University of Alberta

A First Implementation of Modular Smalltalk

by

Wade Holst
Duane Szafron

Technical Report TR 93-07
May 1993

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

A First Implementation of Modular Smalltalk

Wade Holst
Duane Szafron

Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1

{wade, duane}@cs.ualberta.ca

ABSTRACT

A "first" implementation of the Modular Smalltalk object-oriented programming language is presented. The implementation includes an object-oriented parser, object-oriented representation for code fragments and an object-oriented C-code generator, all implemented in Smalltalk-80. This implementation validates two of the five design principles of the Modular Smalltalk language and provides a vehicle for validating the other three design principles. The macro-based C-code generator is easily adaptable to generating production code in other languages like assembler. In addition, the generation technique applies to source languages other than Modular Smalltalk. The implementation includes an efficient method dispatch based on new extensions to incremental cache table coloring.

KEY WORDS: Modular Smalltalk, object-oriented, programming language, compiler, interpreter, method dispatch, code generation.

1. Introduction

Modular Smalltalk (MS) is an object-oriented descendant of the Smalltalk-80 (ST-80) [Gold83] programming language. Several desirable goals for an object-oriented language guided its specification. These goals were [WBW88] :

1. to provide a simple, consistent execution semantics,
2. to increase programmer productivity through code reuse and code re-definition,
3. to provide efficient facilities for design and implementation in multiple programmer applications,
4. to provide for efficient implementations, and
5. to be simple enough for new users to learn easily.

This paper describes a "first" implementation of MS (there are no previously published implementations, although there may be some unpublished ones in the commercial sector). The implementation is a prototype one, implemented in ST-80. ST-80 was chosen to allow us to quickly experiment with different implementation strategies and interpretations of the language specification. However, since one of the language goals is to provide for efficient implementations, the architecture that was developed is an efficient one and C-code can be generated and compiled on a production C-compiler. Section 1 gives a brief specification of the language. A more complete description can be found in [Tek89]. Section 3 describes the internal representation of MS programs as ST-80 objects. The internal representation was designed to separate the extensive compile-time responsibilities from the minimal run-time responsibility (execution). Compile time responsibilities include: symbol tables, coloring dispatch tables, dispatch optimization, de-compilation, and generation of an equivalent C language program. An interpreter written in ST-80 can be used to interpret this internal form. Section 4 describes the parser that translates MS to the internal form. The parser is object-oriented and implemented in ST-80. Section 5 describes our approach to method dispatch. We have chosen an incremental coloring [AR92] approach instead of the slower dynamic look-up approach of ST-80. Section 6 describes the C-code generator that translates the internal representation of a program into an equivalent C program that can be compiled and executed with any ANSI C compiler. Section 7 discusses our conclusions and future directions. Appendix A contains a sample of the C-code generated and Appendix B contains the method dispatch algorithm.

The research contributions of this paper are:

1. Validation of the design goal that the execution semantics of Modular Smalltalk are consistent.
2. Validation of the design goal that an efficient implementation of Modular Smalltalk is possible.

3. The generation of a platform to support the validation of the other three design goals of Modular Smalltalk: increased programmer productivity through code reuse and code re-definition, design and implementation efficiency for multiple programmer applications and simplicity for new users.

4. A concise description of an object-oriented parser, that can serve as the basis for a parser framework.

5. Corrections to the André-Royer [AR92] incremental coloring algorithm and extensions that support inheritance exceptions which occur in Modular Smalltalk.

6. A new macro approach to C-code generation that can easily be modified to generate assembly language code and could be adapted for other programming environments.

2. Modular Smalltalk

Since a complete definition of Modular Smalltalk has appeared in [Tek89], this section gives only a brief description of the language in terms of its differences from ST-80.

2.1 Literals

MS provides for literal representations of integers, floating point numbers, strings, selectors, and arrays of these literals. Such literals are immutable, so the object represented by a literal is always apparent from its lexical representation. The result of attempting to modify literals is implementation dependent. Any object with a literal representation can be used in literal arrays.

2.2 Messages

Message sending in MS follows the normal conventions established by ST-80. All messages return a single object when method execution is complete. A method is executable code that can retrieve state information from the receiver object, modify the state of the receiver object and initiate other messages. Methods can be recursive. The precedence of message sends is the same as in ST-80.

2.3 Identifiers and Expressions

There are three types of identifiers: module constants, method parameters, and temporary variables. Of these three, only temporary variables may have their values changed via an expression. An expression may consist solely of a message send, or it may assign a value to multiple variables.

2.4 Identifier Scoping

Temporary variables are the only variables in MS. Their scoping rules are more complex than for Smalltalk. An object bound to an identifier declared within a block is inaccessible by that identifier outside the block. The same rule applies to identifiers that are declared in modules (module constants), unless the identifier is exported by the declaring module and imported by another module. Identifier scopes can be statically nested by nesting blocks, but modules cannot be nested.

2.5 State

Unlike most other O-O languages, MS has *no* instance variables. Instead, state is represented by groups of messages that are implemented as *stored methods* as opposed to *computed methods*. Named instance variables from Smalltalk (or member-data from C++) are replaced by a pair of messages (accessing and assignment) for a simple state. The assignment message has as its single argument an object that will be used as the new state value of the receiving object and the return value of this message is this new state value. The return value of the accessing message is the last value assigned by the corresponding assignment message, or *nil* if the assignment message has not yet been sent.

Indexed instance variables from Smalltalk are replaced by a set of four messages that define an indexed state. An accessing method uses a single-keyword selector and its argument specifies which part of the state to access. Two different types of indexed state exist: object-valued and byte-valued. If no value has been set at the given index, the value *nil* is returned for object-valued states, and the value 0 is returned for byte-valued states. Object-valued states may assume any value, whereas byte-valued states are restricted to objects representing integers in the range 0 to 255 inclusive. The assignment method uses a two-keyword selector where the first argument is an index that identifies part of the indexed state and the second argument is the new value for that part of the state. The method returns the second argument. Two other selectors are associated with indexed states. One returns the number of indexable states associated with the two selectors above. The other sets the total number of indexable states dynamically.

2.6 Methods

A method declaration consists of a message selector, a visibility attribute and a method. A method declared as *private* is only understood (executed) if its sender and receiver are in the same class. Unlike C++, there is no protected visibility mode that allows senders to be objects from subclasses. This is due to the philosophical view that subclasses are clients of their superclasses with no special status beyond what any other client class can claim.

There are six types of method implementation. The first three are abstract, undefined and primitive. An abstract or undefined method requires no more information. A primitive method is uniquely specified by its class and selector and does not require any other MS information. The fourth type of method, called an *aliased* method, renames a method that is inherited from an immediate superclass. The fifth type of method is a *stored* method that specifies part of a state declaration, as described in Section 2.5.

The sixth type of method is a *block* method. A block method consists of a list of formal parameters, a list of temporary variables, and a list of expressions. When a block is evaluated, it returns a closure. A closure consists of the block and a context. The context maintains the values of variables defined within an enclosing block during a particular execution of the method represented by that block.

2.7 Classes

A class declaration can inherit from zero or more existing class declarations, as long as the intended superclasses are visible within the module containing the class declaration and as long as the intended superclasses do not inherit from the class being defined (the inheritance hierarchy must be acyclic).

