QUICKTALK:
A Smalltalk-80 Dialect for Defining Primitive Methods

Mark B. Ballard
M.S.E, Wang Institute of Graduate Studies, 1983

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

April, 1986

The thesis "QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods" by Mark B. Ballard has been examined and approved by the following Examination Committee:

David Maier
Associate Professor
Department of Computer Science and Engineering
Thesis Research Advisor

Allen Wirfs-Brock
Computer Research Laboratory
Tektronix, Inc.

Richard B. Kieburtz
Professor and Chairman
Department of Computer Science and Engineering

Richard Hamlet
Professor
Department of Computer Science and Engineering

DEDICATION

To Mopsy, my wife and best friend. Without her support and encouragement, this thesis would not have been possible.

ACKNOWLEDGEMENTS

The author wishes to thank the Computer Research Labs (CRL) of Tektronix, Inc. for funding the initial portion of this research and supplying the 4404 Smalltalk workstation. Allen Wirfs-Brock from CRL suggested the task of supporting user-defined primitives and has actively participated in the formulation of this proposed solution. Pat Caudill from CRL helped explain the existing Smalltalk interpreter and wrote the routine which connects the Smalltalk interpreter to a user-defined primitive. Dr. Richard Kieburtz encouraged and developed my understanding of programming language design issues and theories of types. Dr. David Maier, my thesis advisor, directed the conclusion of this research, coined the name "QUICKTALK" for the resulting dialect, and offered many constructive suggestions which greatly improved the quality of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

QUICKTALK: A Smalltalk-80 Dialect for Defining
Primitive Methods

Mark B. Ballard, M.S.
Oregon Graduate Center, 1986

Supervising Professor: David Maier

QUICKTALK is a dialect of Smalltalk-80 that can be compiled directly into native machine code, instead of virtual machine bytecodes. The dialect includes "hints" on the class of method arguments, instance variables, and class variables. The dialect is designed to describe primitive Smalltalk methods. Improved performance over bytecodes is achieved by eliminating the interpreter loop on bytecode execution, by reducing the number of message send/returns via binding some target methods at compilation, and by eliminating redundant class checking. Changes to the Smalltalk-80 system and compiler to support the dialect are identified and performance measurements are given.

1. INTRODUCTION

Some problems require experimentation or prototyping to discover an acceptable programmed solution. One such classic problem is user-interface design since a person's behavior in using the interface is so difficult to predict. Programming languages and programming environments have varying degrees of flexibility to support prototyping. Some languages support prototyping better than others via requiring less specification by the programmer. FORTRAN, at one extreme, requires sufficient programmer specification so that everything can be bound at compile time, including all storage allocation. Pascal binds all procedure calls and all the types of variables but does provide for the dynamic allocation of data items. Lisp, at the other extreme, has no compile-time typing or binding of procedures. Its flexibility even allows a program to construct a function and evaluate it during execution.

Perlis [AbS85,Forward] described the spectrum of flexibility in languages as: "Pascal is for building pyramids--imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms--imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place."

Smalltalk, in the spirit of Lisp, binds procedure names to procedure implementations during execution. Unlike Lisp however, values are given abstract types rather than just representation types and thus allow the interpreter to catch inappropriate function applications at the abstract type level. With delayed binding of procedures in Smalltalk, a programmer can change one part of the application program without recompiling the whole program. Smalltalk encourages a programmer to concentrate on the behavior of objects rather than structure. Specialized behavior and increased structure can be factored incrementally with subclassing. As the application matures, the need for flexibility decreases. The programmer can specify his task more precisely and would be willing to trade flexibility for efficiency. The programmer may want to state types of variables and move some procedure bindings and type checking to compile time in order to get a faster execution of the application.

The idea of QUICKTALK is to allow the Smalltalk programmer a way to gain efficiency in a mature application by typing variables in frequently used procedures. QUICKTALK is a dialect of Smalltalk that can be compiled directly into native machine code, instead of virtual machine bytecodes. The dialect includes declarations of the classes of method arguments, instance variables, and class variables. Improved performance is achieved by eliminating the interpreter loop on bytecode execution, by reducing the number of message send/returns via binding some target methods at compilation, and by eliminating redundant class checking.

Section 2 describes the Smalltalk-80 language, the interpreter, and the virtual machine. Section 3 identifies the performance bottlenecks and proposes the QUICKTALK solution. Section 4 acknowledges related work. Section 5 defines the

QUICKTALK dialect. Section 6 describes the design of the current QUICKTALK compiler and Section 7 reports resulting speed optimizations. The thesis concludes with a discussion of the limitations of the QUICKTALK approach and suggests some extensions.

2. SMALLTALK AND PRIMITIVES

The Smalltalk-80[1] system is a programming language plus a graphical, interactive programming environment. The programming language offers attractive features for experimental programming. A feature important in this thesis is the way names get bound to procedures.

Entities in the Smalltalk-80 system are called *objects*. An object has some private memory called *instance variables* and a set of operations called *methods*, that are written in the Smalltalk language. Instance variables can be named or indexed. A named instance variable is accessed by its name while indexed instance variables are accessed by an integer index. A *message* is a request for an object to carry out one of its operations. The *receiver* of the message determines how to carry out the operation the message selects, by performing a method it associates with that message. A *class* describes the implementation of a set of objects that all represent the same kind of entity. An individual object described by a class is called an *instance* of the class. Each class has a *method dictionary* where the methods defined for its instances are stored. Classes are arranged in a hierarchy. Methods defined for

[1]Smalltalk-80 is a trademark of Xerox Corporation. The description of Smalltalk-80 in this introduction models the description in [GoR83, Chapter 1 and Chapter 26].

instances of a class are inherited by instances of its subclasses.

Figure 1 contains the Smalltalk source for an example method, with message selector **max:**, for finding the maximum of two objects that can be compared by the method denoted by the message selector >. This method could be invoked by the expression

    3.5 **max:** 2

which request the greater of 3.5 and 2 be returned. The method is contained in the class Magnitude and is inherited by all subclasses of Magnitude, including Float and SmallInteger. The method has one argument, named *aMagnitude* and one temporary variable, named *returnValue*.

Methods compute by sending messages to other objects. The pseudo-variable *self* refers to the receiver of the message **max:**. The expression *self* > *aMagnitude* says send the message > to yourself with the argument *aMagnitude*. The message selector > is an example of a *binary message selector*, a selector composed of one or two nonalphanumeric characters that takes a single argument. The selector **max:** is an example of a *keyword message selector*, that is used for a method with one or more arguments. The keyword message selector **ifTrue:ifFalse** has *block expressions* for arguments. Block expressions describe objects that represent deferred activities. The expression ↑ *returnValue* says return the value *returnValue* to the sending method.

The Smalltalk-80 system is specified by a stack-oriented virtual machine. [GoR83] Source methods are translated by the system compiler into compiled methods, that contain sequences of eight-bit instructions, called *bytecodes*, for the virtual machine. Figure 2 has a textual representation of the bytecodes for the method of Figure 1.[2]

The Smalltalk interpreter executes bytecodes. When called on to execute a compiled method, it creates a *method context*[3] that refers to an environment that includes the compiled method being executed, an instruction pointer into the compiled method, the receiver and arguments of the message that the sender used to invoke the compiled method, temporary variables needed by the compiled method, and an evaluation stack. The interpreter cycles through the following steps:

(1) Fetch the next bytecode from the compiled method,

(2) Increment the instruction pointer.

(3) Decode the bytecode.

(4) Perform the function of the bytecode.

The interpretation of most bytecodes involves the interpreter's stack. Bytecodes can be grouped into those that push objects onto the evaluation stack, store (and sometimes pop) objects from the stack, send messages, return from a method, or jump to a bytecode within a method.

[2]The reader might be confused by the bytecodes generated for the **ifTrue:ifFalse:** message As written, the message should be sent to the boolean value resulting from the > comparison with block arguments The boolean value would then choose which argument to evaluate The compiler instead has implemented with jump bytecodes the selected evaluation

[3]A method context can be thought of as an activation record

```
Class: Magnitude


max: aMagnitude
        "Answer the receiver or the argument,
whichever has the greater magnitude."

        | returnValue |

        self > aMagnitude
                ifTrue: [returnValue ← self]
                ifFalse: [returnValue ← aMagnitude].
        ↑ returnValue
```

Figure 1: A Smalltalk Source Method

pc	bytecode	method statement
1	push: self	
2	push: aMagnitude	
3	send: >	self > aMagnitude
4	jumpFalse: 8	
5	push: self	ifTrue: [returnValue ← self]
6	storeInto: returnValue	
7	jumpTo: 10	
8	push: aMagnitude	ifFalse: [returnValue ← aMagnitude]
9	storeInto: returnValue	
10	pop	
11	push: returnValue	
12	returnTop	↑ returnValue

Figure 2: Bytecodes for Source Method **max:**

The interpreter usually responds to a send bytecode, sometimes called a *message send*, by interpreting a compiled method associated with the message name. The class of the receiver determines which compiled method is indicated by the message selector of the send bytecode. When the selected compiled method begins to execute, the receiver of the message send and its arguments are on top of the evaluation stack of the sending method. Upon completion of the compiled method, the value returned replaces the receiver and arguments on the evaluation stack.

The send bytecode causes a significant change to the state of the interpreter. The sending method places the receiver and arguments on the interpreter's stack, then requests a message send. The state of the sending method is remembered in the method context so that the sending method may be resumed upon return from the send. The method indicated by the message selector of the send must be found by the following steps:

(1) Find the receiver on the stack.

(2) Look up the message selector in the method dictionary of the class of the message receiver.

