*To be published in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991*


# Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs*

CRAIG CHAMBERS (*craig@self.stanford.edu*)
DAVID UNGAR† (*ungar@self.stanford.edu*)
*Computer Systems Laboratory, Stanford University, Stanford, California 94305*

**Abstract.** Object-oriented languages have suffered from poor performance caused by frequent and slow dynamically-bound procedure calls. The best way to speed up a procedure call is to compile it out, but dynamic binding of object-oriented procedure calls without static receiver type information precludes inlining. *Iterative type analysis* and *extended message splitting* are new compilation techniques that extract much of the necessary type information and make it possible to hoist run-time type tests out of loops.

Our system compiles code on-the-fly that is customized to the actual data types used by a running program. The compiler constructs a control flow graph annotated with type information by simultaneously performing type analysis and inlining. Extended message splitting preserves type information that would otherwise be lost by a control-flow merge by duplicating all the code between the merge and the place that uses the information. Iterative type analysis computes the types of variables used in a loop by repeatedly recompiling the loop until the computed types reach a fix-point. Together these two techniques enable our SELF compiler to split off a copy of an entire loop, optimized for the common-case types.

By the time our SELF compiler generates code for the graph, it has eliminated many dynamically-dispatched procedure calls and type tests. The resulting machine code is twice as fast as that generated by the previous SELF compiler, four times faster than ParcPlace Systems Smalltalk-80, the fastest commercially available dynamically-typed object-oriented language implementation, and nearly half the speed of optimized C. Iterative type analysis and extended message splitting have cut the performance penalty for dynamically-typed object-oriented languages in half.

---

*This work has been generously supported by National Science Foundation Presidential Young Investigator Grant #CCR-8657631, and by Sun Microsystems, IBM, Apple Computer, Tandem Computers, NCR, Texas Instruments, the Powell Foundation, and DEC.

†Author's present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

This paper was originally published in the *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation* (*SIGPLAN Notices*, 25, 6 (1990) 150-162).

## 1 Introduction

Dynamically-typed object-oriented languages have historically been much slower in run-time performance than traditional languages like C and Fortran. Our measurements of several Smalltalk systems on personal computers and workstations [21] indicate that their performance is between 5% and 20% of the performance of optimized C programs. This disparity in performance is caused largely by the relatively slow speed and high frequency of message passing and the lack of static type information to reduce either of these costs. This paper describes new techniques for extracting and preserving static type information in dynamically-typed object-oriented programs.

This work continues our earlier work on the SELF programming language [12, 22]. SELF is a new dynamically-typed object-oriented language in the spirit of Smalltalk-80[1] [5], but is novel in its use of prototypes instead of classes and its use of messages instead of variables to access state. These features make SELF programs even harder to run efficiently than other dynamically-typed object-oriented languages, since SELF programs send many more messages than equivalent Smalltalk programs.

As part of our earlier work we built an optimizing compiler for SELF that pioneered the use of *customization*, *type prediction*, and *message splitting* [2, 3]. These techniques provided the compiler with much more static type information than was previously available, enabling it to aggressively inline away many of the costly message sends without sacrificing source-code compatibility. This SELF compiler achieved a performance of between 20% and 25% of optimized C on the Stanford integer benchmarks [9], twice that of the fastest Smalltalk implementation on the same machine.

While this performance is a clear improvement over comparable systems, it is still not competitive with traditional languages. To narrow the gap even further, we have developed and implemented *iterative type analysis* and *extended message splitting* in our new SELF compiler. These techniques provide the compiler with more accurate static type information and enable it to preserve this type information more effectively. These techniques are especially important when optimizing loops and often lead to more than one version of a loop being compiled, each version optimized for different run-time types. Using these techniques, our new SELF compiler produces code that runs almost half as fast as optimized C programs, without sacrificing dynamic typing, overflow and array bounds checking, user-defined control structures, automatic garbage collection, and complete source-level debugging of optimized code.

Traditional compilers can be divided into a front-end (parser) and a back-end (optimizer and code generator). To generate good code for a dynamically-typed object-oriented language, we have inserted a new phase between the front-end and

[1]Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

the back-end. This phase performs type analysis, method inlining, and message splitting to construct the control flow graph from abstract syntax trees of the source code. A more traditional back-end performs data flow analysis, global register allocation, and code generation from this control flow graph.

This paper describes the new intermediate phase of our compiler. The next section briefly describes previously published techniques used in our SELF compiler. Section 3 presents our type system, and describes how type analysis works for straight-line code. Section 4 extends the type analysis to handle merges in the control flow graph, and describes extended message splitting. Section 5 completes type analysis and message splitting by describing iterative type analysis for loops, and presents a simple example of compiling multiple versions of a loop. Section 6 compares the performance of our new SELF system against optimized C, the original SELF compiler, and versions of the new SELF compiler with selected optimizations disabled. Section 7 discusses related work.

## 2 Background

The new techniques presented in this paper build upon those introduced in the previous SELF compiler, including *customization*, *type prediction*, *message splitting* (called *local message splitting* in this paper), *message inlining*, and *primitive inlining* [2, 3].

- *Customized compilation.* Existing compilers for Smalltalk-80 (as well as most other object-oriented languages) compile a single machine code method for a given source code method. Since many classes may inherit the same method, the Smalltalk-80 compiler cannot know the exact class of the receiver. Our SELF compiler, on the other hand, compiles a different machine code method *for each type of receiver* that runs a given source method. The advantage of this approach is that our SELF compiler can know the type of the receiver of the message at compile-time.
- *Type prediction.* Sometimes the name of the message is sufficient to predict the type of its receiver. For example, several studies [20] have shown that the receiver of a **+** message is nine times more likely to be a small integer than any other type. Our compiler inserts type tests in these cases so that subsequent code may exploit the type information in the common case.
- *Message inlining.* Once the type of a receiver is known, the compiler can optimize away the message lookup by performing it at compile-time, and then inline the code invoked by the message.
- *Primitive inlining.* SELF also includes primitive operations such as integer addition. Rather than compiling a call to an external routine, the compiler can directly compile many primitives in line. If the primitives include type tests, the compiler's type information may be used to eliminate them.

**Types in the SELF Compiler**

| type name | set description | static information | source |
|---|---|---|---|
| value | singleton set | compile-time constant | literals, constant slots, true and false type tests |
| integer subrange | set of sequential integer values | integer ranges | arithmetic and comparison primitives |
| class | set of all values w/ same class | format and inheritance | self, results of primitives, integer type tests |
| unknown | set of all values | none | data slots, message results, up-level assignments |
| union | set union of types | union | results of primitive operations |
| difference | set difference of types | difference | failed type tests |

## 3 Simple Type Analysis

To compute the static type information necessary for message inlining, the compiler builds a mapping from variable names to types at each point in the program (i.e. between every node in the control flow graph). This mapping is computed from the nodes in the control flow graph, such as assignment nodes, run-time type test nodes, and message send nodes. The type of a variable describes all the information the compiler knows about the current value of the variable, and as such differs from the standard notion of data type in a traditional statically-typed language.

### 3.1 The Type System

A type specifies a non-empty *set of values*. A variable of a particular type is guaranteed to contain only values in the type's set of values at run-time. A type that specifies a single value (called a *value type*) acts as a compile-time constant. The type that specifies all possible values provides no information to the compiler and is called the *unknown type*.