A class declaration consists of an instance behavior (instance methods) and a class behavior (class methods). Class declarations are static. They are not objects or expressions and do not exist during the execution of a program. Thus, classes cannot be created at run-time or have their behavior modified. However, each class itself is an object and its state can change at run time in response to class messages.

The instance behavior of a class consists of the locally declared instance behavior and instance behaviors inherited from all superclasses. The class behavior is similarly derived. There are four rules governing the merging of local and inherited behavior.

1. All method declarations in the local behavior are included and all inherited method declarations with the same selector are excluded.
2. Otherwise, if a selector is bound to exactly one inherited method, that binding is included. This is true even if the binding is shared by more than one superclass. A method is shared only if the superclasses in question inherited the method from the same ancestor, and none of the ancestors between that ancestor and the superclass (including itself) re-defined a selector with the same name.
3. Otherwise, if all but one of the bindings from the inherited behavior is declared as abstract, the non-abstract binding is included.
4. Otherwise, there is a conflict and the program is invalid.

2.8 Modules

Modules manage the scope of names. A module consists of a module name and a set of constant bindings between names and objects. Each binding has one of three forms: an import declaration, a class declaration or an expression declaration. When a module imports an object by name from another module, it may create a local alias for that object in the importing module. Each name binding in a module has an associated visibility attribute, *private* or *public*, that determines whether other modules can import the name. Modules are not objects and do not exist during the execution of a program.

Modules also provide a facility for class extension, whereby behavior can be added to a class (which was presumably imported from a different module). Class extensions are not allowed to remove or redefine existing behavior.

3. The Internal Representation of MS Programs

This section describes the internal representation of MS programs. Each subsection describes one or more of the classes that represent program fragments. These classes make extensive use of inheritance. The implementation described in this paper was written in ST-80, but another object-oriented language could have been used instead.

An MS program consists of a hierarchical collection of node objects. The internal representation was designed to separate the extensive compile-time responsibilities from the minimal run-time responsibilities. Compile-time responsibilities include parsing, de-compilation, re-coloring and generation of an equivalent C language program. However, since the implementation is object-oriented, most of the compile-time responsibilities are distributed throughout the nodes themselves.

The separation was accommodated by defining a ST-80 run-time class that represents each syntactic aspect of an MS program. Additional compile-time behavior was then added by defining a compile-time subclass for each run-time class. Essentially, run-time objects know only how to evaluate (execute) themselves while compile-time objects have the appropriate additional responsibilities. We intend to extend this object-oriented approach to parsing and program fragment representation to produce a framework for object-oriented parsing, representation and code-generation. Such a framework would represent BNF grammar rules as objects that are instances of special node classes.

3.1 MSObject and MSClass

Every instance of MSObject knows its class and its state. The state is divided into three parts: named sub-states, indexed sub-states and byte-indexed sub-states. Syntactically, a class consists of a list of superclasses, an instance behavior, and a class behavior. Required run-time information consists of the number of named, indexed and byte indexed states for the instance behavior. In addition, if a naive recursive look-up method dispatch strategy is used, then a map of selectors to methods and visibility attributes is required for instances and for classes. The equivalent compile-time class stores a mapping from selectors to methods for both instance and class behaviors.

3.2 MSName, MSContext

An MS variable is represented by an instance of class MSName. It consists of a level and an offset that specifies where to find the object bound to the name, starting from the current context. A context is represented by an instance of MSContext and consists of an ordered list of objects bound to all names visible at the current level, plus a reference to the parent context.

3.3 MSStatement, MSAssignment, MSReturn and MSModuleExpression

The MSStatement class is an abstract superclass for all classes that contain a non-trivial reference to an expression: MSReturn, MSAssignment, MSModuleExpression and MSMessageSend. An explicit

return within a block is stored in an MSReturn, and consists solely of the expression to be returned. An assignment consists of an MSName to represent the variable and an expression whose evaluated result will be assigned to this variable at run-time. An MSModuleExpression is a wrapper than contains one of: an MSMessageSend, an MSName or an MSLiteral.

3.4 MSMessageSend, MSCascadedMessageSend and MSMessage

Syntactically, a message send contains a receiver, a selector, and zero or more arguments. However, an instance of MSMessageSend consists of a receiver and a message. The receiver can be any expression. The message is an instance of MSMessage and consists of the selector and the arguments. The receiver-message pair representation simplifies support for cascaded message sends that have a single receiver and multiple selector-arg pairs. A MSCascadedMessageSend is similar to an MSMessageSend except that it contains an array of MSMessages instead of a single one.

3.5 MSModule, MSImport and MSClassExtension,

At run-time, an MSModule maintains a context of name-value pairs. The compiler replaces names by levels and offsets, and the offsets are used to index the context. An MS module can bind a name to one of four things, an MSClass, an MSImport, an MSClassExtension, or an MSModuleExpression.

A module can bind a name to the object represented by a name that is imported from another module. An MSImport object contains an MSName, whose offset reflects the position in the context of the exporting module. Since an exporting module is executed before a module that imports from it, the object will exist when it is referenced. MSClassExtensions have not been implemented yet.

3.6 MSBlock

Syntactically, a block contains lists for: formal arguments, temporary variables and message expressions. The only information needed at run-time is the number of arguments and temporaries, and the list of statements. The compile-time node maintains the list of arguments and the list of temporary variables in dictionaries to facilitate rapid look-up.

3.7 MSMethod

MSPrimitive methods have an associated ST-80 method and C function name. The former is used for interpretation in the environment and the latter is used during C-code generation. Within a class, each MSStateMethod is assigned a unique state index. There are 10 subclasses, representing all possible types of state methods (named, indexed, byte indexed; accessing, assignment, size assignment). MSBlock methods have already been described.

The node classes for MSCAbstractMethod, MSCUndefinedMethod and MSCAliasedMethods are place-holders for de-compilation, since such methods have a trivial implementation. MSCAliasedMethods store a reference to an immediate superclass and the selector from this class which is to be renamed.

3.8 MSClosure

An MSClosure contains a block and a context. At run-time, an MSClosure exists for each method or literal block that is currently executing and for some literal blocks that are not executing. An MSClosure is created for a literal block when the code that contains it is evaluated. The context contains a static link to the containing block. The code for this MSClosure is executed when the block itself is evaluated. For example, assume that a literal block is bound to a name in one method and passed to a second method as an argument. Assume that the block is evaluated in the second method by sending it a value message. An MSClosure is created when the literal block is bound to a name in the first method and is the object that is passed to the second method as the argument block. Since recursive and mutually recursive calls can be made, multiple closures can exist for the same method block.

3.9 MSReturnObject

An MSReturnObject is a special wrapper that contains an MSObject. Consider a method block that contains a literal block. If an explicit return is executed in the literal block, then control must be returned to the caller of the method block. Note that the literal block may have been passed as an argument to an arbitrary number of methods before it was actually executed.

The literal block returns an MSReturnObject to its caller. The MSReturnObject contains the actual return object as well as the method in which the literal block lexically appears. When the caller receives an MSReturnObject, it returns this object to its caller without executing any more code. This process continues until the method that lexically contains the literal block is reached. This method, removes the MSReturnObject "wrapper" and returns the actual result that it contains. The mechanism for determining the lexical container of a literal block is based on a static link comparison.

