors (with Local Splitting)	0.874	2.496	4016
Type Predict for Vectors (with Global Splitting)	0.876	2.754	4428
Vectors Are More Common (with Local Splitting)	0.875	2.485	4016
Without Inlining	26.367	3.253	4428
Without In-Line Caching	0.877	2.493	4016
Without Compile-Time Lookup Caching	0.876	2.554	9676
Without Customization	2.263	3.307	4016
Without Value-Based Type Analysis	1.181	3.159	19936
Without Range Analysis	0.878	2.642	8200
Without Type Prediction	8.63	2.71	4016
Without Deferred Block Creation	7.612	2.598	4016
Without Exposed Block Analysis	2.255	3.445	6940
Without CSE	0.903	2.907	9148
Without CSE of Constants	0.875	2.435	6368
Without CSE of Arithmetic Operations	0.875	2.475	4372
Without CSE of Memory References	0.903	3.007	5312
Without CSE of Memory Cell Type Information	0.88	3.03	4024
Without CSE of Memory Cell Array Bounds Checking	0.878	2.552	4016
Without Eliminating Unneeded Computations	0.883	2.517	5288
Without Delay Slot Filling	0.939	2.391	4708
No Integer Type Tests	0.862	2.438	4256
No Boolean Type Tests	0.877	2.503	4056
No Overflow Checking	0.873	2.467	4616
No Array Bounds Checking	0.885	2.475	3656
No Block Zapping	0.875	2.465	3960
Early Block Zapping	0.875	2.585	3876
Late Block Zapping	0.874	2.476	4016
No Primitive Failure Checking	0.871	2.459	3848
No Debugger-Visible Names	0.876	2.484	3956
No Interrupt Checking at Calls	0.866	2.454	3980
No Interrupt Checking at _Restarts	0.875	2.425	3916
No LRU Compiled Method Reclamation Support	0.856	2.494	3968
Fast	0.853	2.287	4000
Fastest	0.905	2.045	4016

C.26 puzzle

language/configuration	run	compile	space
Optimized C	0.69	9.1	5000
Smalltalk-80	16.058		
T (normal)	4.5	24	32000
T (integer only)	3.1	9.1	9900
Normal SELF	3.107	19.293	16688
No Splitting (Not Lazy)			
No Splitting (Lazy)	3.538	19.392	19348
Local Reluctant Splitting (Not Lazy)			
Local Reluctant Splitting (Lazy)	3.107	19.293	16688
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	3.115	19.785	17788
Divided No Splitting (Not Lazy)			
Divided No Splitting (Lazy)	3.535	19.465	19348
Divided Local Reluctant Splitting (Not Lazy)			
Divided Local Reluctant Splitting (Lazy)	3.11	19.3	16688
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	3.116	19.784	17788
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)			
Eager Splitting and Tail Merging (Not Lazy)			
Eager Splitting and Tail Merging (Lazy)			
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	2.394	21.566	19232
Type Predict for Vectors (with Global Splitting)	2.398	23.522	22144
Vectors Are More Common (with Local Splitting)	2.431	19.889	17532
Without Inlining	506.631	14.849	22144
Without In-Line Caching	3.102	19.438	17532
Without Compile-Time Lookup Caching	3.099	19.911	33832
Without Customization	21.77	25.59	19232
Without Value-Based Type Analysis	5.139	19.541	46652
Without Range Analysis	3.111	71.609	19688
Without Type Prediction	21.622	18.878	16688
Without Deferred Block Creation	39.234	29.576	16688
Without Exposed Block Analysis	4.453	256.907	20968
Without CSE	3.029	18.681	61728
Without CSE of Constants	3.026	19.334	299328
Without CSE of Arithmetic Operations	3.111	19.069	55240
Without CSE of Memory References	3.107	18.983	16536
Without CSE of Memory Cell Type Information	3.105	19.335	16616
Without CSE of Memory Cell Array Bounds Checking	3.104	19.236	16688
Without Eliminating Unneeded Computations	3.228	20.162	16616
Without Delay Slot Filling	3.773	18.587	16688
No Integer Type Tests	2.995	19.345	16688
No Boolean Type Tests	2.917	19.363	16696
No Overflow Checking	3.066	18.844	19308
No Array Bounds Checking	2.851	19.189	16088
No Block Zapping	3.11	19.3	16512
Early Block Zapping	3.108	20.242	16520
Late Block Zapping	3.107	19.293	16688
No Primitive Failure Checking	3.11	19.3	16000
No Debugger-Visible Names	3.107	19.203	14460
No Interrupt Checking at Calls	3.009	19.161	16144
No Interrupt Checking at _Restarts	2.989	17.341	15428
No LRU Compiled Method Reclamation Support	2.917	19.363	16084
Fast	2.767	16.433	16664
Fastest	2.21	15.41	16688