A type that specifies all values that are instances of some class[2] (called a *class type*) provides the compiler with both format and inheritance information for variables of the type, much like a traditional data type. Messages sent to a variable of class type can be looked-up at compile-time and inlined. A type that specifies a subrange of the values in the integer class type is called an *integer subrange type*. The compiler treats integer value types and the integer class type as extreme forms of integer subrange types.

A type may also specify the *set union* of several types or the *set difference* of two types. The chart above summarizes the kinds of types in our type system, the information they provide to the compiler, and how they are created.

[2]Since SELF has no classes, our implementation introduces user-transparent maps to provide information and space efficiency similar to that of classes. Thus in our system the class type becomes the set of all values that share the same map.

### 3.2 Type Analysis Rules

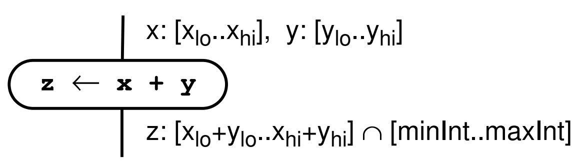
At the start of the method, the type mapping contains bindings for the receiver and each argument. Since our compiler generates *customized* versions of a source method (see section 2), the class of the receiver is known at compile-time for the version being compiled. Therefore, the receiver is initially bound to the corresponding class type. Our system doesn't currently customize based on the types of arguments, and so the arguments are initially bound to the unknown type.

#### *3.2.1 Simple Node Analysis*

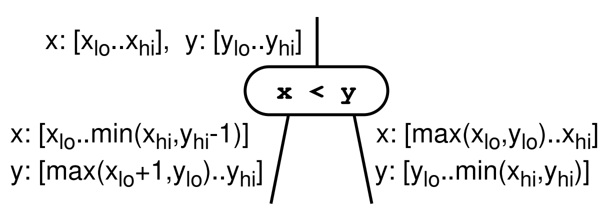
A declaration of a local variable adds a new binding to the type mapping. Since local variables in SELF are always initialized to compile-time constants, each binding will be to some value type. For example, since most variables are (implicitly) initialized to **`nil`**, their types at the start of their scopes would be the nil value type.

Each node in the control flow graph may alter the type bindings as type information propagates across the node. A local assignment node simply changes the binding of the assigned local variable to the type of the value being assigned. A memory load node (e.g. implementing an instance variable access) binds its result temporary name to the unknown type (since the compiler doesn't know the types of instance variables).

Our compiler computes the type of the result of integer arithmetic nodes using *integer subrange analysis*. For example, the following rule is used to determine the result type for the integer addition node:[3]



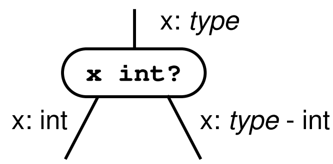

Integer compare-and-branch nodes also use integer subrange analysis. However, instead of adding a binding for a result, compare-and-branch nodes *alter* the type bindings of their arguments on each outgoing branch. For example, the following rule is used to alter the argument type bindings for the compare-less-than-and-branch node:[4]




---

[3]This node is not the integer addition primitive, but just an add instruction. Type checking and overflow checking are performed by other nodes surrounding the simple add instruction node.

Run-time type test nodes are similar to compare-and-branch nodes. Along the success branch of the test the tested variable is rebound to the type of the test; along the failure branch the variable is rebound to the set difference of the incoming type and the tested type. For example, the following rule is used to alter the argument type binding for the integer type test:




### 3.2.2 *Message Send Node Type Analysis, Type Prediction, and Inlining*

To propagate types across a message send node, the compiler first attempts to *inline* the message. The compiler looks up the type bound to the receiver of the message. If the type is a class type (or a subset of a class type, such as a value type or an integer subrange type), the compiler performs message lookup at compile-time, and replaces the message send node with either a memory load node (for a data slot access), a memory store node (for a data slot assignment), a compile-time constant node (for a constant slot access), or the body of a method (for a method slot invocation). If a method is inlined, new variables for its formals and locals are created and added to the type mapping. The type of the result of a message send node that is not inlined is the unknown type.

If the type of the receiver of the message is unknown (or more general than a single class type), the compiler tries to *predict* the type of the receiver from the name of the message (such as predicting that the receiver of a **+** message is likely to be an integer). If it can successfully predict the type of the receiver, the compiler inserts a run-time type test before the message to verify the guess, and uses local message splitting to compile two versions of the predicted message.

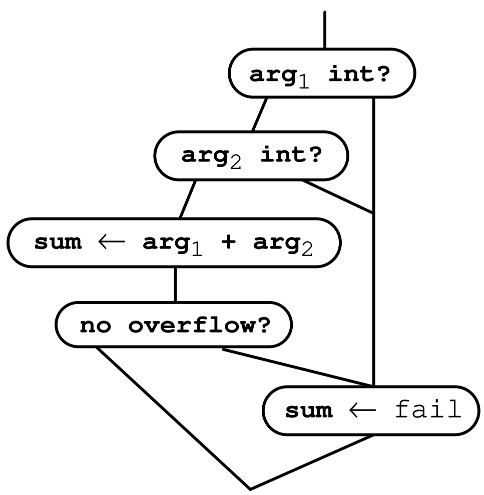
### 3.2.3 *Primitive Operation Node Type Analysis, Range Analysis, and Inlining*

In addition to sending messages, SELF programs may invoke primitive operations. These primitives include integer arithmetic, array accesses, object cloning, and basic graphics primitives. All primitive operations in SELF are *robust*: the types of arguments are checked at the beginning of the primitive and exceptional conditions such as overflow, divide-by-zero, and array-access-out-of-bounds are checked. A call to a primitive can optionally pass a user-defined failure block to invoke in case one of these exceptional conditions occurs; the result of the failure block is used as the result of the primitive operation. If the SELF programmer doesn't provide an explicit failure block, a default failure block is passed that simply calls a standard error routine when invoked.

[4]In all control flow graph diagrams, conditional branch nodes have the `true` outgoing branch on the left, and the `false` outgoing branch on the right.

To propagate types across a primitive operation node, the compiler first attempts to *constant-fold* the primitive. If the primitive has no side-effects and the arguments are value types (i.e. compile-time constants), then the compiler executes the primitive at compile-time and replaces the primitive node with the compile-time constant result. Sometimes the compiler can constant-fold a primitive even if the arguments aren't compile-time constants. For example, if the arguments to an integer comparison primitive are integer subranges that don't overlap, then the compiler can execute the comparison primitive at compile-time based solely on subrange information.

If the compiler can't constant-fold the primitive, and the primitive is small and commonly used (such as integer arithmetic and array accesses), then the compiler *inlines* the primitive, replacing the call to the primitive with lower-level nodes that implement the primitive. For example, the following set of nodes implement the integer addition primitive:



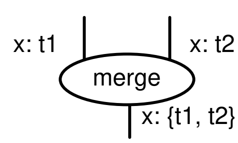

By analyzing the nodes that make up the primitive, the compiler is frequently able to optimize away the initial type tests and even the overflow check. For example, if the arguments to an integer arithmetic primitive are integer subranges that cannot cause an overflow, then the compiler can constant-fold away the initial type tests, the overflow check, and the failure block, leaving a single add instruction node. As it eliminates the type and overflow tests, the compiler comes closer and closer to its goal of eliminating the performance disadvantage of robust primitives. If all the tests can be eliminated, the failure block can be eliminated, which saves space, but more importantly, eliminates subsequent type tests of the result of the primitive.