4. The Parser

The previous section described the run-time responsibilities of MS objects. This section describes their parsing responsibilities. In addition to the node classes, some utility classes are also used for parsing. These include a scanner (MSScanner), a programming environment (MSPEnv), an inheritance conflict resolution class (MSConflictSet) and many token classes (subclasses of MSToken). This approach maps BNF rules to node classes. We are currently developing a framework that encapsulates this approach and it will be described in a future paper.

4.1 Programming Environment

An MSPEnv provides facilities for browsing and manipulating the internal representation of MS code. In addition, it maintains the method cache tables and performs the incremental coloring and re-partitioning needed to provide efficient method dispatch. A discussion of this incremental coloring algorithm, and method dispatch in general is given in Section 5.

4.2 Tokens

All token classes are subclasses of MSToken. As with most languages, the distinction between lexics (that define tokens) and syntax (that defines parse nodes composed of tokens) is somewhat ambiguous. In this implementation of MS, all character sequences representing syntactic delimiters are tokens, identifiers are tokens, and all MS literals are tokens, even literal arrays. Nothing else is a token.

4.3 Scanner

An MSScanner is responsible for converting an MS source string into tokens. It knows the string to be parsed and the current position in the string. The primary behaviors of an MSScanner are to scan the next token and to peek at the next token(s). For MS, a *look-ahead* of more than two tokens is never required. For tokens that consist of a single character, a token object of the appropriate class is returned immediately. For longer tokens, dedicated methods are called based on the initial character. The scanner is also responsible for error handling. If a syntax error occurs either during the tokenization process, or within a node parse, a message is sent to the scanner, who reports the error to the user.

4.4 Node classes

There is no parser class for MS. Instead, each compile-time node knows how to parse the MS source string that it represents. Each node object is passed an instance of a scanner that accesses the program segment being parsed. A node parses its source by continually asking the scanner for the next token and performing activities based on the type of the resulting token and its definition in the grammar. Since node objects are composed of other nodes, each node requires only a few tokens before passing the responsibility of parsing to one of its components. That is, a node obtains its terminal information by asking the scanner for tokens. When appropriate, it creates a component node of the appropriate class and tells its component node to parse itself. In the case when the class of a component node may be dependent on the next token, the node peeks at the next token before creating the component object and telling it to parse itself. When a node finishes parsing itself, it returns itself to its parent node.

5. Method Dispatch

As with other object-oriented languages, the selection of a method dispatch algorithm for MS is a trade-off between time and space [CU91]. The basic ST-80 method dispatch algorithm is a dynamic one, starting in the method dictionary of the receiver of the message[GR 89]. If no method for the message selector is found, the method dictionary of the superclass is checked. If this process continues to the root class, *Object*, and no method is found there, a *messageNotUnderstood* method is invoked to warn the user. This algorithm is slow for long inheritance chains and is even slower in MS where multiple inheritance is involved and multiple superclass chains need to be searched.

On the other hand, the fastest possible implementation would be a cache table which stores a method to use for each class-selector pair. At run-time, a single look-up in this table would contain a reference to the correct method. Although this approach is fast, it is not feasible since the space required is the product of the number of classes and selectors.

A slightly slower cache algorithm [DMSV89] involves the use of a two dimensional cache table with classes that index the columns and colors that index the rows. The algorithm assigns a color to each selector, and it is possible for two different selectors to be assigned the same color. For example, consider the case where there are three groups of classes in a program. Assume the first group recognize the message selector *alpha* but not *beta*. Assume the second group recognize the message selector *beta*, but not *alpha*. Assume that the third group recognizes neither. A single color can be used to represent the selectors *alpha* and *beta*. Cache table entries for the first group in this color row would contain the method for *alpha*. Cache table entries for the second group would contain the method for *beta* and cache table entries for the third group would contain *nil*.

However, as pointed out in [AR92], this method requires prohibitive initial time to perform coloring, since recoloring may be required after adding even a single selector. The same paper suggests an improved algorithm that is based on incremental coloring and we will refer to this algorithm as the André-Royer algorithm. Our iterative color/partition (ICP) algorithm is based on the André-Royer algorithm. Although the ICP algorithm was implemented for MS, it is described in an language-independent manner so it can be applied to other object-oriented languages as well. Unlike the André-Royer algorithm, the ICP algorithm supports inheritance exceptions that can occur in MS when a message is removed from the protocol of a class by aliasing it to a new name.

We are currently investigating optional static typing in MS so the actual ICP algorithm is more complex than the version presented in this paper. For example, the actual ICP algorithm is responsible for recording class-selector pairs where the selector is not redefined in any sub-tree. This information is used to bypass the cache table by inserting a method address directly in the code. We are preparing a separate paper that presents the complete algorithm and introduces some optimizations using multiple cache tables.

5.1 ICPA Dispatch Classes

Four classes are defined to implement the ICP algorithm.

MSSelector and MSDivison

Each MSSelector has a unique integer index, a color (integer) and a set of defining classes. The defining classes do not include classes that just inherit the selector. For each selector, ICP logically divides the classes that recognize it into mutually exclusive MSDivisions. Each division determines a set of classes that form a connected component of the inheritance hierarchy and use the same method for a

given selector. The root of this sub-graph is the defining class for the selector and no other class in the sub-graph defines the selector.

MSCacheTable

An MSCacheTable contains a two dimensional array (matrix) with rows indexed by colors and columns indexed by class indexes. The values of the matrix are MSDivisions. In each row, all of the classes in the same MSDivision have the same cache table entry. Any class that does not support a selector for a given color contains an MSEmptyDivision in its column.

The MSCacheTable supplies other services in addition to the cache matrix. It maintains maps for: ST-80 selector symbols to MSSelectors (for efficient compilation), ST-80 selector symbols to unique selector indices (for efficient execution) and unique selector indexes to colors (for efficient execution).

At run-time, when executing a message send, the MSMessageSend node knows its receiver and MSMessage. The receiver knows its class index. The MSMessage knows its selector index. To find the method, the selector index is used to obtain the color index. The color index and class index are used to access a color/class entry in the cache. Note that a run-time check must be performed to ensure that the selector associated with the stored method is the same as the current selector (due to the coloring algorithm mapping multiple selectors to the same color). This overhead will be discussed in more detail later.

Our implementation contains two cache tables, one for instance methods and one for class methods. For each selector, the compiler generates two indices, one for each cache. At run-time, the appropriate index is used, depending on whether the receiver is a class object or not. It is possible to combine these two cache tables into one, but the compiler must still generate two indices because the same message selector may be used for a class and an instance message and there is no way to tell at compile-time whether the receiver is a class or not. We are currently studying the merits of one cache table versus two as well as multiple cache tables, one for each tree in the inheritance forest.

MSConflictSet

An MSConflictSet is responsible for resolving message selector name conflicts due to multiple inheritance. It does this by recording conflicts and removing them when aliases are compiled. When all conflicts have been removed, an MSConflictSet inserts the correct division into the cache table.

5.2 Actions During Class Definition

When a class declaration is compiled, a new class column is added to the cache table. Each entry in the new column is a shared instance of MSInitialDivision (one of two subclasses of MSEmtpyDivision). For each color, the corresponding entries in all superclass columns are compared. If they are identical, then that division is copied to the new class/color entry. Otherwise, the action taken depends on the conflict resolution scheme of the language. These may include: reporting an error, picking one of the conflicting entries by some algorithm or waiting until conflicts have been resolved, as is done in MS where

conflicting methods (i.e. methods with different implementations but identical selectors) must be aliased to enforce uniqueness. The MSConflictSet class handles this form of resolution automatically.

