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.

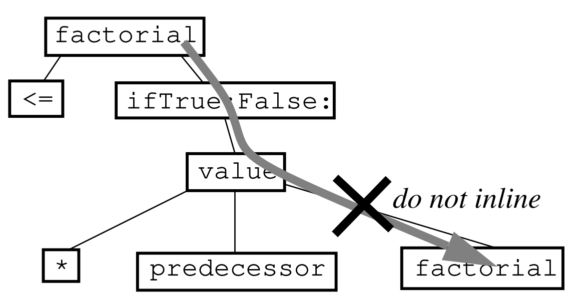
7.1.2.2 Recursion Checks

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.



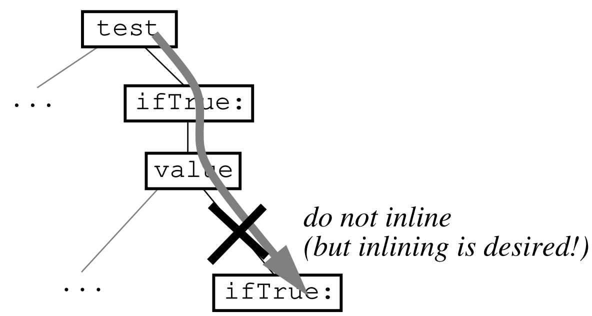
[*] This code may look strange to the SELF novice because all the **`self`** keywords that would be present in the Smalltalk-80 version have been elided using SELF's implicit self message syntax. Thus the receivers of the **`<=`**, **`*`**, and **`predecessor`** messages are all implicitly **`self`**.

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:

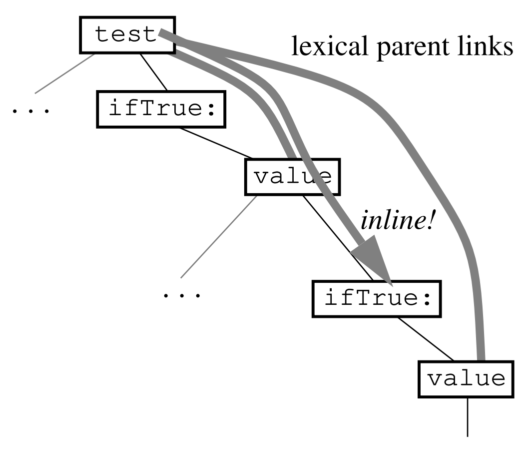
```
test = ( ... ifTrue: [ ... ifTrue: [ ... ] ] )
```

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.

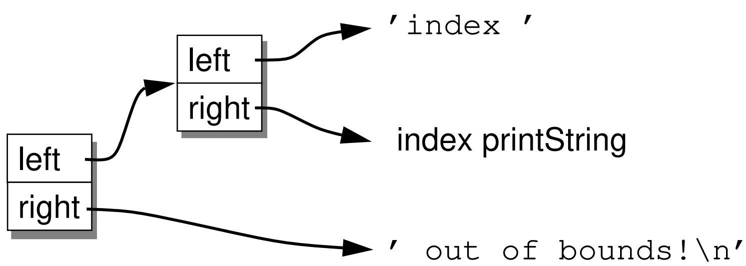


Even this recursion checking is conservative, however. The compiler might be able to inline a recursive call without getting into an infinite loop, if other information available to the compiler makes the processing of the two method invocations somehow different. To illustrate this possibility, consider 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.