C.27 richards

language/configuration	run	compile	space
Optimized C	0.73	13.4	6100
Smalltalk-80	7.74		
T (normal)	9.8	11.5	18000
T (integer only)	8.1	13.7	18000
Normal SELF	2.27	4.39	11592
No Splitting (Not Lazy)	2.86	14.52	36996
No Splitting (Lazy)	2.536	4.274	12724
Local Reluctant Splitting (Not Lazy)	2.462	9.658	24872
Local Reluctant Splitting (Lazy)	2.27	4.39	11592
Global Reluctant Splitting (Not Lazy)	2.652	14.268	35980
Global Reluctant Splitting (Lazy)	2.292	5.388	11992
Divided No Splitting (Not Lazy)	2.692	18.538	46820
Divided No Splitting (Lazy)	2.521	4.369	12724
Divided Local Reluctant Splitting (Not Lazy)	2.306	13.664	33996
Divided Local Reluctant Splitting (Lazy)	2.282	4.368	11592
Divided Global Reluctant Splitting (Not Lazy)	2.348	16.872	37908
Divided Global Reluctant Splitting (Lazy)	2.269	5.411	11992
Eager Splitting (Not Lazy)	2.551	135.679	266876
Eager Splitting (Lazy)	2.311	84.189	166132
Eager Splitting and Tail Merging (Not Lazy)	2.504	89.966	80984
Eager Splitting and Tail Merging (Lazy)	2.446	75.254	45056
Eager Splitting and Requirements Analysis (Not Lazy)	2.465	26.685	53748
Eager Splitting and Requirements Analysis (Lazy)	2.544	8.846	15288
Type Predict for Vectors (with Local Splitting)	2.156	4.914	12452
Type Predict for Vectors (with Global Splitting)	2.308	6.122	13240
Vectors Are More Common (with Local Splitting)	2.204	4.886	12168
Without Inlining	127.488	10.342	13240
Without In-Line Caching	2.182	4.438	12168
Without Compile-Time Lookup Caching	2.185	4.465	25100
Without Customization	5.829	6.521	12452
Without Value-Based Type Analysis	2.747	4.953	41052
Without Range Analysis	2.286	4.934	16236
Without Type Prediction	39.554	6.846	11592
Without Deferred Block Creation	11.461	9.039	11592
Without Exposed Block Analysis	2.625	5.925	14680
Without CSE	2.372	3.948	17228
Without CSE of Constants	2.403	4.187	14176
Without CSE of Arithmetic Operations	2.284	4.366	12436
Without CSE of Memory References	2.34	4.14	11896
Without CSE of Memory Cell Type Information	2.235	4.395	11848
Without CSE of Memory Cell Array Bounds Checking	2.285	4.345	11592
Without Eliminating Unneeded Computations	2.365	4.965	11640
Without Delay Slot Filling	2.59	4.27	11688
No Integer Type Tests	2.209	4.371	11592
No Boolean Type Tests	2.184	4.396	11592
No Overflow Checking	2.209	4.341	13440
No Array Bounds Checking	2.185	4.385	11136
No Block Zapping	2.301	4.419	11408
Early Block Zapping	2.325	4.495	11536
Late Block Zapping	2.27	4.39	11592
No Primitive Failure Checking	2.269	4.361	10984
No Debugger-Visible Names	2.27	4.34	11520
No Interrupt Checking at Calls	2.135	4.275	11340
No Interrupt Checking at _Restarts	2.313	4.307	11400
No LRU Compiled Method Reclamation Support	2.11	4.34	11536
Fast	2.178	4.162	11592
Fastest	1.914	3.946	11592