The type of the result of an inlined primitive can be computed by propagating types across the nodes implementing the primitive. Even if the primitive isn't inlined, the compiler binds the result of the primitive to the result type stored in a table of primitive result types. The original SELF compiler could also constant-fold and inline primitive calls, except it did no range analysis and so couldn't constant-fold a comparison primitive based solely on integer subrange information or eliminate overflow checks and array bounds checks.

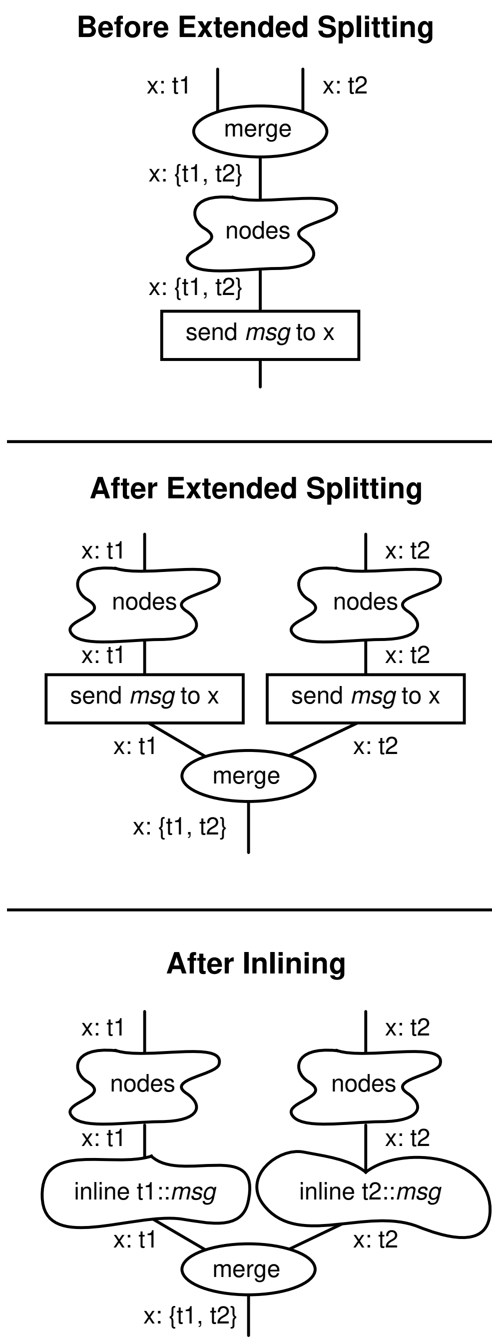
## 4 Extended Message Splitting

Both the original and the new SELF compilers use *message splitting* to take advantage of type information that otherwise would be lost to merges in the control flow graph (see [2, 3]). The original SELF compiler was only able to split messages that immediately followed a merge point; we call this *local message splitting*. Our new SELF compiler performs enough type analysis to detect all splitting opportunities, no matter how much code separates the message send from the merge point; we call this *extended message splitting*.

To propagate type information across a merge node, the compiler constructs the type mapping for the outgoing branch of the merge node from the type mappings for the incoming branches. For each variable bound to a type in all incoming branches, the compiler merges the incoming types. If all incoming types are the same, then the outgoing type is the same as the incoming type. If the types are different, then the compiler constructs a new *merge type* containing the incoming types. A merge type is similar to a union type, except that the compiler knows that the dilution of type information was caused by a merge in the control flow graph. In addition, a merge type records the identities of its constituent types, rather than recording the result of the set union of the merged types. For example, the integer class type merged with the unknown type forms a merge type that contains both types as distinct elements, rather than reducing to just the unknown type as a set union would produce (recall that the unknown type specifies all possible values, and so contains the integer class type).




The compiler takes advantage of merge types when propagating type information across message send nodes. If the type of the receiver of the message is a merge type, containing types of different classes, the compiler may elect to *split* the message and all the intervening nodes back through the control flow graph up to the merge point that diluted the type information. This splitting creates two copies of the nodes in the graph from the send node back to the merge point; the receiver of each copy of the split message send node now has a more specific type, allowing

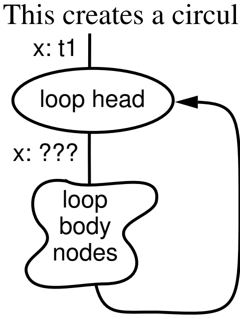

Before Extended Splitting
x: t1
x: t2
merge
x: {t1, t2}
nodes
x: {t1, t2}
send *msg* to x
After Extended Splitting
x: t1
x: t2
nodes
nodes
x: t1
x: t2
send *msg* to x
send *msg* to x
x: t1
x: t2
merge
x: {t1, t2}
After Inlining
x: t1
x: t2
nodes
nodes
x: t1
x: t2
inline t1::*msg*
inline t2::*msg*
x: t1
x: t2
merge
x: {t1, t2}

the compiler to do normal message inlining for each copy of the message separately; without splitting, the original message send couldn't be inlined.

Of course, uncontrolled extended message splitting could lead to a large increase in the size of the graph and thus in compiled code size and compile time. To limit the increase in code size, our compiler only performs extended message splitting when the number of copied nodes is below a fixed threshold, and only copies nodes along the "common case" branches of the control flow graph (i.e. along branches that aren't downstream of any failed primitives or type tests).

## 5 Type Analysis For Loops

Performing type analysis for loops presents a problem. The loop head node is a kind of merge node, connecting the end of the loop back to the beginning. Thus the type bindings at the beginning of the loop body depend not only on the type bindings before the loop, but also on the bindings at the end of the loop, which depend on the bindings at the beginning. This creates a circular dependency.




One solution to this circularity would be to construct a type binding table for the loop head that is guaranteed to be compatible with whatever bindings are computed for the end of the loop. This can be done by rebinding all locals assigned within the loop to the most general possible type: the unknown type. We call this strategy *pessimistic type analysis*. However, it effectively disables the new SELF compiler's type analysis system, including range analysis, at precisely the points that are most important to performance: the inner loops of the program. Without more accurate type information, the compiler is forced to do type prediction and insert run-time type tests to check for expected types of local variables. Since the original SELF compiler performed no type analysis, local variables were always considered to be of unknown type, and so the original SELF compiler could be thought of as using pessimistic type analysis for loops.

Another solution to the circularity would be to use traditional iterative data flow techniques [1] to determine the type bindings for the loop before doing any inlining within the loop. However, most locals changed within the loop would be assigned the results of messages, and since these message aren't inlined yet, their result types are unknown, and so most locals would end up being bound to the unknown type by the end of the loop body. The net effect of standard iterative data flow analysis for type information is the same as for the pessimistic type analysis: assigned locals end up bound to the unknown type.