A common form of conflict occurs when two classes, say *A* and *B*, have different selectors stored at the same color (say *alpha* and *beta* at color 1) and a new class, *C*, inherit from both of them. Then *alpha* and *beta* would no longer be allowed to have the same color. If such a conflict is detected during inheritance copying, one of these selectors must be moved to a different color. In addition to changing the color of the associated MSSelector, this involves moving the MSDivisions from the original color row to the new color row and placing a shared instance of MSInitialDivision in the old row.

5.3 The ICP Algorithm

The cache table also needs to be modified when each new method is compiled. The definitions in Table 1 and Table 2 are required to understand this process. Many of the definitions are loosely based on the André-Royer algorithm. The letters C and C_i refer to classes, G to a group of classes, S to a selector, L to a color, D to an MSDivision and Ω to an empty cache table entry. The notation $C_i < C$ means that class C_i is in the inheritance sub-graph with root class C. Table 2 defines partition types for divisions, where S = divisionSelector(D) and C = divisionDefiningClass(D).

Symbol	**Definition**
divisionAt[L, C]	the cache table entry for color L and class C
color(S)	color mapped to selector S
divisionSelector(D)	the selector associated with division D
divisionDefiningClass(D)	the root class of the division D
definedBehavior(C)	set of all selectors explicitly defined in class C
allSubClasses(C)	$= \{C_i \mid C_i < C\}$
allSuperClasses(C)	$= \{C_i \mid C < C_i\}$
relatedClasses(C)	$=$ allSubClasses(C) $\cup$ allSuperClasses(C) $\cup \{C\}$
colorsFreeFor(G)	$= \bigcap_{C \in G} \{ L \mid \text{divisionAt}[L, C] \neq \Omega\}$
classesUsingColor(L)	$= \{C \mid \text{divisionAt}[L, C] \neq \Omega\}$
classesDefiningSelector(S)	$= \{C \mid S \in \text{definedBehavior}(C)\}$
subBehavior(C)	$= \bigcup_{C_i \in \text{allSubclasses}(C)} \text{definedBehavior}(C_i)$
superBehavior(C)	$= \bigcup_{C_i \in \text{allSuperclasses}(C)} \text{definedBehavior}(C_i)$
dependentClasses(D)	$= \{C_i \mid C_i < C$ and $S \notin \text{definedBehavior}(C_i)$ & if $S \in \text{definedBehavior}(C_j)$ then not $(C_i < C_j)\}$

Table 1 Definitions for the ICP Algorithm

Partition	**Definition**
specific(D)	= [classesDefiningSelector(S) = {C}]

separate(D)	= [relatedClasses(C) ∩ classesDefiningSelector(S) = {C}]
redefined(D)	= [S ∈ superBehavior(C)]
declared(D)	= [not specific(D) and not separate(D) and not redefined(D)]

Table 2 Partition types for divisions.

Figure 1 shows the ICP algorithm. Figure 2 shows a small inheritance graph based on the graph from [AR92], but with an inheritance exception added. The exception occurs by aliasing the *o* message to *a* in class A (denoted o -> a). Table 3 shows the results of coloring this graph using the ICP algorithm.

```
Algorithm ICP(in C : Class, in S : Selector, in/out T: CacheTable)

let D = a new division with divisionSelector(D) = S
   and divisionDefiningClass(D) = C and let L = color(S)
if specific(D) then
   set color(S) from colorsFreeFor(allSubClasses(C) ∪ {C})
else
   let E = divisionAt[C, L]
   if (divisionDefiningClass(E) = C) then
      return
   else
      if (divisionSelector(E) ≠ S) then
         MoveSelectorToFreeColor(divisionSelector(E))
      endif
      if separate(D) or declared(D) then
         if (dependentClasses(D) ∩ classesUsingColor(L) = ϕ) then
            no color change
         else
            let G = classesUsingColor(color(S)) ∪ dependentClasses(D)
            set color(S) from colorsFreeFor(G)
         endif
      else /* redefined */
         no color change
      endif
   endif
endif
for Ci in dependentClasses(D)
   set divisionAt[C, L] = D
endfor
end ICP

Procedure MoveSelectorToFreeColor(S)
   G = classesUsingSelector(S)
   select L from colorsFreeFor(G)
   set D = new MSInitialDivision
   for Ci in G
      set divisionAt[Ci, L] = divisionAt[Ci, color(S)]
      set divisionAt(Ci, color(S)] = D
```

Figure 1 The ICP algorithm

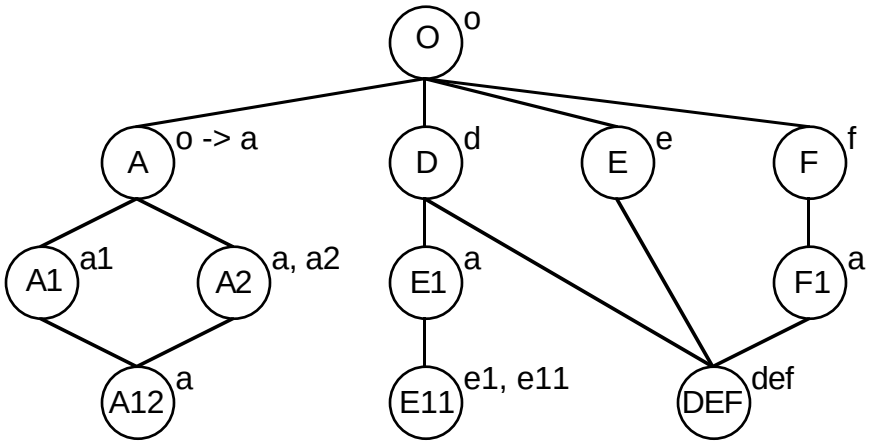

Figure 2 An example inheritance graph with an inheritance exception

L/C	O	A	A1	A2	A12	D	E1	E11	E	F	F1	DEF
1	Ω	a:A		a:A2	a:A12	Ω	a:E1	a:E1	Ω	Ω	a:F1	a:F1
2	Ω	Ω	a1:A1	Ω	a1:A1	Ω	Ω	Ω	Ω	f:F	f:F	f:F
3	o:O	Ω	Ω	a2:A2	a2:A2	o:O	o:O	o:O	o:O	o:O	o:O	o:O
4	Ω	Ω	Ω	Ω	Ω	Ω	Ω	e1:E11	e:E	Ω	Ω	e:E
5	Ω	Ω	Ω	Ω	Ω	Ω	Ω	e11:E11	Ω	Ω	Ω	def:DEF
6	Ω	Ω	Ω	Ω	Ω	d:D	d:D	d:D	Ω	Ω	Ω	d:D

Table 3 An ICP generated coloring for the inheritance graph of Figure 2

To test the algorithm on real graphs, we developed an ST-80 to MS conversion program. We are translating the Collection classes and modifying them to reflect the multiple inheritance class re-factorization developed by [Coo92]. So far, we have translated and re-factored the ST-80 collection classes: Collection, SequenceableCollection and Set to the MS classes: Collection, IndexedCollection, SequenceableCollection and Set. The ICP algorithm generated 49 colors for the 90 selectors.