C.28 parser

language/configuration	run	compile	space
Optimized C			
Smalltalk-80			
T (normal)			
T (integer only)			
Normal SELF	0.076	55.714	128960
No Splitting (Not Lazy)	0.106	150.324	332840
No Splitting (Lazy)	0.077	51.763	133016
Local Reluctant Splitting (Not Lazy)	0.089	107.941	242624
Local Reluctant Splitting (Lazy)	0.076	55.714	128960
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	0.083	82.177	192784
Divided No Splitting (Not Lazy)			
Divided No Splitting (Lazy)	0.078	51.962	133016
Divided Local Reluctant Splitting (Not Lazy)			
Divided Local Reluctant Splitting (Lazy)	0.076	55.714	128960
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	0.084	81.996	192784
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)			
Eager Splitting and Tail Merging (Not Lazy)			
Eager Splitting and Tail Merging (Lazy)			
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	0.079	62.851	137436
Type Predict for Vectors (with Global Splitting)	0.081	101.259	218952
Vectors Are More Common (with Local Splitting)	0.089	105.881	232424
Without Inlining	0.939	18.481	218952
Without In-Line Caching	0.08	56.02	232424
Without Compile-Time Lookup Caching	0.076	57.674	107380
Without Customization	0.241	33.849	137436
Without Value-Based Type Analysis	0.113	51.217	100620
Without Range Analysis	0.077	59.213	118016
Without Type Prediction	0.198	36.722	128960
Without Deferred Block Creation	0.193	61.817	128960
Without Exposed Block Analysis	0.102	78.208	134760
Without CSE	0.082	45.728	174484
Without CSE of Constants	0.076	54.094	163308
Without CSE of Arithmetic Operations	0.075	55.475	134556
Without CSE of Memory References	0.078	47.242	120152
Without CSE of Memory Cell Type Information	0.079	49.511	131584
Without CSE of Memory Cell Array Bounds Checking	0.076	55.264	128960
Without Eliminating Unneeded Computations	0.077	58.933	118072
Without Delay Slot Filling	0.082	53.788	116248
No Integer Type Tests	0.075	55.215	128960
No Boolean Type Tests	0.075	55.665	128960
No Overflow Checking	0.076	55.384	147852
No Array Bounds Checking	0.076	55.654	124832
No Block Zapping	0.078	51.082	127328
Early Block Zapping	0.078	54.942	128176
Late Block Zapping	0.076	55.714	128960
No Primitive Failure Checking	0.076	55.554	125396
No Debugger-Visible Names	0.076	54.914	129752
No Interrupt Checking at Calls	0.073	54.687	126504
No Interrupt Checking at _Restarts	0.075	54.565	127564
No LRU Compiled Method Reclamation Support	0.073	55.497	128612
Fast	0.074	49.256	120748
Fastest	0.073	46.267	124492