## 5.1 Iterative Type Analysis

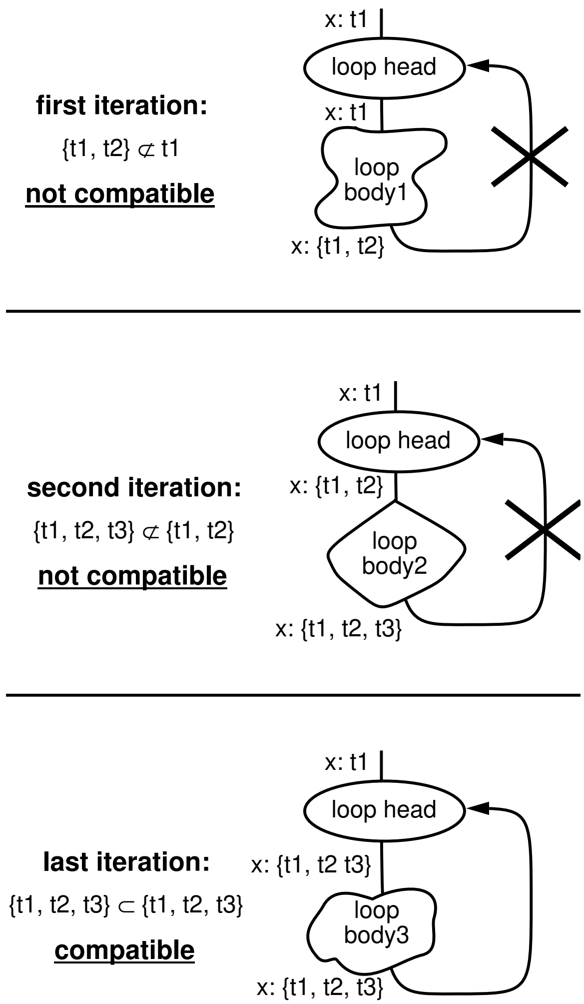
The solution adopted in the new SELF compiler is called *iterative type analysis*. Our compiler first uses the type bindings at the head of the loop to compile the body of the loop, with as much inlining and constant-folding as possible based on those types. It then compares the type bindings for the end of the loop with the head of the loop. If they are the same, then the compiler has successfully compiled the loop and can go on to compile other parts of the program. If some types are different, then the compiler forms the appropriate merge types for those locals whose types are different, and recompiles the body of the loop with the more general type bindings. This process iterates until the fixed point is reached, where the type bindings of the head of the loop are compatible with the type bindings at the end of the loop.

Iterative type analysis computes type bindings over a changing control flow graph, building the control flow graph as part of the computation of the type binding information. Standard data flow techniques, on the other hand, operate over a fixed control flow graph. Iterative type analysis is important for dynamically-typed object-oriented languages, where transformations of the control flow graph (such as inlining) are crucial to compute accurate type information.

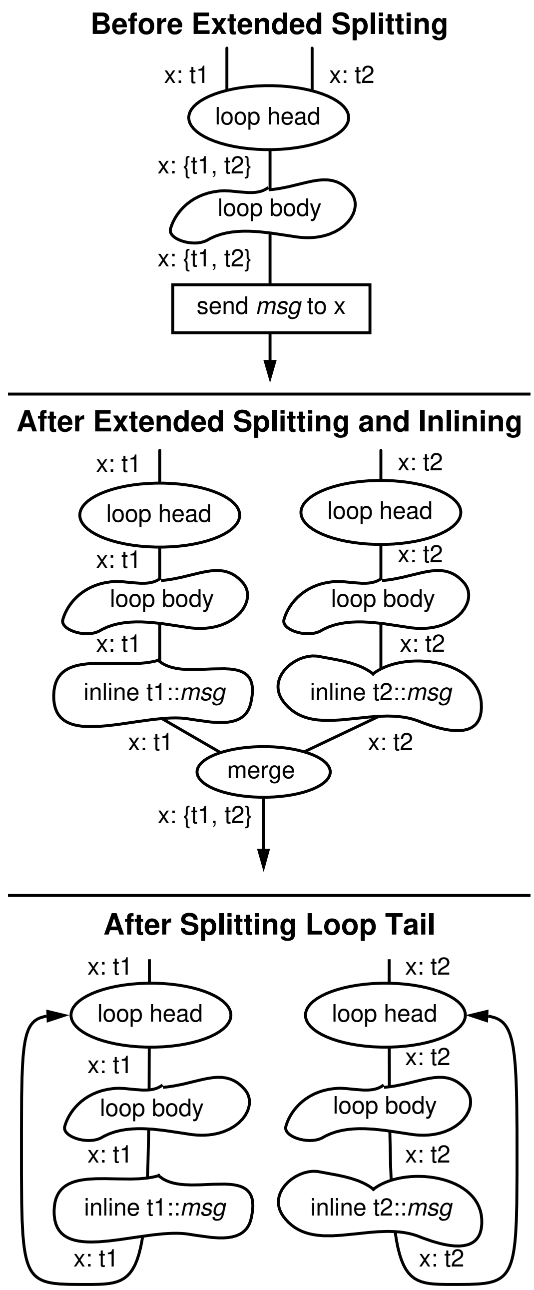
To reach the fixed point in the analysis quickly, loop head merge nodes compute the type binding table in a slightly different way than normal merge nodes. If the loop head and the loop tail produce different value or subrange types within the same class type for a particular local, the loop head merge node generalizes the individual values to the enclosing class type (instead of forming a normal merge type). For example, if the initial type of a local is the `0` value type, and the ending type is the `1` value type (as it would be for a simple loop counter initialized to zero), the loop head node rebinds the local to the integer class type rather than the merge of the two value types. Then the type automatically handles all future integer values of the counter in one iteration. This sacrifices some precision in type analysis, but saves a great deal of compile time and does not seem to hurt the quality of the generated code.

## 5.2 Iterative Type Analysis and Extended Message Splitting

The combination of extended message splitting and iterative type analysis makes it possible to compile multiple versions of loops. For example, consider a loop head that merges two different types together, and so creates a merge type. The compiler should be free to split the merged types apart to inline a message send inside the body of the loop, and so may actually split the loop head node itself into two copies, each with different type information. Each loop head thus starts its own version of the loop, compiled for different type bindings.




When the compiler finally reaches the loop tail node, after compiling the body of the loop, there may be multiple loop heads to choose from. The compiler first tries to find a loop head that is *compatible* with the loop tail, and if it finds one connects the loop tail to the compatible loop head. If it doesn't find a compatible loop head, it tries to split the loop tail node itself to create a copy of the loop tail that is compatible with one of the loop heads. If the type of a local at the loop tail is a merge type, and one of the loop heads contains a binding that is a subset of the merge type, then the loop tail is split to generate a loop tail that only contains the matching subset, and this copy is connected to its matching loop head. The compiler then attempts to match and/or split the other loop tail.


Before Extended Splitting
x: t1
x: t2
loop head
x: {t1, t2}
loop body
x: {t1, t2}
send msg to x
After Extended Splitting and Inlining
x: t1
x: t2
loop head
loop head
x: t1
x: t2
loop body
loop body
x: t1
x: t2
inline t1::msg
inline t2::msg
x: t1
x: t2
merge
x: {t1, t2}
After Splitting Loop Tail
x: t1
x: t2
loop head
loop head
x: t1
x: t2
loop body
loop body
x: t1
x: t2
inline t1::msg
inline t2::msg
x: t1
x: t2

Only if a loop tail doesn't match any of the available loop heads does the compiler give up, throw away the existing versions of the loop, and recompile it with more general type bindings. To compute the type bindings for the head of the new loop, the compiler forms merge types from the bindings for the old loop heads and the remaining unattached loop tail.