(3) If found, save the state of the sending method in the method context and interpret the first bytecode of the compiled method associated with the message selector. If not found, then search the method dictionary of the superclass of the class just searched until found or report an error.

A method can be suspended between any two bytecodes; that is, between any two instructions of the virtual machine. A frequent source of suspension is the unsuccessful search for a method to correspond with a message selector in the attempt to interpret a send bytecode. In this case, an error is reported, and the execution of the compiled method containing the errant send bytecode is suspended.

In addition to Smalltalk source code, some methods, called primitive-calling (PC methods) invoke a primitive routine[4] in native machine code. Primitive routines give Smalltalk the ability to create objects and evaluate expressions. They provide access to and operations upon some virtual machine structures. They are used also to optimize some critical methods that would run too slowly if written in Smalltalk.

A system primitive-calling method (SPC method) has a *primitive section* and a *failure section*. The primitive section simply references a system-supplied primitive routine by number. The interpreter has a table for a maximum of 256 of those routines. The failure section consists of regular Smalltalk code to be performed if the primitive fails. A compiled SPC method compiles its failure section to a regular compiled method, except that a reference number to a system-supplied primitive routine is included. Figure 3 shows the SPC method for the message selector +, that references a system-supplied primitive routine number 1.

A primitive routine fails when it is called with arguments that it was not designed to handle. The most common failure of a primitive occurs when it is called with an argument of the wrong class. In some cases, the value of the argument cannot be handled. The failure section handles these exceptional cases.

[4]Primitive routines are described in [GoR83, Chapter 28].

```
Class: SmallInteger

+ aNumber
        "Add the receiver to aNumber and answer the result
if it is a SmallInteger. Otherwise fail the primitive and
try the superclass method."

        <primitive: 1>

        ↑ super + aNumber
```

Figure 3: A System Primitive-Calling Method

A send bytecode that invokes a compiled SPC method is interpreted by first trying the primitive routine. The primitive routine finds its receiver and arguments on the top of the interpreter's evaluation stack. If the primitive routine completes successfully, the primitive routine replaces the receiver and arguments on the interpreter's stack by the result of the routine. Control passes to the bytecode after the send. If the primitive routine fails, control returns to the interpreter, which interprets the bytecodes for the failure section of the compiled SPC method. The failure section must execute in an environment as if the primitive routine was not attempted. Thus, a primitive must not create side effects until it has determined that its preconditions for successful completion have been met. In Figure 3, primitive routine number 1 attempts to add two SmallIntegers that it finds on top of the interpreter's evaluation stack. Should it not find two SmallIntegers, or the result is not a SmallInteger, then primitive routine number 1 fails, leaving the receiver and argument unmodified. The interpreter then executes the bytecodes of the failure section, that is, the expression ↑ *super* + *aNumber*. The pseudo-variable *super* instructs the interpreter to begin the search for the method to associate with the + message selector in superclass of SmallInteger.

Primitives exist in Smalltalk to perform the six types of operations listed in Figure 4. Arithmetic primitives allow the interpreter to take advantage of its knowledge of number representation and the ALU of the native machine to implement arithmetic much more efficiently than would be possible in the virtual machine. The array-accessing primitives allow access to indexed instance variables. The storage-management primitives use knowledge of the representation of objects to allow manipulating object references, accessing instance variables of objects, creating new instances of a class, and enumerating instances of a class. They also know the representation of the compiled methods. Control primitives provide support for the behavior of processes and semaphores, as well as messages that can take a selector as a parameter. The Input/Output primitives provide access to the state of hardware

(1) Arithmetic, Comparison, and Bitwise
(2) Array and Stream Accessing
(3) Storage Management
(4) Control
(5) Input-Output
(6) Optimization of Critical Loops

Figure 4: Types of Primitive Operations

devices. The optimization primitives include vector drawing and other graphics operations as well as string comparisons and moves. Primitives were chosen to define operations in terms of the hardware in the native machine and to increase efficiency.

Smalltalk programmers would like to write their own primitive methods to express operations in the categories above to improve the performance of their applications. They would like to write these primitive methods without having to know details of the virtual machine interpreter, such as the meaning of values in the registers and special memory locations, or of the native machine code. The QUICKTALK compiler supplies a tool for them to do so. With the QUICKTALK compiler comes the ability to compile critical sections of a Smalltalk application to native code so that they will run much more efficiently than if interpreted by the virtual machine. Users can write their own user PC methods whose primitive section is written in QUICKTALK, rather than invoking one of a fixed set of system-supplied primitive routines.

3. PERFORMANCE BOTTLENECKS AND THE QUICKTALK APPROACH

The following three assumptions [Hag83] about Smalltalk methods and the Smalltalk interpreter motivate our expectations of performance improvements by compiling user primitive-calling (UPC) methods. First, the overhead for delayed binding of messages to methods is high since each procedure call requires an associative lookup in a dictionary of methods (or possibly a hierarchy of dictionaries). Second, each bytecode of the virtual machine must be decoded by the interpreter. And third, every primitive operation must check the types of its arguments.

A dynamic trace of the system would show that many methods send messages to only the existing compiled primitive calling (PC) methods, and none to regular compiled methods. It is the leaves of the method-calling tree that are most frequently interpreted. A large portion of methods have arguments and results of the same class for nearly every call of the method. Thus, many methods do not need the flexibility of delayed binding and could have their message selectors bound to methods during compilation.

QUICKTALK is designed to handle these methods that send messages to only the existing compiled PC methods and whose arguments and results are from the same class for nearly every call. A primitive section for a UPC method can be written in QUICKTALK, which is a Smalltalk subset with types added. By providing types, the QUICKTALK compiler can eliminate the dynamic lookup for methods used within the primitive section. The compiler can find the correct methods once, thus saving the method search during execution. In addition, the type information makes many class checks unnecessary. For short methods[5] used within the UPC method, the QUICKTALK compiler trades space for time by copying the code of the method rather than calling the method.

The QUICKTALK dialect adds type declarations for method arguments as well as instance variable and class variables used in the method.[6] It restricts the use of block expressions to a set of control structures. The selectors that can be used in QUICKTALK also are restricted.

Smalltalk methods that are candidates for compilation, that is, are efficiency problems, can be identified by obtaining execution profiles of existing Smalltalk applications. The method for finding a substring within a string is a good example of a Smalltalk method subject to optimization by rewriting it to be a UPC method. This example is further examined in the chapter titled "Experimental Results."

The problems of adding statically typed UPC methods to Smalltalk without violating the dynamic type security already provided are many. First, the user primitive routine can be called from an untyped environment. Therefore, the routine must check that it is called with arguments of the right type. For structured objects (such

[5]All methods in the current implementation of QUICKTALK copy inline

[6]The current implementation does not handle class variables

as arrays), only those components actually used in the method should be type-checked. For example, consider a UPC method that expects an array of integers and is looking for the index of the first element equal to zero. The UPC method should not care that a non-integer element might occur after the zero.

Second, types have an abstract component, that is, the operations allowed on them, and a representation component. For example, the string type in Smalltalk provides the message **at:** to access a component character by position number. A string is not actually represented as an array of character objects, but as an array of bytes. So the **at:** message first fetches the correct byte and then returns the character object that corresponds to it. Some UPC methods might be able to ignore the character objects and operate directly on the byte representation. Thus QUICKTALK must provide a way for a UPC methods to declare its intention to operate on the representation of an object. Declaring that a particular string object should be treated as an array of bytes, for example, would alter the meaning of **at:** to return the byte.

Third, QUICKTALK type declarations are meant to be "hints" or "expectations". The primitive section of the UPC method is meant to handle a majority of its invocations, while providing a failure section for infrequent invocations with arguments of the wrong type. A failed primitive should be side-effect free. Simple type checking (a structural test) might not guarantee the successful completion of a primitive, for example, SmallInteger overflow. Having QUICKTALK guarantee an undo facility seems too expensive, so the responsibility for restoring state if changes are made rests with the programmer.

Fourth, one must decide what to type. In QUICKTALK, types are associated with arguments to a method and variables used within the method rather than typing the instance variables of a class. Restricting a method to operate on objects of a specified type seemed to be a better way to localize and isolate the constraints imposed by types on a Smalltalk application. Concentrating on typing frequently used methods promised a great increase in efficiency with a minimum of constraint. Consistent with typing methods rather than the instance variables of a class, the object-accessing selectors are typed.

Fifth, most types are equivalent to Smalltalk classes. For reasons of efficient type checking, instances of a subclass are not considered to be of the same type as instances of its parent class.

Sixth, block expressions are not considered values in QUICKTALK and are thus not typed. The complexity introduced by treating functions as values does not seem justified for a language intended to write primitives.

QUICKTALK is designed for writing primitive routines that cannot be suspended. Therefore, the interpreter of a QUICKTALK method need not provide a mapping from its execution environment to that defined by the Smalltalk virtual machine.

Although the focus of this research was on adding types to Smalltalk, a major performance advantage of compiling user-defined primitive methods is the elimination of the interpreter loop on bytecode execution. In the Tektronix Smalltalk interpreter, for example, decoding and dispatching a bytecode takes a minimum of five machine instructions, or between 3-4 microseconds. [Wir85]

4. RELATED WORK

Work related to QUICKTALK can be divided into three areas: adding *optional* typing to Smalltalk, compiling Smalltalk, and improving the performance of interpreted Smalltalk. The goals of proposals for adding typing to Smalltalk include improving program readability and documentation as well as improving code efficiency.