C.29 primMaker

language/configuration	run	compile	space
Optimized C			
Smalltalk-80			
T (normal)			
T (integer only)			
Normal SELF	1.18	147.89	432396
No Splitting (Not Lazy)	1.373	411.817	1023888
No Splitting (Lazy)	1.288	142.932	444412
Local Reluctant Splitting (Not Lazy)	1.394	294.986	762780
Local Reluctant Splitting (Lazy)	1.18	147.89	432396
Global Reluctant Splitting (Not Lazy)			
Global Reluctant Splitting (Lazy)	1.208	156.972	450364
Divided No Splitting (Not Lazy)	1.248	320.122	889716
Divided No Splitting (Lazy)	1.242	143.498	444412
Divided Local Reluctant Splitting (Not Lazy)	1.225	326.075	843484
Divided Local Reluctant Splitting (Lazy)	1.197	148.073	432396
Divided Global Reluctant Splitting (Not Lazy)			
Divided Global Reluctant Splitting (Lazy)	1.213	157.357	450364
Eager Splitting (Not Lazy)			
Eager Splitting (Lazy)			
Eager Splitting and Tail Merging (Not Lazy)			
Eager Splitting and Tail Merging (Lazy)			
Eager Splitting and Requirements Analysis (Not Lazy)			
Eager Splitting and Requirements Analysis (Lazy)			
Type Predict for Vectors (with Local Splitting)	1.22	154.14	439524
Type Predict for Vectors (with Global Splitting)	1.219	217.421	488652
Vectors Are More Common (with Local Splitting)	1.244	171.226	476800
Without Inlining	11.686	79.764	488652
Without In-Line Caching	1.378	148.582	476800
Without Compile-Time Lookup Caching	1.159	153.041	470020
Without Customization	3.612	144.428	439524
Without Value-Based Type Analysis	1.355	163.325	438896
Without Range Analysis	1.216	157.014	494596
Without Type Prediction	3.402	150.098	432396
Without Deferred Block Creation	4.381	177.569	432396
Without Exposed Block Analysis	2.265	208.335	504336
Without CSE	1.198	134.562	603940
Without CSE of Constants	1.199	143.081	515156
Without CSE of Arithmetic Operations	1.198	147.322	446040
Without CSE of Memory References	1.192	140.118	437744
Without CSE of Memory Cell Type Information	1.214	152.256	437336
Without CSE of Memory Cell Array Bounds Checking	1.198	145.322	432308
Without Eliminating Unneeded Computations	1.18	153.59	432812
Without Delay Slot Filling	1.261	142.329	436124
No Integer Type Tests	1.173	147.827	432444
No Boolean Type Tests	1.138	147.342	432396
No Overflow Checking	1.18	147.51	499932
No Array Bounds Checking	1.197	146.793	428012
No Block Zapping	1.197	138.643	426060
Early Block Zapping	1.193	142.917	431108
Late Block Zapping	1.18	147.89	432396
No Primitive Failure Checking	1.213	147.727	409012
No Debugger-Visible Names	1.182	145.728	428872
No Interrupt Checking at Calls	1.153	143.857	423536
No Interrupt Checking at _Restarts	1.199	145.241	430732
No LRU Compiled Method Reclamation Support	1.175	146.835	429084
Fast	1.163	134.767	424096
Fastest	1.125	127.905	426700

C.30 pathCache

language/configuration	run	compile	space
Optimized C			
Smalltalk-80			
T (normal)			
T (integer only)			
Normal SELF	23.587	6.173	17644
No Splitting (Not Lazy)	24.04	18.33	47688
No Splitting (Lazy)	23.958	5.942	18372
Local Reluctant Splitting (Not Lazy)	23.218	15.332	37936
Local Reluctant Splitting (Lazy)	23.587	6.173	17644
Global Reluctant Splitting (Not Lazy)	23.139	47.361	81280
Global Reluctant Splitting (Lazy)	23.233	6.687	19408
Divided No Splitting (Not Lazy)	23.335	15.125	43460
Divided No Splitting (Lazy)	23.636	5.944	18372
Divided Local Reluctant Splitting (Not Lazy)	23.158	15.382	40320
Divided Local Reluctant Splitting (Lazy)	23.687	6.163	17644
Divided Global Reluctant Splitting (Not Lazy)	23.587	23.183	49076
Divided Global Reluctant Splitting (Lazy)	23.021	6.679	19408
Eager Splitting (Not Lazy)	26.783	47.807	101272
Eager Splitting (Lazy)	25.493	29.667	58940
Eager Splitting and Tail Merging (Not Lazy)	23.291	23.749	54044
Eager Splitting and Tail Merging (Lazy)	23.104	15.606	33796
Eager Splitting and Requirements Analysis (Not Lazy)	23.101	23.399	47320
Eager Splitting and Requirements Analysis (Lazy)	24.058	12.482	21116
Type Predict for Vectors (with Local Splitting)	23.449	6.501	18296
Type Predict for Vectors (with Global Splitting)	23.705	7.335	20468
Vectors Are More Common (with Local Splitting)	23.358	6.272	17704
Without Inlining	96.002	7.498	20468
Without In-Line Caching	11.488	6.002	17704
Without Compile-Time Lookup Caching	24.051	6.099	27196
Without Customization	32.775	7.785	18296
Without Value-Based Type Analysis	25.653	6.607	37576
Without Range Analysis	23.133	6.717	20132
Without Type Prediction	35.611	6.079	17644
Without Deferred Block Creation	40.83	7.49	17644
Without Exposed Block Analysis	24.604	9.396	20932
Without CSE	23.364	5.706	23300
Without CSE of Constants	23.855	6.045	24624
Without CSE of Arithmetic Operations	23.699	6.141	18376
Without CSE of Memory References	23.897	5.863	17984
Without CSE of Memory Cell Type Information	23.776	6.174	17740
Without CSE of Memory Cell Array Bounds Checking	23.701	6.109	17644
Without Eliminating Unneeded Computations	24.474	6.796	17884
Without Delay Slot Filling	24.296	5.794	17684
No Integer Type Tests	23.434	5.966	17644
No Boolean Type Tests	23.721	5.999	17644
No Overflow Checking	23.368	6.112	20524
No Array Bounds Checking	23.116	6.124	17228
No Block Zapping	23.227	6.063	17060
Early Block Zapping	23.576	6.244	15952
Late Block Zapping	23.587	6.173	17644
No Primitive Failure Checking	23.676	5.914	15452
No Debugger-Visible Names	23.314	6.106	17448
No Interrupt Checking at Calls	23.126	5.784	17528
No Interrupt Checking at _Restarts	23.618	5.972	17432
No LRU Compiled Method Reclamation Support	23.912	6.058	17516
Fast	23.468	5.662	17476
Fastest			