*Compatibility* needs to be defined carefully to avoid losing type information. A loop tail is compatible with (matches) a loop head if for each type binding the type at the loop head contains the type at the loop tail *and* the type at the loop head does not sacrifice class type information present in the loop tail. This means that the unknown type at the loop head is *not* compatible with a class type at the loop tail. Instead, the type analysis will iterate, forming a merge type of the unknown type and the class type at the loop head. This has the advantage that the body of the loop may split off the class type branch from the unknown type branch, and generate better code along the class type branch.

### 5.3 An Example

Consider a very simple SELF function that sums all the integers from **1** up to its argument **n**:

```
triangleNumber: n = (
| sum <- 0 | "declare and init sum"
1 upTo: n Do: [ | :i |
"i is loop index"
sum: sum + i. "increment sum"
].
sum ). "return sum"
```

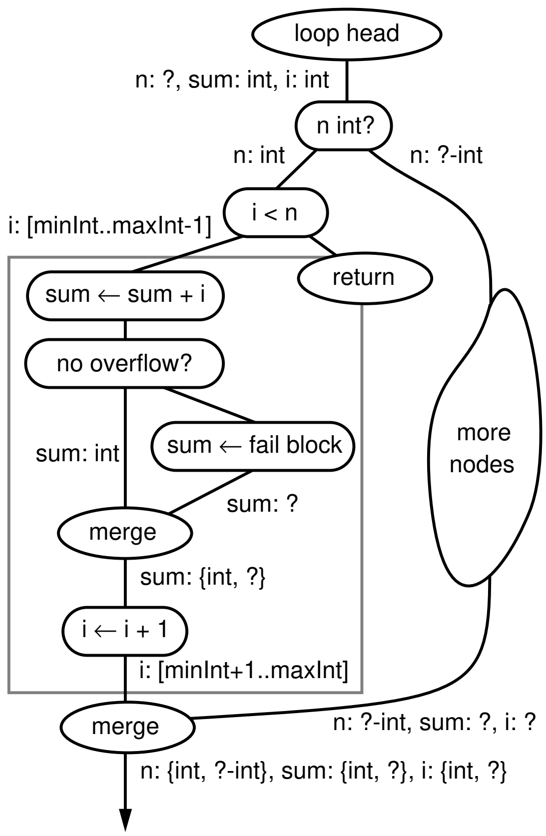
This function uses the user-defined control structure **upTo:Do:** to iterate through the numbers from **1** to **n-1**. After inlining the control structure down to primitive operations, the compiler produces the following:

```
triangleNumber: n = (
| sum <- 0. i <- 1. |
loop:
if i < n then
sum: sum + i.
i: i + 1.
goto loop
sum ).
```

The compiler uses iterative type analysis to compile the body of the loop. The first time through, **sum** is initially bound to the **0** value type and **i** is initially bound to the **1** value type. Both **+** messages will get inlined down to integer addition primitives and constant-folded. At the end of the loop, **sum** is bound to the **1** value type and **i** is bound to the **2** value type. These types are incompatible with the constants assumed at the head of the loop, and so type analysis iterates.

The second iteration starts by generalizing the types of both **`sum`** and **`i`** to the integer class type (remember that loop head merge nodes intentionally generalize their merge types to speed the analysis). After completing this iteration (and assuming that the result type of a failed primitive is the unknown type), the compiler generates the control flow graph pictured below.[5] The type tests for **`sum`** and **`i`** are optimized away using the type information computed for the loop, and the overflow check for the increment of **`i`** is optimized away using integer subrange analysis.

**Second Iteration of TriangleNumber Example**




The portion of this version of the loop in the gray box is the best one could expect a compiler to achieve.[6] Unfortunately, the loop tail still doesn't match the loop head (e.g. sum: {int, ?} $\not\subset$ int), and so type analysis must iterate. Without extended splitting,

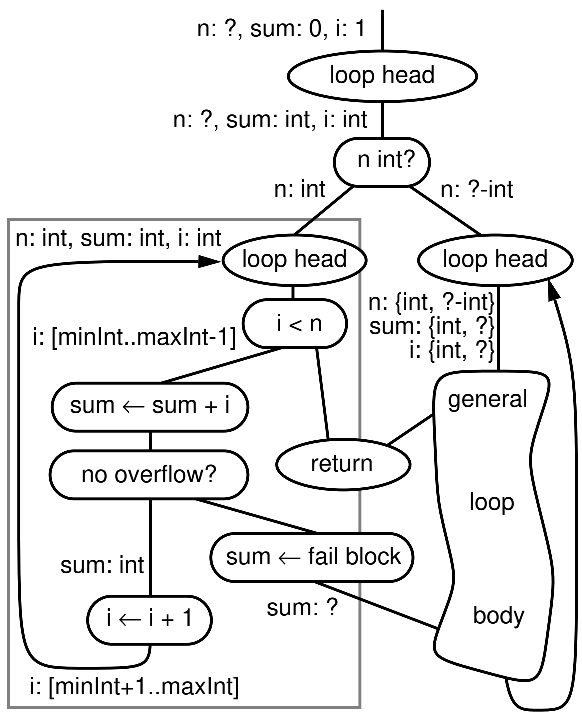
[5] ? denotes the unknown type.

[6] The compiler can't eliminate the remaining overflow check, since it is possible to pass in an `n` argument that would cause `sum` to overflow (e.g. the largest possible integer).

the compiler would have to compile a single version of the loop that worked for all cases. This more general version would need five run-time type tests before the **<** and **+** operations to test for integer arguments. With extended splitting, however, the compiler is able to eliminate *all* run-time type tests from a common-case version of the loop, generating exactly what's in the gray box; another version will be generated to handle overflows and a non-integer `n`.

To restart the type analysis, the compiler builds a new loop initialized with the types resulting from the previous iteration. Analysis then proceeds similarly as before, except that when analyzing the **<** and **+** messages, the compiler *splits* off the integer receiver and argument cases from the non-integer cases, splitting the loop head in the process.[7] When the loop tail is reached, the compiler splits it into two tails, and connects each to its corresponding loop head. The final control flow graph is pictured below:

**Final Results of TriangleNumber Example**




The combination of extended splitting and iterative type analysis has allowed the compiler to optimize all type tests from the common case. A compiler for a statically-typed, non-object-oriented language could do no better.

[7]The actual workings of the compiler, and the final control flow graph, are a bit more complex than those presented here. We have chosen to simplify the exposition of the ideas by glossing over some of these messy details.

### 5.4 Discussion

Combining extended message splitting with iterative type analysis has several beneficial effects. Our compiler can generate multiple versions of loops, each version assuming different type bindings and therefore optimized for different cases at run-time. This is especially important to isolate the negative effects of primitive failure from the normal case of primitive success. For example, a loop that performs arithmetic on locals in the body might get two versions compiled: one that knows all the locals are integers, and a second that handles locals of any type. The first version will branch to the second version only if a primitive fails; if no primitives fail (the common case) control will remain in the fast integer version of the loop. Robustness of integer arithmetic primitives has been implemented at the cost of only an overflow check; no extra type tests are needed if the failure never happens.