4.1. Typing Smalltalk

Borning and Ingalls [BoI81] concentrate on adding a type system to Smalltalk to support compile-time checking and thus adding machine-checkable documentation to programs. They think of types as abstracting classes. Like classes, types specify the messages that an object of that type understands and come in hierarchies. Unlike classes, types can have parameters; e.g. "Collection of: X" where X can be any type. In their proposal, they add to the Smalltalk language explicit type declarations to method arguments and returned values. They mention a need to type instance, class, and global variables but show no examples. The compiler infers the types of temporary variables. They use the explicit declarations to check that messages within the method have acceptable arguments, that only objects of the correct type will be assigned to variables, and that an object of the correct type will be returned.

assigned to variables, and that an object of the correct type will be returned.

Suzuki [Suz81] infers types in the absence of declarations. His types are unions of Smalltalk classes. Types are associated with variables; methods map a Cartesian product of types to types. He wanted to design tools to supply type declaration to current Smalltalk programs. He does not attempt to handle types with parameters.

Suzuki and Terada [SuT84] decided that many type inferences were not tight enough to allow efficient code generation. They introduce type expressions for variables, method arguments, and blocks that will allow them to bind some messages to methods at compilation. They allow union types, which means some messages require a case selection of methods based on the class of the receiver. They do not handle types with parameters.

4.2. Compiling Smalltalk

Hagmann [Hag83], adds a class declaration to method arguments; the class that is expected in the vast majority of method activations. Thus, his types are "hints". For methods where preferred classes are declared, he produces two compiled methods; the standard compiled method and a machine code version. If the machine code version should return a value that does not match the preferred class, then the execution must be continued in the corresponding position in the standard compiled method. Thus, he must be able to support mappings between the native code environment (very different from the virtual machine) and the virtual machine environment. He also attempts to compile any Smalltalk method, not just Smalltalk methods at the leaves of the message-send tree. He must deal therefore with the possibility that his methods can be interrupted and suspended. Mappings between the machine code version and the standard compiled method must be supported for the Smalltalk debugger to work properly.

Larus and Bush [LaB83] propose applying source-to-source transformations on non-polymorphic Smalltalk methods. They require class declarations for variables and libraries of method rewrites. If the class of a receiver of a message is known, then the method associated is known and can be substituted. Their major performance improvement comes from telescoping the message send tree by substituting inline for the message sends. In addition, the operations within a method rewrite might be able to safely forego some type checking and array bounds checking.

4.3. Improving Smalltalk Performance

Deutsch [DeS84] suggested many techniques for improving the efficiency of interpreting Smalltalk. First, he discovered that 95% of all sends, as measured from each point of sending, execute the same method as the previous send from that point. Therefore, Deutsch proposed inline caching of the last method lookup for each send bytecode to reduce this overhead. The cached method must check the class of the receiver to see if it applies. Second, he allocated method contexts (activation records) on a linear stack, only promoting them to standard Smalltalk objects when necessary. This allocation strategy decreased reference counting and garbage collection. These two ideas made the Smalltalk interpreter more efficient instead of improving the code.

To improve the code's efficiency, he suggested that the bytecodes could be dynamically expanded (similar to macro expansion) into their equivalent native code and optimized in native code. Using this technique for arbitrary Smalltalk means he, like Hagmann, must support mappings between the native code and the bytecodes.

5. QUICKTALK LANGUAGE DEFINITION

This section describes the QUICKTALK dialect as it differs from Smalltalk-80. The subsections introduce the user primitive-calling (UPC) method format, the typing discipline, the control structures, and the message selectors that are permitted in the dialect. The section ends with a discussion of side effects in UPC methods, UPC methods as subroutines, and a methodology for using QUICKTALK.

5.1. UPC Method Format

A UPC method follows the structure of a SPC method, that is, a single message selector followed by a primitive section and a failure section. The primitive section is delimited by set braces. See Figure 5 for a BNF description. See Figure 6 for an example UPC method with the sections annotated.

```
<UPC method> ::= <message selector> <user-primitive section> <failure section>.
<user-primitive section> ::= '{' <temporaries> <QUICKTALK statements> '}'.
<failure section> ::= <temporaries> <Smalltalk statements>.
```

Figure 5: BNF for a UPC Method

```
example: arg1 and: arg2          "message selector"
{
        | upcTemp |            "primitive section temporary"
        arg1 declare: SmallInteger.   "type declaration"
        arg2 declare: SmallInteger.                          primitive section
        upcTemp ← arg1 < arg2.
        ↑ upcTemp
}

        | failureTemp |        "failure section temporary"
        failureTemp ← arg1 < arg2.                       failure section
        ↑ failureTemp
```

Figure 6: Example UPC Method

In Figure 6, notice that the type declaration statements (identified in the next section) appear among the QUICKTALK statements within the user-primitive section. Also, notice that the primitive section and the failure section each has its own set of temporary variables.

5.2. Typing

The previous chapter reviewed various efforts to add a typing discipline to Smalltalk. In QUICKTALK, types are used to discriminate which instance of a polymorphic operator applies. For example, a + can mean either of the following two operators:

+ $\quad$ SmallInteger $\times$ SmallInteger $\rightarrow$ SmallInteger
+ $\quad$ Float $\times$ Float $\rightarrow$ Float

The notation $\alpha \times \beta \rightarrow \xi$ means the operator takes two arguments, the first of type α, the second of type β, and returns a value of type ξ. All method arguments and class variables used in the primitive section of a UPC method must have declared types. In addition, the value returned by object-accessing selectors must be typed. Typing the value returned by object-accessing selectors is the way that instance variables of objects are typed. The types of temporary variables are inferred when assigned the value of an expression that can be typed. In the current implementation of QUICK-TALK, the assignment to a temporary variable must be made textually before it is used in another expression and each temporary can have only a single type.

Types of identifiers (method arguments and class variables), and object-accessing selectors are declared by the messages in Figure 7.[7] The figure shows the syntax of the messages. The declaration messages may appear as statements anywhere in the user-primitive section. In the current implementation, identifiers and

<ident> **declare:** <class>
<ident> **declareInternal:** <representation>
<ident> **declareArrayOf:** <class>

<symbol> **declareAccess:** <class> **inClass:** <class> **forFieldNamed:** <ident>
<symbol> **declareAccess:** <class> **inClass:** <class>
<symbol> **declareUpdateInClass:** <class> **forFieldNamed:** <ident>
<symbol> **declareUpdateInClass:** <class>

Figure 7: Type-declaring and Object-accessing Selectors

[7]The syntax of the **declareAccess:** messages is different in the current implementation. Also, the **declareUpdateInClass:** messages have not been implemented.

object-accessing selectors must be declared before they are used. The message **declare:** declares an identifier that will denote only objects of the given class. Subclasses of the given class are excluded by the declaration. For example,

```
x declare: Point.
```

declares the identifier *x* will denote an object of class Point. An exception to excluding subclasses with this declaration occurs when the declared class is Object. This exception allows a method to accept an arbitrary object when its type is not needed by any operations, for example, when only the object's identity or size is used by the method. The class Object is assumed for all undeclared variables.

The message **declareInternal:** declares an intent to treat the object denoted by the identifier in terms of its internal representation rather than its external interface when interpreting messages sent to the object. Externally, Smalltalk objects appear to be indivisible units (SmallInteger, Character, Boolean) or composed of references to other objects. Internally, however, some objects that look externally like an Array of Characters or an Array of Boolean are represented internally as lists of numbers or bit strings rather than lists of references. The writer of a primitive may need to exploit the internal representation of objects for efficiency. For example,

```
y declareInternal: ByteArray.
```

declares the identifier *y* will denote an object which is represented as a ByteArray. Note that this declaration is subtly different from

```
y declare: ByteArray
```

since the second declaration declares *y* to be an object of exactly the class ByteArray and not, for example, a String, which is a subclass. The message **at:** applied to a String means to return a Character object. When applied to a ByteArray object, it means return a SmallInteger between 0-255.

The message **declareArrayOf:** is used to declare that an identifier denotes an Array whose elements are of a single class. Note that the syntax allows one to declare an Array whose elements are Array's, but the elements of the second Array cannot be typed. For example,

```
a declareArrayOf: SmallInteger
```

declares the identifier *a* to be an Array of SmallInteger elements.

The messages **declareAccess:inClass:** and **declareAccess:inClass:forFieldNamed:** are used to type an object's instance variables by typing the value returned by an instance variable selector. For example,

```
#origin declareAccess: Point inClass: Rectangle forFieldNamed: origin.
```

declares that the message **origin** returns the instance variable named *origin* of type Point when applied to any object of the class Rectangle. When the message name is the same as that of the instance variable name, the above declaration can be abbreviated to:

```
#origin declareAccess: Point inClass: Rectangle
```

The messages **declareUpdateInClass:** and **declareUpdateInClass:forFieldNamed:** are used to identify a selector used to update an instance variable of a class. For example,

#origin: declareUpdateInClass: Rectangle forFieldNamed: origin

declares that the message **origin:** can be used to assign a value to the instance variable named *origin* for any object in the class Rectangle. Again, if the field name is not specified, it defaults to the name of the selector.

The various declarations determine the way types are checked. All identifiers declared to be of a particular class are checked upon entry to see if they match the declaration. Identifiers declared to have an internal representation or to be an Array of elements are also checked upon entry. Elements of Array's are checked upon extraction by the message **basicAt:**, so elements not extracted will never be checked. Object-accessing selectors declared with the message **declareAccess:inClass:forFieldNamed:** invoke methods which check the type of the value they return. QUICKTALK selectors (defined in the next section) invoke methods that do not check the types of their arguments but must check the type of the value returned.

Of the Smalltalk pseudo-variables, only *self, nil, true, false*, are allowed in QUICKTALK user primitive sections. The type of *self* is assumed to be the same as the class containing the method definition unless it is declared otherwise. The pseudo-variable *nil* is given the type UndefinedObject.[8] The pseudo-variables *true* and *false* are given the type Boolean. The pseudo-variables *super* and *thisContext* are not allowed in QUICKTALK user-primitive sections.