Bibliography

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[Ado85] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, Reading, MA, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[AJ88] Randy Allen and Steve Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 241-249, Atlanta, GA, June, 1988. Published as *SIGPLAN Notices 23(7)*, July, 1988.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 1-11, San Diego, CA, January, 1988.

[Atk86] Robert G. Atkinson. Hurricane: An Optimizing Compiler for Smalltalk. In *OOPSLA '86 Conference Proceedings*, pp. 151-158, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[BBB+57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The Fortran Automatic Coding System. In *Western Joint Computer Conference*, pp. 188-198, 1957.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *OOPSLA '86 Conference Proceedings*, pp. 78-86, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[BH90] Andrew P. Black and Norman C. Hutchinson. Typechecking Polymorphism in Emerald. Technical report TR 90-34, Department of Computer Science, University of Arizona, December, 1990.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In *SIGPLAN Notices 23*, September, 1988.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 275-284, Portland, OR, June, 1989. Published as *SIGPLAN Notices 24(7)*, July, 1989.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. In *Computer Languages 6*, pp. 47-57, 1981.

[Cha82] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 98-105, Boston, MA, June, 1982. Published as *SIGPLAN Notices 17(6)*, June, 1982.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 146-160, Portland, OR, June, 1989. Published as *SIGPLAN Notices 24(7)*, July, 1989.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 150-164, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in SELF. Published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp. 1-15, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(10)*, October, 1991.

[CH84] Frederick Chow and John Hennessy. Register Allocation by Priority-Based Coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pp. 222-232, Montreal, Canada, June, 1984. Published as *SIGPLAN Notices 19(6)*, June, 1984.

[CH90] Fred C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. In *ACM Transactions on Programming Languages and Systems 12(4)*, pp. 501-536, October, 1990.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 238-252, January, 1977.

[CMR88] Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, Atlanta, GA, June, 1988. Published as *SIGPLAN Notices 23(7)*, July, 1988.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Real-Time Garbage Collector. In *Communications of the ACM 19(10)*, October, 1976.

[Deu83] L. Peter Deutsch. The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 297-302, Salt Lake City, UT, January, 1984.

[Deu88] L. Peter Deutsch. Richards benchmark source code. Personal communication, October, 1988.

[Deu89] L. Peter Deutsch. atAllPut: benchmark suggestion. Personal communication, September, 1989.

[DMSV89] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *OOPSLA '89 Conference Proceedings*, pp. 211-214, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

[GR84] Leonard Gilman and Allen J. Rose. *APL: An Interactive Approach.* Wiley, New York, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[Gra89] Justin Owen Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1989.