Extended message splitting also may "hoist" type tests out of a loop, as it did with the **`n`** integer type test in the above **`triangleNumber:`** example. If the initial types of some variables are unknown (such as for method arguments), and the body of a loop does arithmetic on the variables, our compiler will compile a version of the loop for the unknown types, and embed type tests to check for integer values at run-time. If the values turn out to be integers, the second iteration of the loop will branch to another version that was compiled assuming integer types for the locals. Control will remain in the second version of the loop as long as the locals remain integers (e.g. until an overflow occurs). The first version of the loop contains the type tests, while the second version contains none. If the normal case is to have integer values, then the type tests effectively have been hoisted out of the integer version into the unknown version, which is executed only on the first loop iteration.

Extended message splitting and iterative type analysis have been carefully designed to automatically compile multiple versions of loops. No additional implementation techniques or special algorithms are needed. No special treatment of integers or loop control variables is needed, nor is any special work performed to hoist type tests out of loops. The compiler just uses type prediction and message splitting to create and preserve the type information needed to inline messages and avoid type tests and sometimes ends up creating multiple versions of a loop.

Of course, extended message splitting exacts a price in compile time and compiled code space. However, compiling an additional specialized version of most loops is probably not too costly. This is because the specialized version tends to be much smaller than the more general version of the loop that is littered with type tests, message sends, and failure blocks. Unfortunately, our current implementation sometimes compiles more than just two versions of a loop; we plan to work on minimizing the number of extra versions of loops that get compiled.

## 6 Performance Measurements

We measured the performance of the compiled code, the compiled code size, and compile time. All measurements were taken on a Sun-4/260 SPARC-based workstation. Our measurements are summarized for four sets of benchmarks:

- **stanford** is the set of eight integer benchmarks from the Stanford benchmark suite. These benchmarks typically measure the speed of accessing and iterating through fixed-length arrays of integers.
- **stanford-oo** consists of the same eight benchmarks rewritten in an object-oriented style. The changes are chiefly to redirect the target of messages from the benchmark object to the data structures manipulated by the benchmark (such as the array being sorted); none of the underlying algorithms were changed, nor were any source-level optimizations performed as part of the rewrite. The **puzzle** benchmark was not rewritten, but is included in this group anyway in the interest of fairness.
- **small** is a group of "micro-benchmarks" used as an initial test suite when implementing the new techniques.
- **richards** is a larger, operating system simulation benchmark, written in about 400 lines of SELF source code.

The benchmarks were run with five compilers:

- **optimized C** is the C compiler supplied with SunOS 4.0 and invoked with the **–O2** flag. For **richards**, which is written in C++, the C version includes the effect of the AT&T cfront 1.2.1 preprocessor.
- **ST-80** refers to the ParcPlace Systems Version 2.4 Smalltalk-80 implementation. This system uses dynamic compilation [4], and is tied with Version 2.5 for the distinction of being the fastest commercially available Smalltalk system.
- **old SELF-89** refers to the measurements taken for the old SELF compiler in early 1989 and published in [2]. This was a well-tuned SELF system with a simpler compiler based on expression trees, customization, and local splitting.
- **old SELF-90** is our current, production SELF system which uses the old SELF compiler. This system includes more elaborate semantics for message lookup and blocks, and is not as highly tuned as it was a year ago. For these reasons, the performance has worsened from last year's numbers. However, comparing its performance to that of the new SELF compiler allows us to isolate the effects of the improvements in the new compiler.
- **new SELF** is our new SELF compiler as described in this paper, but without compiling multiple versions of loops. At the time of this writing, the part of the new compiler that recomputes the type information within a loop after splitting a loop head is broken. The results we have observed in the past for compiling multiple versions of loops leads us to expect even better performance when this part of the compiler is repaired.

**Speed of Compiled Code (as a percentage of optimized C)**
**median ( min — max )**

| | small | stanford | stanford-oo | richards |
|---|---|---|---|---|
| ST-80 | 10% (5-10) | 9% (5-53) | 9% (5-80) | 9% |
| old SELF-89 | | 19% (10-48) | 28% (13-56) | 26% |
| old SELF-90 | 11% (7-12) | 14% (9-41) | 19% (9-69) | 17% |
| new SELF | 24% (21-53) | 25% (19-47) | 42% (19-91) | 21% |

**Compile Time and Code Size**
**median / 75%-ile / max**

| | small | stanford+stanford-oo | puzzle | richards |
|---|---|---|---|---|
| **compile time (in seconds of CPU time)** | | | | |
| optimized C | | 3.0 / 3.4 / 3.9 | 9.1 | 13.4 |
| old SELF-90 | 0.3 / 0.3 / 0.4 | 0.7 / 0.8 / 1.1 | 6.9 | 2.1 |
| new SELF | 5.2 / 5.7 / 6.4 | 21.1 / 31.9 / 123.3 | 362.3 | 35.6 |
| **compiled code size (in kilobytes)** | | | | |
| optimized C | | 2.7 / 2.9 / 3.3 | 5.0 | 6.1 |
| old SELF-90 | 2.6 / 2.9 / 5.3 | 11.6 / 13.2 / 18.5 | 81.3 | 34.3 |
| new SELF | 1.5 / 1.6 / 1.8 | 7.7 / 10.2 / 16.2 | 41.3 | 25.5 |

The rest of this section summarizes the results. Raw data for individual benchmarks are given in the appendices.[8]

## 6.1 Speed of Compiled Code

These results in the above table show that the new SELF compiler is around 40% the speed of optimized C for the **stanford-oo** benchmarks. This performance is four times faster than Smalltalk-80 and more than twice as fast as the current **SELF-90** version of the original compiler. Some of this improvement over the original SELF compiler results from better register allocation and delay slot filling. Much of the rest can be credited to better type analysis and especially the inclusion of range analysis.

[8]Since this paper was originally published, the compiler's implementation has been refined and made reliable. Execution performance is faster than reported in this paper (over 60% the speed of optimized C for the stanford and stanford-oo benchmarks), and compilation speed is between one and two orders of magnitude faster (about the same speed as the optimized C compiler).

The **richards** benchmark is worthy of further mention. Its performance is not as good as some of the other benchmarks, and we have traced this problem to a single bottleneck: the call site that runs the next task on the task queue. This call is polymorphic (since different tasks handle the `run` message differently), and by invoking a different procedure almost every call defeats the traditional inline-caching optimization [4] intended to speed monomorphic call sites. The result is that the overhead to handle this single call site is the same as the total optimized C time of the benchmark. We think we could nearly eliminate this overhead by generating call-site-specific inline-cache miss handlers. If implemented, this would probably increase the performance of the **richards** benchmark to 25%.

### 6.2 Compile Time

We have not yet optimized compile time in the new compiler and the measurements suggest we will need to. Almost all of benchmarks take from 15 to 35 seconds to compile with the new compiler. We expect that these numbers can be reduced quite substantially because the old SELF compiler compiles most of the programs in less than a second. By contrast, the C compile takes about three seconds for most of these programs. We suspect that new SELF compiler contains a few exponential algorithms for data flow analysis and register allocation, and we hope to improve them.

### 6.3 Code Space

The new compiler's generated code size is about four times larger than for the optimized C programs. However, the difference cannot be blamed solely on our new techniques. In fact, the original SELF compiler uses even more space than the new SELF compiler. A substantial part of the space overhead can be attributed to the large inline caches for dynamically-bound procedure calls and to code handling primitive failures like overflow checking and array bounds checking. We have done only rudimentary work on conserving compiled code space, and expect to be able to reduce this space overhead.