Comparisons with the André-Royer Algorithm

Figure 3 contains the André-Royer algorithm, using the definitions from this paper. The following notes compare it with algorithm ICP.

```
Switch partition of S
1. specific: find free color for subclasses of C                    (see NOTE 1)
2. redefined: no change
3. separate
      If color(S) ∈ freeColorsFor({C}) then no change                (see NOTE 2)
      else find free color for classesUsingSelector(color(S))        (see NOTE 3)
4. declared
      If ∀ X < C, color(S) ∈ colorsFreeFor({X}) ∨ S ∈ definedBehavior(X)  then
         no change                                                   (see NOTE 4)
      else find free color for classesUsingColor(color(S))           (see NOTE 5)
```

Figure 3 The André-Royer Algorithm

1. The André-Royer Algorithm makes the assumption that there are no exceptions to inheritance. Since MS does allow for such exceptions (by aliasing an inherited selector), the ICP algorithm must check class C for a free color as well. Without exceptions, we don't need to check C. In fact, we don't really have to check all subclasses either, only the leaf subclasses. With exceptions, a superclass may use selectors that a subclass does not, so the entire sub-graph must be checked.

2. In fact, it is not sufficient to check only that the current color is free in C. It must also be free in all subclasses of C. For example, suppose class B inherits from class A, and a selector *beta* is defined in class B, with color of 1. If class A later defines *alpha*, it is not sufficient to ensure that color 1 is free only for A - it must also be free for B, since B will inherit *alpha*. With this change, the code for the *separate* case now matches the code for the *declared* case so they can be merged.

3. Finding a free color for all classes currently using the current color is not sufficient. In addition, the color must be free for class C and all of its subclasses. For example, suppose classes D and E have color 3 free, and that selector *alpha* has been defined somewhere and has color 1. Suppose further that class A is in a different tree from D and E, and that *alpha* is being defined for it. If color 1 in A is not free, a new color must be found. It is not sufficient to look for free colors in D and E - obviously the color must be free in A and all of its subclasses.

4. It is not necessary to check every subclass of C against one or another of the two tests. The second test can be avoided, and the number of subclasses tested reduced by asking only for dependent subclasses (those subclasses which inherit S from C).

5. As in note 3, we must also insure the color is free for the dependent classes of C. In this case, however, this will be some subset of C and its subclasses, since some subclasses re-define S (otherwise, the partition would be separate instead of declared).

5.4 Space Efficiency (Tail Removal)

There are several ways to reduce the run-time space requirements for cache table dispatch. These reductions are often obtained by increasing compile-time space requirements which in most cases are not important.

Suppose the basic cache table has size N (colors) x M (classes). Let n be the color index of the last non-empty selector in a class column, m. The entries from row index n+1 to N are called the *tail* of column m and this tail can be discarded. Therefore the cache table can be implemented as an array of columns of variable size.

Note that the specific color associated with a group of selectors is not important; only that the selectors in the group have the same color. Thus, entire rows of the cache table can be swapped. To optimize the size of the cache table, rows should be swapped to maximize the sum of the tail sizes.

5.5 The Method Dispatch Algorithms

Appendix B contains two method dispatch algorithms. One based on cache tables, the other on the traditional ST-80 class look-up algorithm. The information required to perform method dispatch in each case is listed in Table 4.

	Cache Table Approach	**Class Look-up Approach**
Divisions	a selector and a method	a method and a visibility (public/private)
Selectors	a unique index and a color index	a unique index
Classes	an index	an index and a dictionary of divisions

Table 4 Method dispatch information

The current implementation uses the table approach with separate cache tables for class and instance selectors with tails removed. For table dispatch, the appropriate selector index and cache table are determined at run-time by determining whether the receiver is a class or instance object. Next, the color/class entry in the appropriate table is obtained. This entry is a division, but its method is not necessarily the correct one. Two exceptional conditions must be checked for; the division may be empty or the division may define a selector that is different than the one in question. In either case, the desired selector is not understood by the receiver, and an appropriate error message is generated. If neither of these cases occur, the division specifies the method to execute.

For class look-up dispatch, the execute algorithm simply asks for a division by calling *lookup_dispatch*. If the resulting division is empty, a *messageNotUnderstood* message is sent, otherwise, the associated method is executed and returned as the result.

The *lookup_dispatch* algorithm is recursive. It needs a receiver, a current class, and the instance and class selectors. The current class is initially the class of the receiver, but may change in recursive calls. The behavior type of the receiver (instance or class object) is used to obtain the appropriate selector index, as well the proper method dictionary from the current class. Next, the specified selector is searched for in the method dictionary. Even if the selector exists, a test must be made to determine whether the method is private. If it is, the method is only applicable if the class of the receiver is equal to the current class (definition of *private* in MS). If a method is found and is legal, its division is returned. Otherwise, a recursive call must be made to *lookup_dispatch*, with the receiver and selectors remaining the same, but with the current class being changed to one of the superclasses of the current class. If the result of this recursive call is an empty division, it is attempted again on another superclass, until a non-empty division is found, or all super classes have been searched. The resulting division is returned as the result of the algorithm, even if it is empty.

Table 5 contains comparative performance results for the class look-up and coloring dispatch algorithms, based on illustrative code that will not occur in practice. Subclasses of the array class were

created at depths of: 1, 5, 10, 15 and 19. The same array initialization message was sent to instances of the leaf node classes of these inheritance chains and the time was recorded. The array initialization code is given in Appendix A for the subclass at depth 1. In every case, an array size of 50 and iteration size of 5000 were used so that a total of 250,000 dispatches were done. The trials were done on a SPARC ELC and the average of 10 trials was used in each case.

Depth	**class look-up dispatches / sec**	**coloring cache dispatches / sec**	**improvement**
1	19,400	24,500	26%
5	10,400	24,500	135%
10	5,720	24,500	328%
15	3,750	24,500	553%
19	2,980	24,500	722%

Table 5 Method dispatch comparison

6. C-Code Generation.

Each of the node classes representing the parse tree of an MS program knows how to generate C code to execute itself correctly.

6.1 C-Structures

A variety of C data structures are used. A program's execution environment is stored in an MSPEnv structure. The environment includes: an array of modules (type MSModule), an array of literal objects, an array of contexts and two dispatch tables (type MSCacheTable), one for instance messages and one for class messages.

A module is represented by an MSModule structure that contains a context and an integer representing the index of the module within the environment's module array. The cache tables store C function addresses and unique selector indices for each class-color pair that define a method.

The most import C data structure, called an MSObject, is used to represent an MS object. For efficiency reasons, objects are dynamically allocated arrays. In this section, stored behaviors will be referred to as instance variables, where each instance variable is named, indexed or byte-indexed. Each object also stores a flag denoting whether the object is the result of an explicit return from a method, a flag denoting whether the object is immutable or not, the position within the object of the first indexed instance variable and first byte indexed instance variable, and the size of the object. Indexed and byte indexed instance variables are implemented as arrays, whose first element is the MS integer object denoting the size of the variable, and whose successive elements are the values of the variable.

For uniformity, MS class objects are stored as MSObjects. A variety of predefined named instance variables exist for every class. These include the number of named, indexed and byte indexed variables for instances and classes, and the unique integer index representing the class. To compare the class look-up method dispatch algorithm with color-table dispatch, additional tables were temporarily included in each class. The extra tables were hash tables for defined instance and class behaviors and associated visibility tables.