[GJ90] Justin O. Graver and Ralph E. Johnson. A Type System for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 136-150, San Francisco, CA, January, 1990.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[Gup90] Rajiv Gupta. A Fresh Look at Optimizing Array Bounds Checking. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 272-282, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990.

[Hal85] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. In *ACM Transactions on Programming Languages and Systems 7(4)*, pp. 501-538, October, 1985.

[Har92] Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[HDB90] Robert Heib, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 66-77, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990.

[Hei90] Richard Louis Heintz, Jr. *Low Level Optimizations for an Object-Oriented Programming Language*. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[Hen82] John L. Hennessy. Symbolic Debugging of Optimized Code. *ACM Transactions of Programming Languages and Systems 4(3)*, July, 1982.

[Hen88] John Hennessy. Stanford benchmark suite source code. Personal communication, June, 1988.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufman, San Mateo, 1990.

[HCC+91] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The SELF Manual, Version 1.1*. Unpublished manual, February, 1991.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, pp. 21-38, Geneva, Switzerland, July, 1991.

[Höl91] Urs Hölzle. In-line cache improvements. Personal communication, August, 1991.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic De-Optimization. To appear in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June, 1992.

[Hut87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* Ph.D. thesis, University of Washington, 1987.

[HRB+87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, October, 1987.

[HC89] Wen-mei W. Hwu and Pohua P. Chang. Inline Function Expansion for Compiling C Programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 246-257, Portland, OR, June, 1989. Published as *SIGPLAN Notices 24(7)*, July, 1989.

[Ing78] Daniel H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 9-16, Tucson, AZ, 1978.

[Ive62] Kenneth Iverson. *A Programming Language.* Wiley, New York, 1962.

[JW85] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, 1985.

[Joh86] Ralph E. Johnson. Type-Checking Smalltalk. In *OOPSLA '86 Conference Proceedings*, pp. 315-321, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Joh87] Ralph Johnson. Workshop on Compiling and Optimizing Object-Oriented Programming Languages. In *Addendum to the OOPSLA '87 Conference Proceedings*, pp. 59-65, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 23(5)*, May, 1988.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Conference Proceedings*, pp. 18-26, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[JLHB88] Eric Jul, Henry Levy, Normal Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. In *ACM Transactions on Computer Systems 6(1)*, pp. 109-133, February, 1988.

[Jul88] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* Ph.D. thesis, University of Washington, December, 1988.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kil88] Michael F. Kilian. Why Trellis/Owl Runs Fast. Unpublished manuscript, March, 1988.

[KKR+] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 219-233, Palo Alto, CA, June, 1986. Published as *SIGPLAN Notices 21(7)*, July, 1986.

[Kra88] David Andrew Kranz. *Orbit: An Optimizing Compiler for Scheme*. Ph.D. thesis, Yale University, 1988.

[KHM89] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 81-90, Portland, OR, June, 1989. Published as *SIGPLAN Notices 24(7)*, July, 1989.

[Kra89] David Kranz. Optimization of object-oriented features in the ORBIT compiler. Personal communication, June, 1989.

[Kra90] David Kranz. Optimization of generic arithmetic in the ORBIT compiler. Personal communication, June, 1990.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA, 1983.

[LH86] James R. Larus and Paul N. Hilfinger. Register Allocation in the SPUR Lisp Compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 255-263, Palo Alto, CA, June, 1986. Published as *SIGPLAN Notices 21(7)*, July, 1986.

[Lea90] Douglas Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, pp. 301-314, San Francisco, CA, April, 1990.

[Lee88] Elgin Lee. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University, 1988.

[LZ74] Barbara H. Liskov and Stephen N. Zilles. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, pp. 50-59, April, 1974. Published as *SIGPLAN Notices 9(4)*, 1974.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. Abstraction Mechanisms in CLU. In *Communications of the ACM 20(8)*, pp. 564-576, August, 1977.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, Berlin, 1981.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.

[McC89] Carl D. McConnell. *The Design of the RTL System*. Master's thesis, University of Illinois at Urbana-Champaign, 1989.