[8]The use of *nil* in QUICKTALK user-primitive sections has not been implemented

5.3. Blocks

A block expressions in Smalltalk describes an object representing a deferred sequence of actions. The sequence of actions actually takes place when the block receives the message **value**. Blocks are most often used to implement nonsequential control structures. Blocks also are used as an iterator over Collections or Arrays as in CLU [LSA77], to express actions under exceptional conditions, and as a simple way to pass a function as an argument to another method. The class SortedCollection uses a Block to store a function that can determine the order of any pair of the Collection's elements.

A QUICKTALK method may use blocks only with the selectors identified in Figure 8. These blocks and selectors supply the Smalltalk programmer with the standard conditional and looping control structures. The occurrence of block expressions is severely constrained in QUICKTALK. In particular, blocks cannot be assigned to

```
<Bool> ifTrue: <Block>
<Bool> ifFalse: <Block>
<Bool> ifTrue: <Block>  ifFalse: <Block>
<Bool> ifFalse: <Block>  ifTrue: <Block>
<Bool> and: <Block>
<Bool> or: <Block>

<Block> whileTrue: <Block>
<Block> whileFalse: <Block>

<SmallInteger> to: <SmallInteger> do: <BlockWithOneArgument>
```

Figure 8: Primitive Blocks

variables, cannot be evaluated explicitly, and cannot be returned by UPC methods, since the simple typing discipline does not support function types. Typed blocks are not essential for primitive methods and would make QUICKTALK significantly more complex.

5.4. Selectors

Figures 9, 10, and 11 (and Figure 7 on type-declaring and object-accessing selectors) contain the set of all selectors that can be used in UPC methods.[9] The Greek letters in the figures are type variables. Thus,

basicAt: (Array of: α) $\times$ SmallInteger $\rightarrow \alpha$

means that the selector **basicAt:** can be applied with a SmallInteger argument to an Array of objects of any type α and will return an object of type α. These *typed* selectors are the only selectors that QUICKTALK allows.

Figure 11 list messages that are novel in QUICKTALK in addition to those in Figure 7. The selectors **failIfTrue** and **failIfFalse** allow a UPC method to fail after computing an arbitrary predicate.

5.5. Side Effects

A UPC method must determine that its preconditions for success have been met before it can update arguments or global objects. Upon failure, the failure section of the primitive calling methods must execute in an environment as if the primitive had not been tried. Responsibility for insuring that the primitive leaves its environment

[9]Floating point selectors have not been implemented.

+	SmallInteger $\times$ SmallInteger $\rightarrow$ SmallInteger
+	Float $\times$ Float $\rightarrow$ Float
	... similarly for -, *, /
<	SmallInteger $\times$ SmallInteger $\rightarrow$ Boolean
<	Float $\times$ Float $\rightarrow$ Boolean
<	Character $\times$ Character $\rightarrow$ Boolean
	... similarly for >, <=, >=
=	Float $\times$ Float $\rightarrow$ Boolean
=	$\alpha \times \alpha \rightarrow$ Boolean (interpreted as identity except Float)
	... similarly for ~=
bitShift:	SmallInteger $\times$ SmallInteger $\rightarrow$ SmallInteger
	... similarly for **bitAnd:**, **bitOr:**, //, \\\\

Figure 9: Compiler-Known Selectors — Arithmetic selectors

@	SmallInteger $\times$ SmallInteger $\rightarrow$ Point
@	Float $\times$ Float $\rightarrow$ Point
basicAt:	ByteArray $\times$ SmallInteger $\rightarrow$ SmallInteger
basicAt:	(Array of: α) $\times$ SmallInteger $\rightarrow \alpha$
basicAt:put:	ByteArray $\times$ SmallInteger $\times$ SmallInteger $\rightarrow$ SmallInteger
basicAt:put:	(Array of: α) $\times$ SmallInteger $\times \alpha \rightarrow \alpha$
basicSize	$\alpha \rightarrow$ SmallInteger
==	$\alpha \times \alpha \rightarrow$ Boolean

Figure 10: Compiler-Known Selectors — Non-Arithmetic

failIfFalse	Boolean → (causes control change)
failIfTrue	Boolean → (causes control change)

Figure 11: Compiler-Known Selectors — Additional

unchanged upon failure rests solely with the programmer. QUICKTALK provides no support for recovering from exceptions that depend on the values of types when the primitive has already performed some operations with side effects.

For example, suppose a QUICKTALK UPC method expects an Array of SmallInteger, where it will double each element. It proceeds to replace each element by its double until it finds a Float that it did not expect and fails. The failure section can observe that the primitive section was attempted thus violating the Smalltalk definition of a primitive.

5.6. User Primitive-Calling Methods As Subroutines

This first design of QUICKTALK does not allow UPC methods to invoke other UPC methods. Allowing this important capability must be postponed to later work, although there does not seem to be any fundamental problem. At that time, primitive routines will need the concept of an activation record and there might be a different argument-passing mechanism.

5.7. Methodology

QUICKTALK can be used to gain better performance from an existing prototype written in Smalltalk. By using the profiling capabilities of the Smalltalk system, the most active branches on the method-call tree can be identified. After the classes of the method arguments are identified, the branches can then be collapsed into a single QUICKTALK method. The methods might need some modifications to fit into the QUICKTALK subset of Smalltalk.

Consider the existing Smalltalk compiler as an application prototype needing some performance improvement. Running a profile on the standard benchmark for the compiler reveals that 65% of the execution time is spent in building the parse tree and symbol table and 32% of the time is spent generating bytecodes. At the leaves, 6.8% is spent in a Dictionary method, **findKeyOrNil:**, and 5% of the compilation is spent in a Scanner method, **xLetter**, that forms a word or keyword. A special method for accessing the symbol table (the most active Dictionary) could be written in QUICKTALK to replace the use of method **findKeyOrNil:**. The method, **xLetter**, has some sections that could be easily converted to QUICKTALK.

6. SYSTEM DESIGN

This section describes design decisions and changes made to the Smalltalk-80 system to support user primitive-calling (UPC) methods.

6.1. User Interface

The user defines his UPC methods through the Smalltalk system browser, the standard interface to class and message definitions. A browser presents a hierarchical index to classes and messages. The compiler is invoked on the UPC method by the same mechanism as for a regular source method. In concept, compiling a UPC method creates a compiled method for the failure section that refers to the primitive routine for the primitive section.

6.1.1. Browser Interface

No change was made to the browser. It might be useful to place the primitive section and the failure section in separate browser panes.

6.1.2. Compiler Interface

Upon unsuccessful compilation of the primitive section of a UPC method, the compiler indicates why it failed. The QUICKTALK compiler can fail in all the ways the current compiler fails. In addition, a syntactically correct primitive section might not be compiled if an expression cannot be typed, a temporary variable is assigned with conflicting types, or a message selector appears that is not among the ones allowed for QUICKTALK. As an enhancement, the compiler could suggest changes that would allow it to complete.

6.2. Smalltalk Compiler

The Smalltalk-80 compiler was changed to store the compiled primitive methods. The next sections will describe this change as well as the changes made to the parser and code generator.

6.2.1. Storing Primitive Compiled Methods

A new dictionary called the *primitive method dictionary* (PMD), which is not associated with any class, has keys that are selectors of the messsages available in QUICKTALK. Since the same selector can refer to different methods based on the types of its arguments, the dictionary's values are an ordered list of primitive method descriptions. A primitive method description has the selector, receiver type, argument types, and return type, plus a selector and arguments, which, when sent to the code generator, will return machine code. The PMD currently holds the primitive method descriptions for the selectors that the compiler knows about; that is, those selectors identified in Section 5.4. In addition, the PMD holds the descriptions of the object-accessing selectors. When UPC methods are allowed as messages in other UPC methods, we plan for the PMD to index the UPC method selectors as well.

6.2.2. Changes to System Parser

The standard Smalltalk parser, after handling the message selector in a method, checks for a primitive section. This check has been generalized to handle either a system-primitive section (in angle brackets) or a user-primitive section (in set braces). The unmodified standard system parser handles the failure section while a modified parser handles the user-primitive section. For a user-primitive section, the parser must maintain a new temporary-variable name environment and create a separate parse tree.

The standard parser creates a parse tree whose root node is called the *method root*. (See Figure 12.) The method root has an instance variable that holds the integer for the system primitive referenced. If the method being parsed is not a PC method, this variable is zero. In the case of a UPC method, the new parser generalizes this instance variable to be a *primitive-method root*. The primitive-method root heads the primitive parse tree.

Each node of a primitive parse tree has an additional instance variable where it can store the type of expression it represents. The node type is assigned in a pass of the primitive parse tree before code generation. The node types are used to decide which primitive method description in the PMD is meant by a selector.

6.2.3. Changes to Code Generation

The first pass of the primitive parse tree produces a compiled method nearly identical to the standard system compiled method. Bytecodes are generated as a linearized intermediate form of the parse tree. (See Figure 13.)