Even with the current compiled code sizes, large SELF applications can be executed without an exorbitant amount of code space. For example, our prototype graphical user interface and its supporting data structures are written in 7000 lines of SELF source code and compile to less than a megabyte of machine code (more space is currently used to store debugging and relocation information for the compiled code). In addition, since SELF compiles code dynamically, it need only maintain a working set of code in memory; unused compiled code is flushed from the code cache to be recompiled when next needed. Although final proof must await larger SELF programs, we believe that extra code space will not be a problem.

## 7 Related Work

Other systems perform type analysis over programs without external type declarations. ML [14] is a statically-typed function-oriented language in which the compiler is able to infer the types of all procedures and expressions and do static type checking with virtually no type declarations. Researchers have attempted to extend ML-style type inference to object-oriented languages, with some success [15, 16, 23, 24]. However, most of these approaches use type systems that describe an object's interface or protocol, rather than the object's representation or method dictionary. While this higher-level view of an object's type is best for flexible polymorphic type-checking, it provides little information for an optimizing compiler to speed programs.

A different approach is taken by the Typed Smalltalk project [10, 11]. Their type system is based on sets of classes, and a variable's type specifies the possible object classes (not superclasses) that objects stored in the variable may have. If the number of possible classes associated with a variable is small, then messages sent to the variable can be inlined (after an appropriate series of run-time type tests).

The Typed Smalltalk system includes a type inferencer that infers the types of most methods and local variables based on the user-declared types of instance variables, class variables, global variables, and primitives [6, 7]. The type inferencer is based on abstract interpretation of the program in the type domain, and an expression is type-correct if and only if the abstract interpretation of the expression in the context of the current class hierarchy is successful. The type of a method is determined by partially evaluating the abstract interpretation of the body of the method, and as such frequently cannot be completely determined to a simple type, but may contain unresolved constraints on the types of the method's arguments. These constraints must be checked at each call site.

This type-checking and type inference system is very powerful and should be able to type-check much existing Smalltalk-80 code. It is also suitable for optimizing compilation, since the types of variables and expressions describe their possible representations and method dictionaries. Unfortunately, their system could take a long time to infer the type of an expression, since an arbitrarily large portion of the entire system will be abstract-evaluated to compute the type of the expression.

None of these statically-typed systems handles dynamically-typed languages like SELF (the Typed Smalltalk systems disallows Smalltalk programs that cannot be statically type-checked). Our type analysis system is designed to compute as much exact type information about the receivers of messages as possible, while still handling uncertain and unknown types gracefully. It operates with a limited amount of initial type information (just the type of the receiver and the types of constant slots), and so attempts to extract and preserve as much new type information as it can.

Range analysis is performed in many traditional optimizing compilers. However, Fortran compilers typically determine subrange information by looking at the bounds specified in `do` loops. This approach doesn't work in languages with user-defined control structures like SELF, Smalltalk, and Scheme [18], since the compiler has no fixed `do` construct to look for loop index ranges. Our approach of computing range information based on primitive arithmetic and comparison operators rather than high-level statements lets our compiler perform range-based optimizations (like eliminating overflow checks and array bounds checks) in the context of user-defined control structures.

A useful extension to this scheme would be to record the results of comparisons with non-constant integer values, in case the same comparison is performed again. This would help eliminate many array bounds checks where the exact size of the array is unknown, but the index is still always less than the array length, and so the array bounds check can be eliminated. Our current range analysis cannot eliminate these bounds checks, since the integer subrange of the array index overlaps the integer subrange for the array length. On the other hand, the TS compiler for Typed Smalltalk [8, 11, 13] is able to optimize many of these bounds checks away, since it uses simple theorem proving to propagate the results of conditional expressions and thus avoid repeated tests, such as that the index is less than the array length. However, their implementation only uses a single premise at a time to evaluate conditional expressions, whereas our integer subrange types can represent the combined effects of several comparisons. More work is needed to explore these approaches.

The TS compiler for Typed Smalltalk performs an optimization similar to message splitting. A basic block with multiple predecessors may be copied and some of its predecessors rerouted to the copy if a conditional expression in the basic block may be eliminated for a subset of the original block's predecessors; this is similar to local message splitting. An extension is proposed that could also copy blocks that intervened between the block containing the conditional and the predecessor(s) that would enable eliminating the conditional, similarly to extended message splitting. However, these techniques only apply to eliminating conditional expressions, and is performed after type analysis and message inlining has been completed. Our extended message splitting is performed at type analysis time as part of message inlining, and additionally can be used to split branches of the control flow graph based on any other information available at type analysis time, such as splitting for available values [1] in order to perform more common subexpression elimination.

Extended message splitting with iterative type analysis may lead to more than one version of a loop being compiled, each for different initial type bindings. This is similar to an optimization in some parallelizing Fortran compilers called *two-version loops* [17]. If a loop could be parallelized if certain run-time conditions held (e.g. that some variable was positive), then a compiler could insert a run-time test for the desired conditions before the loop, and branch to either a parallelized version or a sequential version.

Our type analysis is also similar to partial evaluation [19]. Type analysis is a form of abstract interpretation of the nodes in the control flow graph using compile-time types instead of run-time values. Our system partially-evaluates methods with respect to the customized receiver type to produce an optimized version of the method specific to that receiver type. Within the method, type analysis propagates type information in a similar manner as partial evaluators propagate constant information. However, our compiler terminates over all input programs, while partial evaluators traditionally have been allowed to go into infinite loops if the input program contains an infinite loop. Partial evaluators also support more complex descriptions of their input data, and generate specialized versions of residual (non-inlined) function calls to propagate type information across procedure calls; our SELF compiler performs no interprocedural analysis or type propagation across non-inlined message sends.

## 8 Conclusions

Static type analysis is feasible even in a dynamically-typed object-oriented language like SELF. Our type analysis system computes enough static information to eliminate many costly message sends and run-time type tests. Value types serve to propagate constants throughout the control flow graph, while integer subrange types computed from arithmetic and comparison primitives are used to avoid overflow checks and array bounds checks in a language with no built-in control structures. Iterative type analysis with recompilation serves to compute accurate type information for variables used within loops.

Type information lost by control flow merges can be regained using extended message splitting. Extended message splitting is especially important within loops, and may lead to more than one version of a loop being generated. This is accomplished simply by allowing loop heads and tails to be split like other nodes; no extra implementation effort is needed to implement multi-version loops. Typically, one version of a loop will work for the common case types (e.g. integers and arrays), and contain no type tests and few overflow checks. Another version of the loop will be more general and contain more type tests and error checks, but will only be executed for unusual run-time type conditions.

Iterative type analysis, integer subrange analysis, and extended message splitting are powerful new techniques that have nearly doubled the performance of our SELF compiler. SELF now runs at nearly half the speed of optimized C, without sacrificing dynamic typing, user-defined control structures, automatic garbage collection, or source-level debugging. We feel that this new-found level of performance is making dynamically-typed object-oriented languages practical, and we hope they will become more widely accepted.

## 9 Acknowledgments

Urs Hölzle provided invaluable assistance collecting performance data. He and Bay-Wei Chang helped improve the visual appearance of this paper significantly.