Contexts are treated as MS objects, that are instances of the MSContext class. A context has two named instance variables, one referencing its static link, the other referencing the end of the static link chain (i.e. the context of the module in which the context was created). Static links are used to access non-local variables. This second variable is used to recognize the appropriate time to remove the wrapper from a return object (see Section 3.9). The class also has one indexed instance variable, that represents the values of the context. The dynamic execution context stack is implemented as an array of contexts, so contexts do not need to explicitly store dynamic links.

6.2 Library and Primitive C Functions

There are several C-code library functions and primitive functions. The *execute* library function was described in Section 5.5. The *undefinedMethod* and *abstractMethod* primitives are used to generate errors if an inappropriate selector is used. The *msBasicNew* primitive function has a class object as an argument and is used to create a new instance object and to initialize its state. Named instance variables are initialized to nil and the size of indexed instance variables is set to zero. The *makeClass* library function creates a new class object and its arguments are used to initialize the new class. There are a variety of other less important library functions as well.

6.3 C-code implementation

The translation of an MS program from internal representation to C code relies entirely on macros. There is no explicit reference to any variable, nor are there any syntactic constructs (other than the assignment operator which is easily changed) that tie the code generated to the C language.

The rationale behind the macro abstraction is the ability to use the C preprocessor to implement the code in an arbitrary language by changing the macro expansions. This provides a method of switching from an implementation in C to one in assembly language without having to modify the ST-80 based code generator. This macro abstraction incurs no time-inefficiencies if the C-compiler optimizes simple literal expressions like *1+3* and *2*4*.

A distinction must be made between function addresses and function strings. Function addresses will be referred to by un-delimited names. Function strings will be denoted by C-strings. For example, the function address of the function main() is main, and the function string is "main". This distinction is

crucial in order to understand the argument types expected by the macros. Appendix A includes a sample of the macros generated.

Program Code Generation

A program consists of a main module and all of its recursively imported modules. Each time a program is generated, a new ST-80 compiler environment is created, all required modules are re-parsed, and the resulting minimal cache table and selector coloring is used. Thus, only those modules that are used by a program are included and no references to selectors from unused modules occur in the cache tables.

A function for executing the program and functions for initializing the cache tables and selector-to-color mappings, as well as a function for creating all literals used in the environment are placed in the *program* file. For efficiency, the MSPEnv environment structure is a global variable, declared within the program file. Executing the program consists of first initializing the environment, then executing each module in sequence. This sequence is dictated by the constraint that any module that imports another module must be executed after the imported module.

The MS specification states that literals are immutable. In this implementation, each literal is only created once and stored as part of the environment. The classes that have literal representations are: Integer, Float, Character, MethodSelector, String, and Array. The first three of these do not have any explicit state, so nothing needs to be done to insure their immutability. For the last three classes, literal instances are marked as immutable. Inside the state methods for these classes, an instance is checked to see if it is immutable before changing its state.

Module Code Generation

A module is executed by a function that allocates space for the module's context, then executes each binding specified in the module. Modules exist only within the environment, and the execution of a module results in a context that is inserted into the module array of the environment. The name of the function used to execute a module is uniquely determined by the name of the module.

There are only three kinds of bindings that exist within modules: class declarations, import specifications and module expressions. An import specification copies the value of an index of one module's context to an index within the current module's context. A class declaration or module expression assigns the result of a function call to the appropriate slot in the module's context. The function names are uniquely defined by the binding names. Note that class extensions have not yet been implemented, and explicit module imports are treated as short-hand for implicit importation of every binding within the specified module. All code generated for a module is placed in a single file.

Class Definition Code Generation

The default action of a class defining function is to simply call the *makeClass* library routine with the appropriate arguments. If class look-up method dispatch is used then code to create a hash table for each selector in the instance and class behaviors is generated before the call to *makeClass* is made. This is another source of inefficiency for the class look-up method dispatch approach.

Method Code Generation

Method definitions are common module expressions. Each method definition generates a three argument C-function that is used to execute the method. The function name is uniquely determined by the class name and selector for the method. The first argument is the receiver of the message, the second is an array of arguments to the method and the third is an integer representing the number of arguments.

The code generated for methods depends on the type of method. For *undefined* and *abstract* methods, the name of the C-function is renamed (using the C #define preprocessor command) to the primitive functions *undefinedMethod* and *abstractMethod* respectively. For primitive methods, the function is renamed to the name of the C primitive since all primitives within the ST-80 environment are expected to specify the name of the primitive in both ST-80 and in C. For aliased methods, the code generated depends on the type of the method that is aliased.

The code for state methods depends on the kind. An accessing state method returns the value of the specified variable (or the size, in the case of an indexed size access state method). An assignment state method assigns a new value to the specified variable (or changes the size of the variable in the case of an indexed size assignment state method), and return the new value.

For a block method, a function is generated that executes the code generated by each statement of the method and then returns a result. Before any statements are executed, two local variables are declared. The *result* variable stores the return value of the method. The *subBlockResult* variable stores the results of message sends within the method, and will be described later. A new context is created and pushed onto the dynamic context stack. The first element in the context is self, the next *numArg* elements of this context are initialized with the values of the *args* variable, and the next *numTemp* elements are initialized to the distinguished MS object *nil*. A block method's static link is set to the context of the module that defines it (a literal block's static link is set to its containing block when it is created). After the new context has been initialized and placed on the context stack, code for each statement is generated.

Code Generation for Variables and Self and Literals

All MS variable references (including class references) are mapped to level-offset pairs, where the level is the distance on the static link chain, relative to the current context (the top of the dynamic context stack). Note that most variable references have a level of zero and are in the current context, so no static links need to be followed. Self is treated as the zeroth argument of a block and is stored always stored in

the current context. Literals are created when the program begins execution, so the code generated for a literal is simply a reference to a particular index of the initialized literal array.

Code Generation for Message Sends

An MSMessageSend generates C-code that uses C-subblocks which can be nested arbitrarily deep, with variable definitions allowed within any sub-block. Such a structure is ideal for implementing not nested message sends and imbedded literal block statements (but not literal blocks which are arguments to message sends).

Each message send is delimited by a pair of macros. The message start macro, MESS_SEND, creates a subblock and defines two local variables, *args* and *rec* while the message end macro, MESS_END, simply ends the subblock. Between this macro pair, the argument array *args* is dynamically allocated to be of size *numArgs*, the number of arguments expected by the message. The argument array and the receiver are assigned the appropriate values by generating one assignment line per argument. The left hand side of the assignment is a macro that references one of the arguments or the receiver and the right hand side is the macro needed to represent the object in question. For example, the macros for variables, self and literals have already been described. The other possibilities are: assignment, return and message send. Assignments and returns are described in the next two sub-sections. The macro for a message send generates an entire imbedded MESS_SEND, MESS_END macro specification.

After the assignments to the receiver and arguments, the *execute* function is invoked and the result is assigned to the variable, *subBlockResult*.

Code Generation for Assignments and Returns

Since the left hand side of an assignment is always a variable reference, it will always be represented in C as a context reference. The right hand side will be represented by one of the macros already described. An assignment cannot have an MSCReturn as its expression since the compiler would have generated an error.

An MSReturn, must generate code that immediately returns its contents as the value of the entire method. It generates an assignment of its contents to the *result* variable and a return statement.