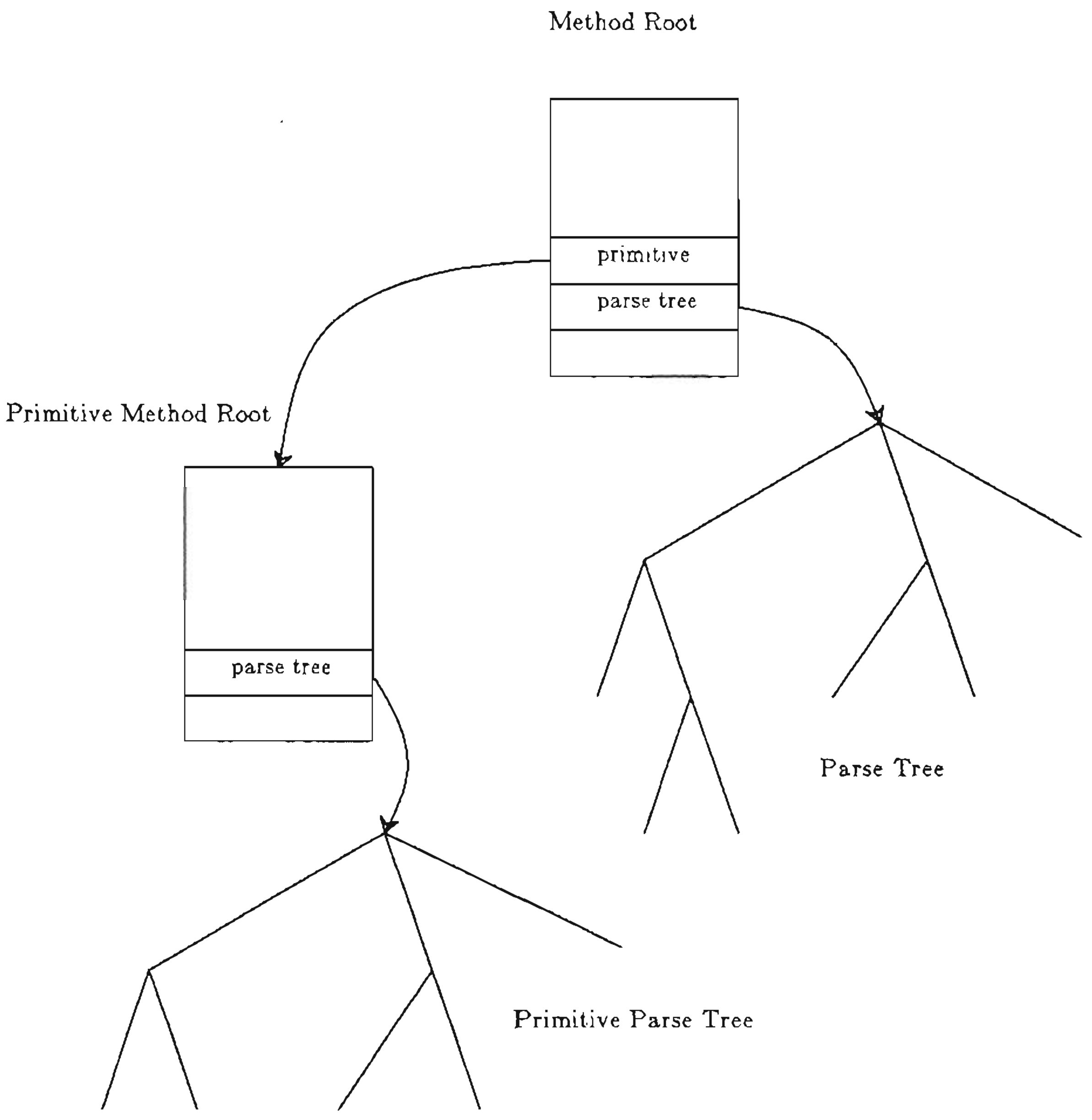
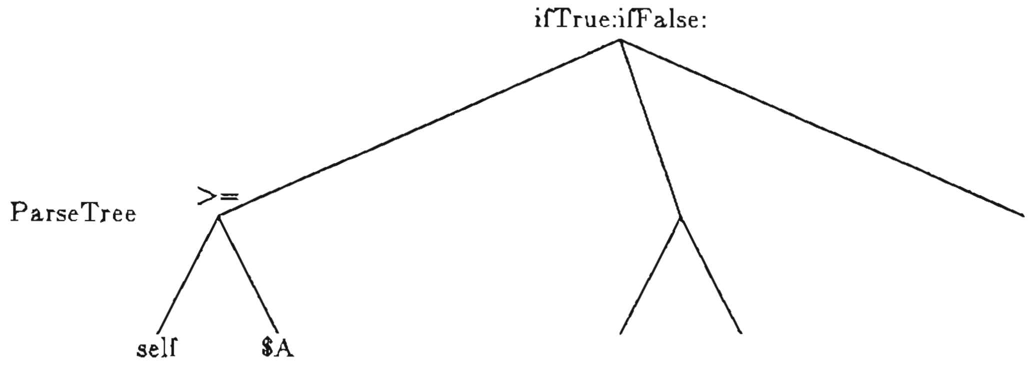


Figure 12: A Parse Tree for a User Primitive-Calling Method



```
                 ...
          15:   pushSelf
Bytecodes 16:   pushConstant: $A
          17:   send: >= Character x Character → Boolean
          18:   jumpFalse: 23
                 ...
```

```
                   ...
           pc15:   move.w    (receiver), freeReg
           pc16:   move.w    6+LiteralOffset(myHeader), anotherReg
           pc17:   sub.w     freeReg, anotherReg
68010              bgt.s     1f
NativeCode         move.w    #trueOop, anotherReg
                   bra       2f
           1       move.w    #falseOop, anotherReg
           2
           pc18:   sub.w     #falseOop, anotherReg
                   beq       pc23
                   ...
```

Figure 13: Code Generation

In a standard Smalltalk compiled method, send bytecodes reference their selectors as symbols stored in the literal frame, an area for storing constants that cannot be stored in bytecodes. In a compiled UPC method, for the QUICKTALK section, send bytecodes reference their selectors stored as primitive method descriptions. In Figure 13, the send bytecode (numbered 17) references the primitive method description for a character comparison.

The bytecode form is converted to assembler code for the native machine. Each bytecode in the compiled method is expanded to equivalent native code. Each send is expanded to inline code found in its primitive-method description. The native code uses unallocated registers to simulate the primitive routine's evaluation stack. The hardware stack of the native machine is used to spill registers. Register *receiver* of the native machine points to the message receiver on the interpreter's evaluation stack. It is used to access the receiver and method arguments. Register *myHeader* of the native machine points to the head of the primitive routine being executed. It is used to access the literals of the primitive routine. Finally, the assembler code is assembled to object code, which then replaces the bytecodes of the primitive routine.

6.3. Compiled Methods

A compiled UPC method consists of two objects; a compiled method for the failure section and a user-defined primitive routine. The user-defined primitive routine is the compiled version of the primitive section of a UPC method. The compiled method has an extra literal field that contains the object pointer of its user-defined primitive routine. This extra literal field is just above (ahead of) the literal field reserved for the extension of the header. (See Figure 14.)

A user-defined primitive routine is represented as a CompiledMethod that contains a header, literal frame, native code, and a trailer. The header contains the same information as the header of compiled methods. For example, the header contains the number of temporaries used by the compiled method. The literal frame contains references to floats, strings, arrays, and other constants that cannot be stored in immediate instructions. The trailer is retained to conform to the description of a CompiledMethod. Its contents are not yet defined.

6.4. Interpreter

The interpreter has a new Smalltalk primitive numbered 137. This primitive cannot be accessed from a user program directly, but is inserted in a compiled method by the QUICKTALK compiler. The primitive knows how to find the object reference to a user-defined primitive routine stored in a compiled method. The primitive performs the following steps:

(1) It finds the offset in the user-defined primitive routine where the native code begins.

(2) It jumps to the user-defined primitive routine's native code, passing two items: (a) the address of the top of interpreter stack, so the primitive routine can find its receiver and arguments, and (b) the header of the primitive routine, so the routine can find its literals.

The primitive routine assumes that the receiver and the method arguments are on top of the interpreter stack. Upon completion, the primitive routine returns