## References

1. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA (1986).
2. Chambers, C., and Ungar, D. 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*. Published as *SIGPLAN Notices*, 24, 7 (1989) 146-160.
3. Chambers, C., Ungar, D., and Lee, E. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 49-70. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
4. Deutsch, L. P., and Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages* (1984) 297-302.
5. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).
6. Graver, J. O. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. Ph.D. thesis, University of Illinois at Urbana-Champaign (1989).
7. Graver, J. O., and Johnson, R. E. A Type System for Smalltalk. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages* (1990) 136-150.
8. Heintz, R. L., Jr. *Low Level Optimizations for an Object-Oriented Programming Language*. Master's thesis, University of Illinois at Urbana-Champaign (1990).
9. Hennessy, J. Stanford integer benchmarks. Personal communication (1988).
10. Johnson, R. E. Type-Checking Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 315-321.

11. Johnson, R. E., Graver, J. O., and Zurawski, L. W. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA '88 Conference Proceedings*. Published as *SIGPLAN Notices*, 23, 11 (1988) 18-26.

12. Lee, E. *Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language*. Engineer's thesis, Stanford University (1988).

13. McConnell, C. D. *The Design of the RTL System*. Master's thesis, University of Illinois at Urbana-Champaign (1989).

14. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, MA (1990).

15. Mitchell, J. Personal communication (1989).

16. Ohori, A., and Buneman, P. Static Type Inference for Parametric Classes. In *OOPSLA '89 Conference Proceedings*. Published as *SIGPLAN Notices*, 24, 10 (1989) 445-456.

17. Padua, D. A., and Wolfe, M. J. Advanced Compiler Optimizations for Supercomputers. In *Communications of the ACM*, 29, 12 (1986) 1184-1201.

18. Rees, J., and Clinger, W., editors. *Revised$^3$ Report on the Algorithmic Language Scheme* (1986).

19. Sestoft, P., and Søndergaard, H. A Bibliography on Partial Evaluation. In *SIGPLAN Notices*, 23, 2 (1988) 19-27.

20. Ungar, D. M. *The Design and Evaluation of a High-Performance Smalltalk System*. Ph.D. thesis, the University of California at Berkeley (1986). Published by the MIT Press, Cambridge, MA (1987).

21. Ungar, D. A Performance Comparison of C, SELF, and Smalltalk Implementations. Unpublished manuscript (1989).

22. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. Also to be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

23. Wand, M. Complete Type Inference for Simple Objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science* (1987) 37-44.

24. Wand, M. Type Inference for Record Concatenation and Multiple Inheritance. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science* (1989) 92-97.

# Appendix A  Performance Data

**Compiled Code Speed (as a percentage of optimized C)**

| benchmark | ST-80 | old SELF-89 | old SELF-90 | new SELF[1] |
|---|---|---|---|---|
| **stanford** | | | | |
| perm | 7% | 18% | 13% | 24% |
| perm-oo | 8% | 28% | 20% | 56% |
| towers | 8% | 21% | 16% | 23% |
| towers-oo | 19% | 34% | 21% | 43% |
| queens | 9% | 19% | 14% | 26% |
| queens-oo | 10% | 34% | 16% | 35% |
| intmm | 10% | 16% | 19% | 35% |
| intmm-oo | 6% | [2] | 22% | 40% |
| puzzle | 5% | 13% | 9% | 19% |
| quick | 9% | 19% | 14% | 28% |
| quick-oo | 10% | 21% | 17% | 40% |
| bubble | 6% | 10% | 9% | 22% |
| bubble-oo | 7% | 14% | 9% | 63% |
| tree | 53% | 48% | 41% | 47% |
| tree-oo | 80% | 56% | 69% | 91% |
| **small** | | | | |
| sieve | 5% | | 7% | 23% |
| sumTo | 10% | 22% | 11% | 24% |
| sumFromTo | 10% | | 10% | 21% |
| sumToConst | 10% | | 11% | 33% |
| atAllPut | 6% | | 12% | 53% |
| **richards**[3] | 9% | 26% | 17% | 21% |

[1]Since this paper was originally published, the compiler's implementation has been refined. Execution performance is now faster than reported in this paper (over 60% the speed of optimized C for the stanford and stanford-oo benchmarks).

[2]Empty entries in the performance tables indicate unavailable information.

[3]See section 6.1 for a discussion of the performance results for **richards**.

## Appendix B Code Space

**Compiled Code Size (in kilobytes)**

| benchmark | Optimized C | old SELF-90 | new SELF |
|---|---|---|---|
| **stanford** | | | |
| perm | 2.4 | 7.5 | 5.8 |
| perm-oo | – | 10.1 | 7.1 |
| towers | 3.1 | 12.1 | 16.2 |
| towers-oo | – | 7.0 | 7.4 |
| queens | 2.5 | 11.7 | 12.0 |
| queens-oo | – | 9.1 | 8.0 |
| intmm | 2.5 | 11.2 | 5.7 |
| intmm-oo | – | 18.5 | 8.3 |
| puzzle | 5.0 | 81.3 | 41.3 |
| quick | 2.8 | 14.1 | 11.9 |
| quick-oo | – | 16.2 | 10.2 |
| bubble | 2.7 | 11.5 | 6.7 |
| bubble-oo | – | 8.0 | 5.9 |
| tree | 3.3 | 13.2 | 9.5 |
| tree2 | – | 12.1 | 7.2 |
| **small** | | | |
| sieve | 0.5 | 5.3 | 1.5 |
| sumTo | | 2.6 | 1.6 |
| sumFromTo | | 2.9 | 1.8 |
| sumToConst | | 2.6 | 0.8 |
| atAllPut | | 2.2 | 0.9 |
| **richards** | 6.1 | 34.3 | 25.5 |

# Appendix C Compile Time

**Compile Time (in seconds of CPU time)**

| benchmark | Optimized C | old SELF-90 | new SELF[1] |
|---|---|---|---|
| **stanford** | | | |
| perm | 2.8 | 0.58 | 11.8 |
| perm-oo | – | 0.85 | 19.8 |
| towers | 3.7 | 0.73 | 31.9 |
| towers-oo | – | 0.37 | 7.6 |
| queens | 3.1 | 0.71 | 65.4 |
| queens-oo | – | 0.62 | 25.2 |
| intmm | 2.9 | 0.84 | 20.7 |
| intmm-oo | – | 1.1 | 30.1 |
| puzzle | 9.1 | 6.9 | 362.3 |
| quick | 3.0 | 0.70 | 122.9 |
| quick-oo | – | 0.90 | 123.3 |
| bubble | 2.9 | 0.63 | 15.9 |
| bubble-oo | – | 0.55 | 21.5 |
| tree | 3.9 | 0.56 | 10.2 |
| tree-oo | – | 0.74 | 7.0 |
| **small** | | | |
| sieve | 1.6 | 0.40 | 6.4 |
| sumTo | | 0.31 | 5.2 |
| sumFromTo | | 0.29 | 5.7 |
| sumToConst | | 0.32 | 2.9 |
| atAllPut | | 0.20 | 1.4 |
| **richards** | 13.4 | 2.1 | 35.6 |

[1]Since this paper was originally published, the compiler's implementation has been refined. Compilation speed is now between one and two orders of magnitude faster (about the same speed as the optimized C compiler).