6.4 Optimizations

At the time that C-Code generation occurs, the ST-80 MS Environment knows which selectors are uniquely defined. These are the selectors whose divisions are specific or whose receiver is a literal or *self*. In these cases, message sends can be optimized - the *execute* function does not need to be called to obtain the function. Instead, where the *execute* would have occurred, a direct call to the appropriate function can be made. That is, a macro can be generated that pushes the appropriate arguments onto the stack and performs this direct call.

This optimization can be extended in the case of state accessing methods if *self* is the receiver. In this case, no function call is needed at all - the code to access the desired variable can be placed in-line where the message is called.

If static typing is added to the language, it is possible for many selector to be uniquely defined, in which case method look-up would not be needed. These are the selectors that are separate or determined (a special case of redefined in which there are no non-dependents in the division). These results will be discussed in a forthcoming paper.

7. Conclusions

We have presented a "first" implementation of Modular Smalltalk that validates two of the language design goals: consistent execution semantics and efficient code execution We intend to use this implementation to validate the other three design goals: increased programmer productivity through code reuse and code re-definition, design and implementation efficiency for multiple programmer applications and simplicity for new users.

The implementation is an object-oriented one and illustrates the use of an object-oriented approach to: parsing, program representation and code generation. The code generation is modular and based on macros so it can be easily modified to support a variety of target languages including C and assembly language. The efficient implementation includes a cache table approach to method dispatch that uses extensions to the André-Royer incremental coloring algorithm. The new coloring algorithm includes support for optional static typing and we are currently studying the effects of typing and multiple cache tables on dispatch efficiency. Currently, we have only implemented a simple reference-counting storage manager. We intend to incorporate a more efficient scavenging approach [UJ88] [WM89]. Our implementation will be available by anonymous ftp when the documentation is complete.

Acknowledgements

We are indebted to Brent Knight for his preliminary work on the internal representation of MS and to Brian Wilkerson and Daniel Lanovaz for the many comments they made throughout the project.

References

[AR92] P. André and J. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *OOPSLA '92 Conference Proceedings*, pp.110-126, October 1992.

[Coo92] W. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *OOPSLA '92 Conference Proceedings*, pp.1-15, October 1992.

[CU91] C. Chambers and D. Unger. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp.1-15, October 1991.

[DMSV89] R. Dixon et. al. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *OOPSLA '89 Conference Proceedings*, pp.211-214, October 1989.

[GR89] A. Goldberg and D. Robson. *ST-80, The Language*, Addison-Wesley, Reading Mass, 1989.

[Tek89] Tektronix. Modular Smalltalk Language Proposal, Technical Report CRL-89-03, 1989.

[UJ88] D. Unger and F. Jackson. Tenuring Policies for Generation-Based Storage Reclamation. In *OOPSLA '88 Conference Proceedings*, pp.1-17, September 1988.

[WBW88] A. Wirfs-Brock and B. Wilkerson. An Overview of MS. In *OOPSLA '88 Conference Proceedings*, pp.123-134, September 1988.

[WM89] P. Wilson and T. Moher. Design of the Opportunistic Garbage Collector. In *OOPSLA '89 Conference Proceedings*, pp.23-35, October 1989.

Appendix A.

Sample code generated for a program:

```
# include "cgen_lib.h"
# include "test2_all_modules.h"

DECLARE_GLOBALS();

PROGRAM_MAIN()
   PROGRAM_INIT( "test2" );
   EXECUTE_MODULE( KernelModule, 0 );
   EXECUTE_MODULE( test2Module, 1 );
PROGRAM_END();

INIT_DEF()
   INSTCOLORS()
      9  , 8  , 5  , 6  , -1 , -1 , -1 , -1 , 4  , 0  ,
      1  , 2  , 3  , 4  , 0  , 1  , 2  , 3  , 0  , 1  ,
      2  , 3  , 7  , 0  , 1  , 2  , 3  , 0  , 1  , 4
   END_COLORS();

   CLASSCOLORS()
      0
   END_COLORS();

   CLASSNAMES()
      "CreatableObject",        /* 0 */
      "Integer",                /* 1 */
      "Array",                  /* 2 */
      "Character",              /* 3 */
      "Closure",                /* 4 */
      "Float",                  /* 5 */
      "Message",                /* 6 */
      "MethodSelector",         /* 7 */
      "String",                 /* 8 */
      "UndefinedObject",        /* 9 */
      "A",                      /* 10 */
   END_NAMES();
```

```
   INIT_ENV_NAMES();
   INIT_ENV_COLORS();

   TABLE_NEW( INST_TABLE, 11 );
   TABLE_NEWAT( INST_TABLE, 0, 0 );
   TABLE_NEWAT( INST_TABLE, 1, 5 );
   TABLE_ATPUT( INST_TABLE,1,0,9, Integer_binary_K /* + */, "Integer_binary_K   " );
   ...
   TABLE_ATPUT( INST_TABLE, 1, 4, 13, Integer_toK_doK, "Integer_toK_doK" );
   TABLE_NEWAT( INST_TABLE, 2, 4 );
   TABLE_ATPUT( INST_TABLE, 2, 0, 14, Array_size, "Array_size" );
   TABLE_ATPUT( INST_TABLE, 2, 1, 15, Array_sizeK, "Array_sizeK" );
   TABLE_ATPUT( INST_TABLE, 2, 2, 16, Array_atK, "Array_atK" );
   TABLE_ATPUT( INST_TABLE, 2, 3, 17, Array_atK_putK, "Array_atK_putK" );
   ...
   TABLE_NEW( CLASS_TABLE, 11 );
   TABLE_NEWAT( CLASS_TABLE, 0, 1 );
   TABLE_ATPUT( CLASS_TABLE, 0, 0, 0, CreatableObject_class_new,
      "CreatableObject_class_new" );
   ...
   LITERAL_ASSIGN_SPACE( 4 );
   LITERAL_SET_INDEX( 0,
      ASSIGN_NEW_SIMPLE_LITERAL( LITERAL_TMP, GO_CLASS_INTEGER, 0) );
   LITERAL_SET_INDEX( 1,
      ASSIGN_NEW_SIMPLE_LITERAL( LITERAL_TMP, GO_CLASS_INTEGER, 1) );
   LITERAL_SET_INDEX( 2,
      ASSIGN_NEW_SIMPLE_LITERAL( LITERAL_TMP, GO_CLASS_INTEGER, 892) );
   LITERAL_SET_INDEX( 3,
      ASSIGN_NEW_SIMPLE_LITERAL( LITERAL_TMP, GO_CLASS_INTEGER, 298) );
INIT_END();
```

Sample code generated for a module:

MS source code was read from a file and parsed into the environment. The C-code generation also used the decompiler to provide the MS Source code listing that is included in the following generated C-code to document it.

```
/* This file implements the creation of all classes in the
  test2 module, as well as the test2 module itself. */

# include "test2.h"
# include "cgen_lib.h"
/*** A ***/
/*
   class { refines Array }
   instance
   { behavior
      initialize:iterations: -> method
         [:sz :iter |
            self size: sz.
            1 to: iter do:
               [ :i |
                  0 to: sz - 1 do:
                     [ :j |
                        ( self at: j put: j.
                     ].].].
   }
```

```
*/
CLASSDEF_HEADER( AClass )
   CLASSDEF_VARS( "AClass", 1, 0 );
# ifdef LOOKUP_DISPATCH
   CLASSDEF_INIT_HASH( 1, 0, 1 );
   CLASSDEF_ADD_SUPERCLASS( 2 );
   CLASSDEF_SET_INST( 0, 29, A_initializeK_iterationsK, VIS_PUBLIC );
# endif
   CLASSDEF_MAKE( 0, 1, 0, 0, 0, 0, 10 );
CLASSDEF_END();

BLOCKDEF_HEADER( A_initializeK_iterationsK_block_1_1 )
   BLOCKDEF_VARS( "A_initializeK_iterationsK_block_1_1",1,0,1, BLOCK_IS_LITERAL );
   MESS_SEND( 2 );
      MESS_ARG( 0 ) = CNTXT_OFFSET( CC, 0 );   /*  CNTXT ( 0, 0)  */
      MESS_ARG( 1 ) = CNTXT_OFFSET( CC, 0 );   /*  CNTXT ( 0, 0)  */
      MESS_REC() = CNTXT_OFFSET( CN( CN( CC ) ), 0 );   /*  CNTXT ( 2, 0)  */
      SEND_MESSAGE( 2, 17, -1 );   /* at:put: */
   MESS_END( 0 );
   BLOCKDEF_DEFAULTRETURN( MESS_RESULT() );
BLOCKDEF_END();

BLOCKDEF_HEADER( A_initializeK_iterationsK_block_1 )
   BLOCKDEF_VARS( "A_initializeK_iterationsK_block_1",1,0,1, BLOCK_IS_LITERAL );
   MESS_SEND( 2 );
      MESS_SEND( 1 );
         MESS_ARG( 0 ) = LITERAL_AT_INDEX( 1 );     /* 1 */
         MESS_REC() = CNTXT_OFFSET( CN( CC ), 1 );   /*  CNTXT ( 1, 1)  */
         SEND_MESSAGE( 1, 10, -1 );   /* - */
      MESS_END( 0 );
      MESS_ARG( 0 ) = MESS_RESULT();
      ASSIGN_NEW_CLOSURE( MESS_ARG( 1 ), A_initializeK_iterationsK_block_1_1,1,CC );
      MESS_REC() = LITERAL_AT_INDEX( 0 );     /* 0 */
      SEND_MESSAGE( 2, 13, -1 );   /* to:do: */
   MESS_END( 0 );
   BLOCKDEF_DEFAULTRETURN( MESS_RESULT() );
BLOCKDEF_END();

BLOCKDEF_HEADER( A_initializeK_iterationsK )
   BLOCKDEF_VARS( "A_initializeK_iterationsK", 2, 0, 1, BLOCK_IS_METHOD );
   MESS_SEND( 1 );
      MESS_ARG( 0 ) = CNTXT_OFFSET( CC, 1 );   /*  CNTXT ( 0, 1)  */
      MESS_REC() = CNTXT_OFFSET( CC, 0 );   /*  CNTXT ( 0, 0)  */
      SEND_MESSAGE( 1, 15, -1 );   /* size: */
   MESS_END( 1 );
   MESS_SEND( 2 );
      MESS_ARG( 0 ) = CNTXT_OFFSET( CC, 2 );   /*  CNTXT ( 0, 2)  */
      ASSIGN_NEW_CLOSURE( MESS_ARG( 1 ), A_initializeK_iterationsK_block_1,1,CC );
      MESS_REC() = LITERAL_AT_INDEX( 1 );     /* 1 */
      SEND_MESSAGE( 2, 13, -1 );   /* to:do: */
   MESS_END( 1 );
   BLOCKDEF_DEFAULTRETURN( MESS_RESULT() );
BLOCKDEF_END();

/*** test2Module_anA ***/

/* anA -> { expression anA new initialize: 892 iterations: 298 } */
```

```
MODBLOCK_HEADER( test2Module_anA )
   MODBLOCK_VARS( "test2Module_anA" );
   MESS_SEND( 2 );
      MESS_ARG( 0 ) = LITERAL_AT_INDEX( 2 );     /* 892 */
      MESS_ARG( 1 ) = LITERAL_AT_INDEX( 3 );     /* 298 */
      MESS_SEND( 0 );
         MESS_REC() = CNTXT_OFFSET( CC, 1 );   /*  CNTXT ( 0, 1)  */
         SEND_MESSAGE( 0, -1, 0 );   /* new */
      MESS_END( 1 );
      MESS_REC() = MESS_RESULT();
      SEND_MESSAGE( 2, 29, -1 );   /* initialize:iterations: */
   MESS_END( 1 );
   BLOCK_RESULT() = MESS_RESULT();
MODBLOCK_END();

/***  test2Module ***/

MODULE_DEF( test2Module )
   ASSIGN_MODULE_SPACE( 3, "test2Module" );
   ASSIGN_MODULE_IMPORT( 0, 0, 2 );
   ASSIGN_MODULE_CLASS( 1, AClass );
   ASSIGN_MODULE_EXPRESSION( 2, test2Module_anA );
MODULE_END();
```

Appendix B.

Method Dispatch Algorithms

The following definitions extend the definitions given in the ICP algorithm discussion:

```
executeMethod(D)              return the result of execute the method for division D
divisionAt[T,C,L ]            entry in table T at class index of C and color index L
isClass(O)                    object O is a class.
divisionEmpty(D)              [not division isKindOf: MSEmptyDivision]
classMethods(C)               hash table of all class methods (as divisions)
instanceMethods(C)            hash table of all instance methods (as divisions)
selectorPrivateInClass(C,S)   [if selector S is private in class C, true else false]
maxColorStoredForClass(C,T )  maximum color index stored for class C in table T.
colorStoredForClass(L,C,T )   [ L <= maxColorStoredForClass( C, T ) ]
```

Method Dispatch Algorithm For Modular Smalltalk using Two Cache Tables

```
execute(receiver : Object; args : Object; size : Integer; instSelector,
   classSelector : Selector) : Object

begin
   if isClass( receiver ) then
      set S = classSelector
      set T = class table
   else
      set S = instSelector
      set T = inst table
   endif

   set D = divisionAt[ T, class(receiver), color(S) ]

   if ((colorStoredForClass(color(S), class(receiver), T) and
```

```
      (divisionSelector(D) == S)) then
      set result = executeMethod( D )
   else
      generate messageNotUnderstood error.
   endif
   return result
end
```

Class Look-up Dispatch Algorithm for Multiple Inheritance

```
execute( receiver : Object; args : Object; size : Integer )
    instSelector, classSelector : Selector ) : Object

begin
   set D = lookup_dispatch(receiver, class(receiver), instSelector, classSelector)

   if (not divisionEmpty(D))
      set result = executeMethod(D)
   else
      generate messageNotUnderstood error.
   endif
   return result
end
lookup_dispatch(receiver: Object; class: Object; instSelector, classSelector :
   Selector) : Division

begin
    if isClass(receiver) then
       set selector = classSelector
       set methodDictionary = classMethods(class)
    else
       set selector = instSelector
       set methodDictionary = instanceMethods(class)
    endif

    find bucket K for selector S in methodDictionary hash.
    if K includes S,
        set D = division stored for S in K.
        if (selectorPrivateInClass(class(receiver), S)) then
           if (class(receiver) != class) then
              set D = emptyDivision
           endif
        endif
    else
       set D = emtpyDivision
    endif

    if (divisionEmpty(D)) then
       loop over superclasses(C)
          set D = lookup_dispatch(receiver, newClass)
       until ((not divisionEmpty(D)))
    endif

    return D
end
```