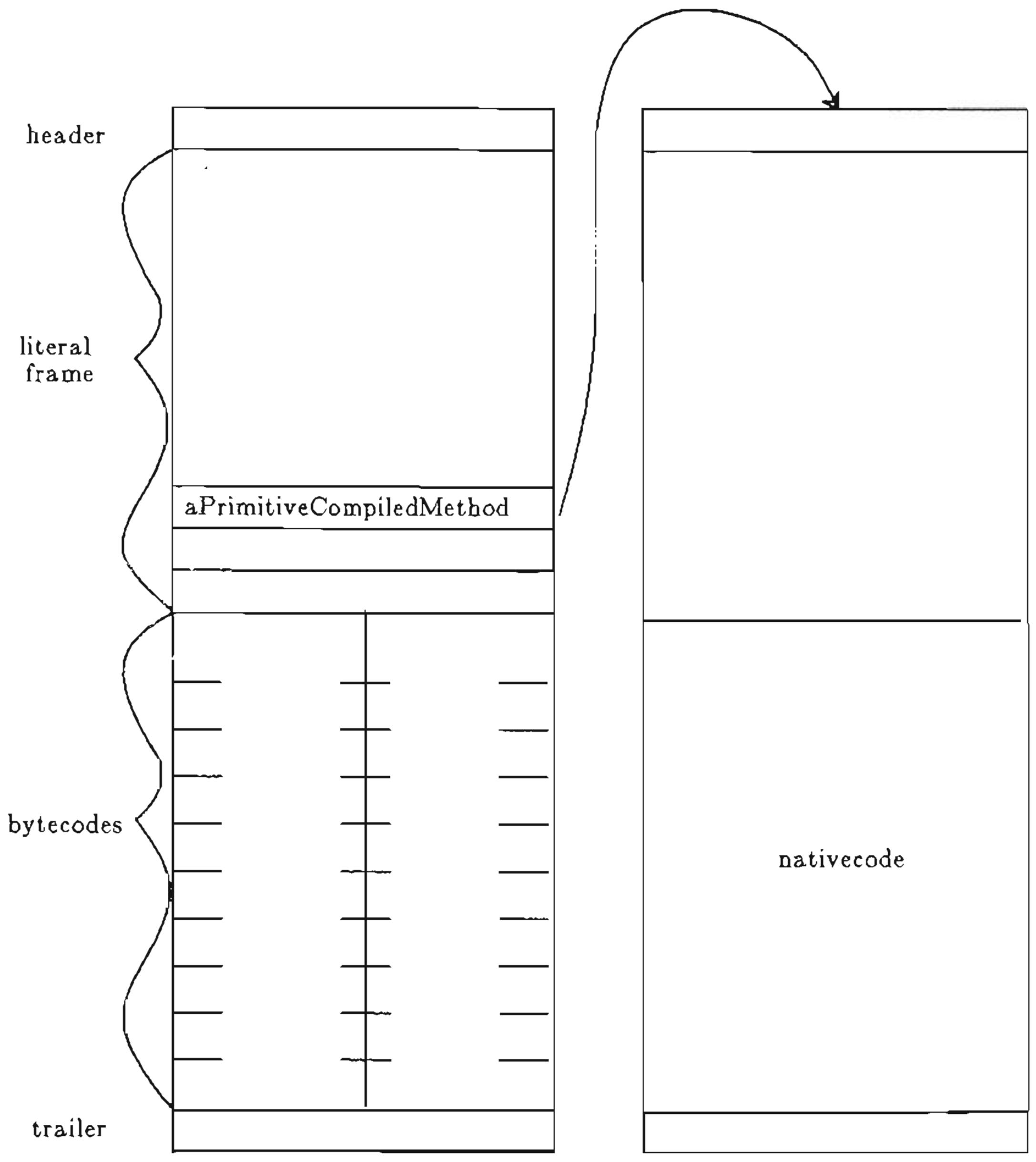


Figure 14: Compiled Method with a User-Defined Primitive Routine

control to primitive 137 and passes to primitive 137 a return value (an object reference) and a return code. The return code value can mean:

0 — failure, or
n — success, where n is the number of object pointers to pop from
the interpreter stack.

Upon successful completion of the primitive routine, primitive 137 should

(1) pop the number of object pointers specified from interpreter evaluation stack,

(2) push the return value onto the evaluation stack, and

(3) proceed to the next bytecode.

Upon failure of the primitive routine, primitive 137 jumps to the part of the interpreter that knows how to start the failure section. The interpreter assumes that the failed primitive routine has had no side effects.

6.5. Method Dependencies

Keeping a dictionary of methods that are dependent upon each other is not necessary until user-defined primitives can reference other user-defined primitives. At that time, dictionaries of dependencies of compiled primitive methods on types of instance variables, class variables, and other primitive-method argument types must be maintained. A technique for lazy recompilation could be devised so as not to degrade the interactive programming environment when a change to a method requires recompilation of its dependents.

6.6. Context Switching

Smalltalk primitive methods are considered indivisible operations. They need not be prepared to be interrupted. Thus, we need not worry about suspended executions of user-defined primitive routines.

6.7. Debugger

The normal Smalltalk debugger need not be modified. These new primitive Smalltalk methods are unobservable in the same way that normal primitives are not observable. Since QUICKTALK is a subset of Smalltalk, one can debug the logic of QUICKTALK methods by transforming them back to Smalltalk. This transformation consists of providing in the class Object a method that is just a no-op for each of the declaration messages, commenting out the failure section, and removing the set braces delimiting the QUICKTALK section. The resulting Smalltalk method should have the same meaning as the QUICKTALK method. In practice, some debugger should be provided to observe any incorrect behavior introduced by the implementation provided by the QUICKTALK compiler.

7. EXPERIMENTAL RESULTS

To measure the improvements in speed gained with UPC methods, some example methods have been compiled by the current QUICKTALK compiler and executed by a modified Smalltalk interpreter[10] that knows about UPC methods. QUICKTALK methods decrease execution time but increase the amount of space needed to represent compiled methods. This section quantifies the tradeoff for the example methods.

The execution times reported used the Smalltalk timing facility. The object *Time* is sent the message **millisecondsToRun:**, whose argument is a block containing the expression to be timed. Within that block, a message **to:do:** executes repeatedly the expression of interest it surrounds in order to get a valid measurement. The time required just for the **to:do:** looping was computed by timing the **to:do:** surrounding a null expression. This overhead time has been subtracted from the reported timing figures. The difference in time required for the lookup of these timed methods in its method dictionary is believed to be of negligible importance.[11] The *speedup factor* is the time required to execute the regular Smalltalk method divided by the time

[10]The interpreter was Version XI 5e Experiment < Fri Sep 6 1985 > running on a Tektronix 4404 68010 based workstation with two megabytes of memory

[11]Each method should reside in the method cache after the initial lookup Thus the difference in lookup, if any, would be amortized over each iteration

required to execute the UPC method.

The speedup factors for the dot product of arrays, substring searching, and substring replacement methods depended on the size of the problem. A percentage of the execution time difference is due to a one-time setup, and the rest depends upon the size of the objects involved. The results reported are for problems sizes where the speedup is near the asymptotic speedup.

7.1. Character Testing

Figure 15 compares a Smalltalk method and a QUICKTALK method for testing if a character is uppercase. Figure 16 compares the code-size expansion. Figure 17 reports timing results for the two methods. The expression $*A* means the character *A*. Characters are ordered as in ASCII; that is, $A-$Z < $a-$z. The UPC method

```
Class: Character
                              Regular Method
isUppercase
        "Answer whether the receiver is an uppercase letter."

        ↑ self >= $A and: [self <= $Z]
                         User Primitive-Calling Method
newIsUppercase
        "Answer whether the receiver is an uppercase letter."
{
        ↑ self >= $A and: [self <= $Z]
}
         self error: 'newIsUppercase failed'
```

Figure 15: isUppercase Methods

```
UPC method - 111 bytes
regular method - 19 bytes
expansion factor - 5.84
```

Figure 16: isUppercase Code Expansion

```
Time millisecondsToRun: [1 to: iteration do: [ :each | $a isUppercase]].
Time millisecondsToRun: [1 to: iteration do: [ :each | $a newIsUppercase]].
```

isUppercase Timing Results			
iteration	UPC	regular	speed-up factor
1000	65	830	12.77
10000	649	8314	12.81

Figure 17: isUppercase Timing

executes faster for two reasons. First, the sends for the Boolean tests are eliminated and, second, the comparison can be done with the character's object pointer instead of extracting the ASCII representation as defined in the Smalltalk class Character.[12] The timing results reveal a 12.8 speedup factor. Running the timing experiment where $A is tested for uppercase increased the UPC method execution times very slightly but only reduced the speedup factor to 11.9.

Figure 18 shows the bytecodes for the method **isUppercase**. Notice that the normal Smalltalk compiler compromises the meaning of the **and:** message by assuming the receiver is of the class Boolean. The block evaluation of **and:** is compiled to

[12]The correspondence between the ordering of the character's object pointer and the character's ASCII value is implementation dependent

truth-valued jump bytecodes. Since the message **and:** is not sent within the method, no class can reimplement its meaning. Without this optimization in the Smalltalk-80 compiler, the QUICKTALK method demonstrates a 21.4-fold speedup.

pc	bytecode	method statement
7	push: self	
8	push: $A	
9	send: >=	self >= $A
10	jumpFalse: 15	
11	push: self	
12	push: $Z	
13	send: <=	self <= $Z
14	jumpTo: 16	
15	push: false	
16	returnTop	↑

Figure 18: isUppercase Bytecodes

7.2. Iterative Sum

Figure 19 compares a Smalltalk method with a QUICKTALK method to add all the integers in an interval to the message receiver. Figure 20 compares the code-size expansion. Figure 21 compares the performance of these two methods. The experiment demonstrates a 22-fold speedup for integer addition with the compiled iterative control structure **to:do:**. Half of the speed up is due to eliminating the block evaluation.[13] The rest is due to eliminating bytecode decoding and simplifying the increment of the loop control variable.

Figure 22 shows the Smalltalk bytecodes for the method **sumFrom:to:**. The block for the increment of *sum* is compiled into all the bytecodes between the *send:* **blockCopy:** bytecode and the *blockReturn* bytecode. The **blockCopy:** message creats a block context. The method **to:do:** (bytecodes not shown) repeatedly evaluates this block context by sending it the message **value:** with the argument of the next element of the interval between *start* and *stop*.

[13]The Smalltalk method was rewritten to use a **whileTrue:** message which optimized the block evaluation to jump instructions This method ran twice as fast as the Smalltalk method with **to:do:**.

