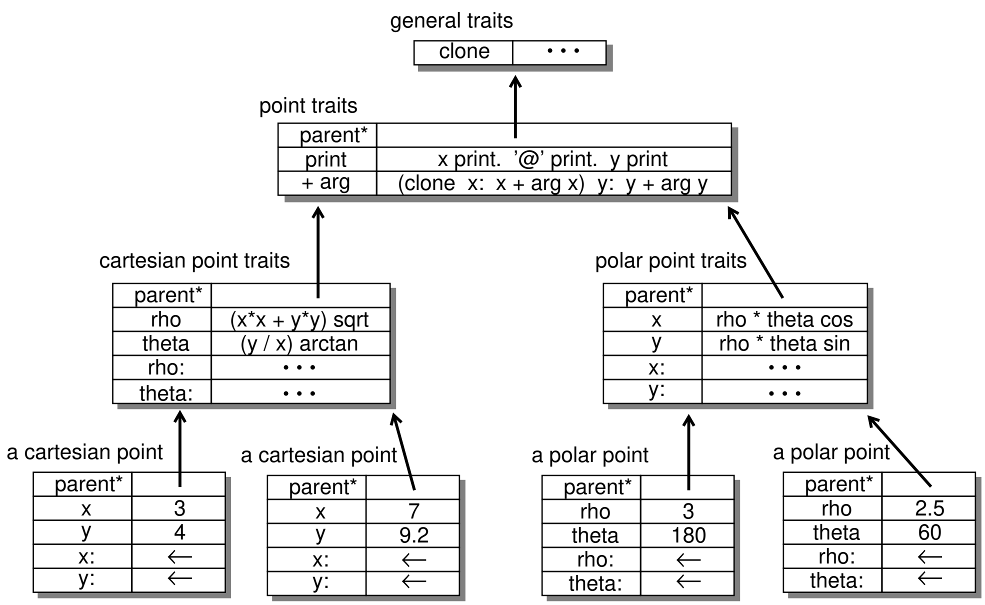
Chapter 8 Customization

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

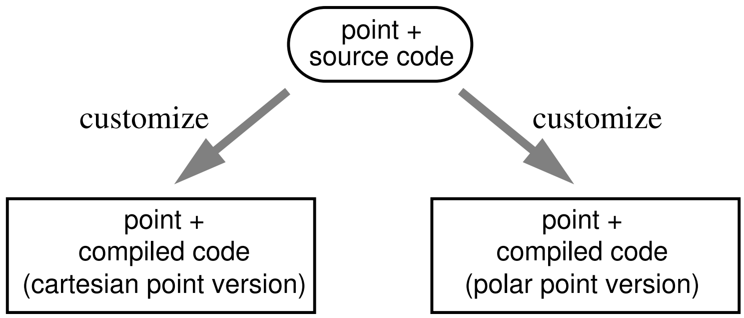
Programmers using object-oriented languages receive much of their expressive power by applying inheritance to organize code, factoring common fragments of code out into shared ancestors. In many cases, the factored code must be parameterized by specific information available to the inheriting objects 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.