[McF91] Scott McFarling. Procedure Merging with Instruction Caches. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 71-79, Toronto, Ontario, Canada, June, 1991. Published as *SIGPLAN Notices 26(6)*, June, 1991.

[Mey86] Bertrand Meyer. Genericity versus Inheritance. In *OOPSLA '86 Conference Proceedings*, pp. 391-405, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[Mot85] Motorola. *MC68020 32-Bit Microprocessor User's Manual, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[PW86] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. In *Communications of the ACM 29(12)*, pp. 1184-1201, December, 1986.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, New York, 1987.

[PW90] William Pugh and Grant Weddell. Two-Directional Record Layout for Multiple Inheritance. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 85-91, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990.

[RA82] Jonathan Rees and Norman Adams. T: a Dialect of Lisp or, Lambda: the Ultimate Software Tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 114-122, August, 1982.

[RC86] Jonathan Rees and William Clinger, editors. *Revised^3 Report on the Algorithmic Language Scheme*. In *SIGPLAN Notices 21(12)*, December, 1986.

[RAM90] Jonathan Rees, Norman Adams, and James Meehan. *The T Manual, Fifth Edition*, Yale University, October, 1990.

[RG89] Steve Richardson and Mahadevan Ganapathi. Interprocedural Analysis vs. Procedure Integration. In *Information Processing Letters 32(3)*, pp. 137-142, August, 1989.

[Ros88] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. In *OOPSLA '88 Conference Proceedings*, pp. 27-35, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment, Language Reference Manual. Technical report DEC-TR-372, November, 1985.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Sch77] Robert W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. In *Communications of the ACM 20(9)*, pp. 647-654, September, 1977.

[SS88] Peter Sestoft and Harald Søndergaard. A Bibliography on Partial Evaluation. In *SIGPLAN Notices 23(2)*, pp. 19-27, February, 1988.

[Shi88] Olin Shivers. Control Flow Analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 164-174, Atlanta, GA, June, 1988. Published as *SIGPLAN Notices 23(7)*, July, 1988.

[Shi90] Olin Shivers. Data-Flow Analysis and Type Recovery in Scheme. Technical report CMU-CS-90-115, March, 1990. To appear in *Topics in Advanced Language Implementation*, Peter Lee (ed.), MIT Press.

[Sla87] Stephen Slade. *The T Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Ste76] Guy Lewis Steele Jr. LAMBDA: The Ultimate Declarative. AI Memo 379, MIT Artificial Intelligence Laboratory, November, 1976.

[SS76] Guy Lewis Steele Jr. and Gerald Jay Sussman. LAMBDA: The Ultimate Imperative. AI Memo 353, MIT Artificial Intelligence Laboratory, March, 1976.

[Ste84] Guy L. Steele Jr. *Common LISP*. Digital Press, 1984.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Str87] Bjarne Stroustrup. Multiple Inheritance for C++. In *Proceedings of the European Unix Users Group Conference*, pp. 189-207, Helsinki, May, 1987.

[Sun91] Sun Microsystems. *The SPARC Architecture Manual, Version 8*. January, 1991.

[Ung84] David Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 157-167, Pittsburgh, PA, April, 1984, [Ung87] David Michael Ungar. *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, Cambridge, MA, 1987.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs without Classes. Published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[Weg81] Mark N. Wegman. *General and Efficient Methods for Global Code Improvement*. Ph.D. thesis, University of California at Berkeley, 1981.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987.

[Wik87] Åke Wikström. *Functional Programming Using Standard ML*. Prentice-Hall, London, 1987.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *OOPSLA '89 Conference Proceedings*, pp. 23-35, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[WH81] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, Reading, MA, 1981.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. In *Communications of the ACM 14(4)*, pp. 221-227, April, 1971.

[Zel84] Polle T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. Ph.D. dissertation, Computer Science Department, University of California, Berkeley, 1984. Also published as Xerox PARC Technical Report CSL-84-5, May, 1984.

[ZJ91] Lawrence W. Zurawski and Ralph E. Johnson. Debugging Optimized Code with Expected Behavior. Unpublished manuscript, 1991.