Class: SmallInteger

Regular Method

```
sumFrom: start to: stop
        "Add to the receiver the sum of the integers between
                start and stop; inclusive"
        | sum |
        sum ← self.
        start to: stop do: [ :index | sum ← sum + index].
        ↑ sum
```

User Primitive-Calling Method

```
mySumFrom: start to: stop
        "Add to the receiver the sum of the integers between
                start and stop; inclusive"
{
        | sum |
        start declare: SmallInteger.
        stop declare: SmallInteger.
        sum ← self.
        start to: stop do: [ :index | sum ← sum + index].
        ↑ sum
}
        ↑ self sumFrom: start to: stop
```

Figure 19: sumFrom:to: Methods

```
UPC method - 133 bytes
regular method - 27 bytes
expansion factor - 4.93
```

Figure 20: sumFrom:to: Code Expansion

```
Time millisecondsToRun:
        [1 to: iteration do: [ :each | 0 sumFrom: 1 to: 100]].
Time millisecondsToRun:
        [1 to: iteration do: [ :each | 0 mySumFrom: 1 to: 100]].
```

sumFrom Timing Results			
iteration	UPC	regular	speed-up factor
100	81	1799	22.21
1000	807	17573	21.78

Figure 21: sumFrom:to: Timing

```
pc    bytecode              method statement

 5    push: self
 6    popInto: sum          sum ← self.
 7    push: start
 8    push: stop
 9    push: ThisContext
10    push: 1
11    send: blockCopy:      (create block context with one arg)
12    jumpTo: 19
13    popInto: index
14    push: sum
15    push: index
16    send: +
17    storeInto: sum        sum ← sum + index.
18    blockReturn
19    send: to:do:          start to: stop do:[]
20    pop
21    push: sum
22    returnTop             ↑ sum
```

Figure 22: sumFrom:to: Bytecodes

7.3. Integer Point Addition

Figure 23 compares a Smalltalk method with a QUICKTALK method to return the Point that represents the sum of two Points. The Smalltalk method is much more general than the QUICKTALK method, since it can accept any argument that can be coerced to a Point by the message **asPoint**, and the Points can have coordinates that are a kind of Number. The QUICKTALK method, in contrast, is designed to handle only a Point argument whose coordinates are SmallIntegers. Figure 24 compares the code-size expansion. Figure 25 compares timing results for these two methods. Variables *x* and *y* name the coordinates. The message @ constructs a Point from two SmallIntegers. The experiment demonstrates a minor 1.38-fold speedup due to eliminating bytecode decoding. The large code expansion results from the inline type checking and the inline object allocation. Thus, the code expansion could be moderated with a small increase in execution time by jumping to a subroutine.

```
Class: Point
                         Regular Method
+ delta
        "Answer a new Point that is the sum of the receiver and delta
        (which is a Point or Number)."

        | deltaPoint |
        deltaPoint ← delta asPoint.
        ↑ x + deltaPoint x @ (y + deltaPoint y)
                     User Primitive-Calling Method
intPlus: deltaPoint
        "Answer a new Point that is the sum of the receiver and deltaPoint.
        Both points should have SmallInteger coordinates."
{
        x declare: SmallInteger.
        y declare: SmallInteger.
        deltaPoint declare: Point.
        x declareAccess: SmallInteger inClass: Point forFieldNamed: #x.
        y declareAccess: SmallInteger inClass: Point forFieldNamed: #y.
        ↑ (x + (deltaPoint x)) @ (y + (deltaPoint y))
}
        Transcript show: 'intPlus user primitive calling method failed'.
        ↑ self + deltaPoint
```

Figure 23: intPlus: Methods

UPC method - 511 bytes
regular method - 20 bytes
expansion factor - 25.55

Figure 24: intPlus: Code Expansion

```
aPoint ← 3 @ 4.
bPoint ← 5 @ 6.
Time millisecondsToRun:
        [1 to: iteration do: [ :each | aPoint + bPoint]].
Time millisecondsToRun:
        [1 to: iteration do: [ :each | aPoint intPlus: bPoint]].
```

intPlus: Timing Results			
iteration	UPC	regular	speed-up factor
1000	211	291	1.38
10000	2179	2999	1.38

Figure 25: intPlus: Timing

7.4. Dot Product

Figure 26 compares a Smalltalk method with a QUICKTALK method that answers the sum of the products of corresponding elements of two vectors with SmallInteger elements. Figure 27 compares the code-size expansion. Figure 28 compares the performance of these two methods. The experiment demonstrates a 5.0-fold speedup due to converting the **to:do:** block evaluation to a simple loop and by specializing the **at:** accessing message to the Array's **basicAt:**.

```
Class: Array
                              Regular Method
dot: anArray
        "Answer the sum of corresponding elements of self and anArray."
        | sum |
        sum ← 0.
        1 to: self size do: [:index | sum ← sum + ((self at: index)
                                        * (anArray at: index))].
        ↑ sum
                         User Primitive-Calling Method
myDot: anArray
        "Answer the sum of corresponding SmallInteger elements of self
        and anArray."
{
        | sum |
        self declareArrayOf: SmallInteger.
        anArray declareArrayOf: SmallInteger.
        sum ← 0.
        1 to: self basicSize do: [:index | sum ← sum + ((self basicAt: index)
                                        * (anArray basicAt: index))].
        ↑ sum
}
        Transcript show: 'myDot user primitive calling method failed'.
        ↑ self dot: anArray
```

Figure 26: myDot: Methods

```
UPC method - 435 bytes
regular method - 34 bytes
expansion factor - 12.79
```

Figure 27: myDot: Code Expansion

```
aArray ← Array new: 128.  "with each element an integer"
bArray ← Array new: 128.  "with each element an integer"
Time millisecondsToRun:
        [1 to: iteration do: [ :each | aArray dot: bArray]].
Time millisecondsToRun:
        [1 to: iteration do: [ :each | aArray myDot: bArray]].
```

myDot: Timing Results			
iterations	UPC	regular	speed-up factor
1000	6836	34281	5.01
10000	68356	342751	5.01

Figure 28: myDot: Timing

7.5. Substring Search

Figure 29 compares the standard Smalltalk system method for finding a substring of a given string with an equivalent QUICKTALK method. Figure 30 compares the code-size expansion. Figure 31 compares the performance of these two methods. The experiment demonstrates a 5.13-fold speedup. As before, the speedup is mainly due to eliminating the **to:do:** block evaluation. In addition, the messages **size** and **isEmpty** are specialized to **basicSize** and **at:** to **basicAt:**.

Class: String

Regular Method

```
findString: subString startingAt: start
        "Answer the index of subString within the receiver, starting
        at start. If the receiver does not contain subString, answer 0."

        | aCharacter index |
        subString isEmpty ifTrue: [↑ 0].
        aCharacter ← subString first.
        start to: self size - subString size + 1 do:
                [:startIndex |
                (self at: startIndex) = aCharacter ifTrue:
                        [index ← 1.
                        [(self at: startIndex+index-1) =
                                (subString at: index)] whileTrue:
                                [index = subString size ifTrue: [↑ startIndex].
                                index ← index+1]]].
        ↑ 0
```

User Primitive-Calling Method

```
myFindString: subString startingAt: start
        "Answer the index of subString within the receiver, starting
        at start. If the receiver does not contain subString, answer 0."

{
        | charRep index |
        self declareInternal: ByteArray.
        subString declareInternal: ByteArray.
        start declare: SmallInteger.
        subString basicSize = 0 ifTrue: [↑ 0].
        charRep ← subString basicAt: 1.
        start to: self basicSize - subString basicSize + 1 do:
                [:startIndex |
                (self basicAt: startIndex) = charRep ifTrue:
                        [index ← 1.
                        [(self basicAt: startIndex+index-1) =
                                (subString basicAt: index)] whileTrue:
                                [index = subString basicSize ifTrue: [↑ startIndex].
                                index ← index+1]]].
        ↑ 0
}
        Transcript show: 'findString:startingAt: user primitive calling method failed'.
        ^self findString: subString startingAt: start
```

Figure 29: myFindString: Methods

```
UPC method - 965 bytes
regular method - 76 bytes
expansion factor - 12.70
```

Figure 30: myFindString: Code Expansion

```
targetString ← '...eef'. "string 163 characters long ending with
             eef containing many false patterns starting eex"
searchString ← 'eef'.
start ← 1.
Time millisecondsToRun:
        [1 to: iteration do: [ :each |
             targetString findString: searchString startingAt: start]].
Time millisecondsToRun:
        [1 to: iteration do: [ :each |
             targetString myFindString: searchString startingAt: start]].
```

myFindString: Timing Results			
iterations	UPC	regular	speed-up factor
1000	15187	77882	5.13

Figure 31: myFindString: Timing

7.6. String Replacement

Figure 32 compares a Smalltalk PC method, a QUICKTALK method, and a Smalltalk method. Each method destructively replaces characters in a range of the receiving string using a range of elements in the replacement string. The Smalltalk PC method uses a system primitive whose functionality can be easily expressed in Smalltalk but is provided as a primitive for performance. Figure 33 compares the code-size expansion. Figure 34 compares the performance of the Smalltalk PC method (which has a handcoded primitive section) and the QUICKTALK UPC method. The experiment demonstrates a 0.038-fold speedup compared with the handcoded primitive, that is, 25-30 times slower. The handcoded primitive takes advantage of knowing that Array elements are stored in sequential memory. It insures that the arguments to the method *replace* do not violate array boundaries and then copies memory from one Array to the other. The QUICKTALK method accesses both Arrays one element at a time and checks bounds on each access.

Figure 35 compares the performance of the UPC method with the normal Smalltalk code. A 3.31-fold speedup results compared with the equivalent Smalltalk method.

Class: String

Regular Method

```
primReplaceFrom: start to: stop with: replacement startingAt: repStart
        "This destructively replaces elements from start to stop in the receiver
        starting at index, repStart, in the collection, replacement. Answer the
        receiver. The range errors cause the primitive to fail."

        <primitive: 105>
        super replaceFrom: start to: stop with: replacement startingAt: repStart
```

User Primitive-Calling Method

```
myReplaceFrom: start to: stop with: replacement startingAt: repStart

        "This destructively replaces elements from start to stop in the receiver
        starting at index, repStart, in the string, replacement. Answer the
        receiver."
{
        | index repOff |
        self declareInternal: ByteArray.
        start declare: SmallInteger.
        stop declare: SmallInteger.
        replacement declareInternal: ByteArray.
        repStart declare: SmallInteger.
        repOff ← repStart - start.
        index ← start - 1.
        [(index ← index + 1) <= stop]
                whileTrue: [self basicAt: index put: (replacement basicAt: repOff + index)]
}
                Transcript show: 'replace: user primitive calling method failed'.
```

Regular Non-Primitive Method

```
failedReplaceFrom: start to: stop with: replacement startingAt: repStart
        "This destructively replaces elements from start to stop in the receiver
        starting at index, repStart, in the string, replacement. Answer the
        receiver."
        | index repOff |
        repOff ← repStart - start.
        index ← start - 1.
        [(index ← index + 1) <= stop]
                whileTrue: [self at: index put: (replacement at: repOff + index)]
```

Figure 32: myReplaceFrom: Methods

UPC method - 469 bytes
regular method - 45 bytes
expansion factor - 10.42

handcoded primitive method - 266 bytes

Figure 34: myReplaceFrom: Code Expansion

```
target ← 'aaa...a'. "200 a's"
replSource ← 'bb...b'. "160 b's"
initial ← 6.
stop ← replSource size + initial - 1.
start ← 1.
Time millisecondsToRun:
        [1 to: iteration do: [ :each | target primReplaceFrom: initial
                to: stop with: replSource startingAt: start ]].
Time millisecondsToRun:
        [1 to: iteration do: [ :each | target myReplaceFrom: initial
                to: stop with: replSource startingAt: start ]].
```

myReplaceFrom: Timing Results			
iterations	UPC	regular	speed-up factor
1000	9124	350	0.038
10000	91227	3445	0.038

Figure 34: myReplaceFrom: Timing Against Handcoded Primitive

```
target ← 'aaa...a'. "200 a's"
replSource ← 'bb...b'. "160 b's"
initial ← 6.
stop ← replSource size + initial - 1.
start ← 1.
Time millisecondsToRun:
        [1 to: iteration do: [ :each | target failedReplaceFrom: initial
                to: stop with: replSource startingAt: start ]].
Time millisecondsToRun:
        [1 to: iteration do: [ :each | target myReplaceFrom: initial
                to: stop with: replSource startingAt: start ]].
```

myReplaceFrom: Timing Results			
iterations	UPC	regular	speed-up factor
1000	9124	30218	3.31
10000	91241	302128	3.31

Figure 35: myReplaceFrom: Timing Against Equivalent Smalltalk

8. SUMMARY AND EXTENSIONS

The following sections summarize limitations in the design of QUICKTALK. We propose extensions that have been ordered beginning with those we feel most important. Each proposed extension in the dialect must be evaluated against the purpose of the dialect, producing primitive methods. The quality of code produced by the compiler must not be degraded by adding features to the dialect, since the major motivation for writing a primitive is performance.

8.1. Limitation of Approach

The most severe constraint in the design of QUICKTALK is that imposed by maintaining the semantics of primitives as transactions whose execution cannot be suspended and whose effects are not visible upon failure. On the other hand, not supporting suspensions makes QUICKTALK attractive from the engineering viewpoint. A mapping does not need to be provided between suspended QUICKTALK methods and the bytecodes of the Smalltalk virtual machine.

A second limitation lies in the amount of performance improvement one should expect from a QUICKTALK compiler. Recall the QUICKTALK method for replacing a substring of a string. The current, very naive, QUICKTALK compiler generated code for this method which compared most unfavorably with the equivalent handcoded primitive. It would be hard, though not impossible, to construct a compiler sufficient to recognize the block memory move and thus approach the speed of the handcoded primitive.

8.2. Float Operations

Adding floating-point operations will complete the arithmetic. We expect to get much performance improvement here. QUICKTALK should be able to use a native-machine-dependent representation of floating-point numbers, converting to the Smalltalk form for returned values. For example, computing the dot product of two arrays of floating-point numbers should perform much faster in a QUICKTALK primitive than in an equivalent Smalltalk method.

8.3. Creation of Objects

User-defined primitives need to create objects for internal use and to return computed objects to the calling environment. With object creation comes the possibility that the garbage collector might interrupt a user-defined primitive routine and move any object in memory. Most insidiously, the primitive routine itself is an object and might be moved by the garbage collector. Thus, if the primitive routine wishes to call any interpreter subroutines, like object creation, a simple return address mechanism for returning to the primitive method is not sufficient.

8.4. Robust Compiler

A robust compiler should be able to explain its failures to compile. It should fail when the UPC method is syntactically incorrect or mistyped. Of course, QUICKTALK code should have the same semantics as the Smalltalk code. If the code that QUICKTALK generated for system primitives used in user-defined primitives was copied from the same source as the interpreter's primitive, then maintaining equivalent semantics would be more easily guaranteed.

After the above extensions, that is, floating-point operations, secure linkage to interpreter subroutines, and a robust compiler are completed, QUICKTALK should be quite useful.

8.5. Improved Code Generation

A significant improvement in code size and speed was gained by simulating the evaluation stack inside the compiler and using the 68010's registers. More sophisticated techniques could uncover further optimizations. For example, the compiler could identify redundant bounds checks on an Array access. Thus, the reported code expansions should be understood as an upper bound and the speedup factors a lower bound on what is readily achievable.

8.6. Inline Insertion vs. Subroutines

Currently, QUICKTALK only generates inline code. It should be able to share common support routines, such as, object allocation. It should be able to call existing UPC methods. This ability requires the concept of an activation record for the primitive. The compiler could then make the space/time tradeoff of jumping to a subroutine or copying the subroutine inline. Calling subroutines would make recursion possible. The UPC method writer should be aware of the ramifications of a primitive method preventing interrupts from being serviced and should use care. UPC methods requiring intensive computation might lock out a user from his terminal.

8.7. UPC Methods with Union Types

Some UPC methods would be more conveniently expressed if they were allowed to operate on arguments each of which might come from a set of classes. For example, a method to add two Points should be able to accept Points with SmallInteger or Float coordinates. The type system could be generalized to allow union types. With a more general type system, the compiler would be responsible for generating the case selection.

8.8. Summary

The QUICKTALK dialect of Smalltalk-80 can be viewed as an experiment in adding a notion of static typing to a dynamically typed language. The dialect is designed to describe primitive Smalltalk methods. Improved performance over bytecodes is achieved by eliminating the interpreter loop on bytecode execution, by reducing the number of message send/returns via binding some target methods at compilation, and by eliminating redundant class checking.

REFERENCES

[AbS85] Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[AhU79] Aho, A. V. and Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1979.

[BoI81] Borning, A. H. and Ingalls, D. H. H., "A Type Declaration and Inference System for Smalltalk," 81-08-02a, U. of Washington, Seattle, WA, Nov. 1981.

[CiP83] Citrin, W. and Ponder, C., "Implementing a Smalltalk Compiler," CS292R, University of California, Berkeley, CA, Mar. 1983.

[DeS84] Deutsch, L. P. and Schiffman, A. M., "Efficient Implementation of the Smalltalk-80 System," *11th Annual ACM Symp. on Prin. of Programming Languages*, Jan. 1984, pp. 297-302.

[GoR83] Goldberg, A. and Robson, D., *Smalltalk-80, The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Hag83] Hagmann, R., "Preferred Classes: A Proposal for Faster Smalltalk-80 Execution," *Smalltalk-80 Bits of History, Words of Advice*, Reading, MA, 1983, pp. 323-330.

[Ing78] Ingalls, D. H. H., "The Smalltalk-76 Programming System Design and Implementation," *5th Annual ACM Symp. on Prin. of Programming Languages*, Jan. 1978, pp. 9-15.

[LaB83] Larus, J. and Bush, W., "Classy: A Method for Efficiently Compiling Smalltalk," CS292R, University of California, Berkeley, CA, Mar. 1983.

[LSA77] Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., "Abstraction Mechanisms in CLU," *Comm. ACM*, vol. 20, 8 (Aug. 1977), pp. 564-576.

[Rob85] Roberts, G., "An Experimental Type Declaration and Type Checking System for Smalltalk-80," M.Sc. project, Queen Mary College, University of London, London, England, Oct. 1985.

[Suz81] Suzuki, N., "Inferring Types in Smalltalk," *8th Annual ACM Symp. on Prin. of Programming Languages*, Jan. 1981, pp. 187-199.

[SuT84] Suzuki, N. and Terada, M., "Creating Efficient Systems for Object-Oriented Languages," *11th Annual ACM Symp. on Prin. of Programming Languages*, Jan. 1984, pp. 290-296.

[Wir85] Wirfs-Brock, A., "The Design of a High-Performance Smalltalk Implementation," *Nikkei Electronics* , June 3, 1985, pp. 233-245. (in Japanese).

BIOGRAPHICAL NOTE

The author was born September 17, 1953, in Shelbyville, Kentucky. He graduated from Shelbyville High School in 1971. In 1975, he received a Bachelor of Arts degree in Mathematics (Magna Cum Laude) from Duke University, Durham, North Carolina. In September, 1982, the author attended the Wang Institute of Graduate Studies in Tyngsboro, Massachusetts, and received a Master of Software Engineering degree in August, 1983. In September, 1983, he began study at the Oregon Graduate Center and during the first year held the Tektronix Fellowship.

From 1975 to 1977, the author worked as a programmer on the User Services Staff of the Duke University Computation Center. In 1979, he worked as a programmer for the Urban Studies Center in Louisville, Kentucky. Between 1980 and 1982, the author worked as a programmer analyst for the Washington Library Network in Olympia Washington. In 1978, he was an assistant project leader for a self help water-well construction project in Mali, Africa with United World Missions.

The author has been married since 1983 to Marguerite Mautner, and they enjoy biking and backpacking in the Oregon Cascades. The author has accepted a position with the Artificial Intelligence Machines Program, a group within Tektronix, Inc.