QUICKTALK:
A Smalltalk-80 Dialect for Defining Primitive Methods

Mark B. Ballard
David Maier
Allen Wirfs-Brock

Oregon Graduate Center
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999

Technical Report CS/E-86-004
April, 1986

QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods

Mark B. Ballard
David Maier
Oregon Graduate Center

Allen Wirfs-Brock
Computer Research Lab
Tektronix, Inc. 50-662
PO Box 500
Beaverton, OR 97077
(503) 627-6195

ABSTRACT

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. We designed the dialect to describe primitive Smalltalk methods. QUICKTALK achieves improved performance over bytecodes 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. We identify changes to the Smalltalk-80 system and compiler to support the dialect, and give performance measurements.

1. INTRODUCTION

Some problems require experimentation or prototyping to discover an acceptable programmed solution. User-interface design is one such problem, 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 by requiring less specification from 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.

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 programmer to gain efficiency in a mature application by typing variables in frequently used procedures.

2. SMALLTALK AND PRIMITIVES

We assume the reader is familiar with the Smalltalk-80[1] system, and implementation techniques for it, as described in Goldberg and Robson [GoR83] and Krasner [Kra83]. In this section we restate some of the properties of the Smalltalk virtual machine that are of importance to the QUICKTALK strategy.

The Smalltalk-80 system is specified by a stack-oriented virtual machine. Source methods are translated by the system compiler into *compiled methods,* which contain sequences of eight-bit instructions, called *bytecodes*, for the virtual machine. The Smalltalk interpreter executes the bytecodes. The interpretation of most bytecodes involves an evaluation 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.

The interpreter usually responds to a send bytecode, sometimes called a *message send*, by interpreting a compiled method associated with the message name. The send bytecode causes a significant change to the state of the interpreter. The sending method places the receiver and arguments on the evaluation 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. 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.

[1]Smalltalk-80 is a trademark of Xerox Corporation.

Some methods, called ***primitive-calling*** (PC) methods invoke a ***primitive routine***[2] in native machine code, in addition to Smalltalk source code. Primitive routines give Smalltalk the ability to create objects, evaluate expressions, provide access to some virtual machine structures, and are used to optimize some critical methods.

A ***system primitive-calling*** (SPC) method has a ***primitive section*** and a ***failure section.*** The primitive section simply references a system-supplied primitive routine by number. The failure section consists of regular Smalltalk code to be performed if the primitive fails. An SPC method has its failure section compiled to a regular compiled method, except that a reference number for to a system-supplied primitive routine is included. Figure 1 shows the SPC method for the message selector +, which 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, such as an argument of the wrong class. The failure section handles these exceptional cases.

A send bytecode that invokes a compiled SPC method is interpreted by first trying the primitive routine. If the primitive routine completes successfully, it replaces the receiver and arguments on the evaluation stack by the result of the routine. 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.

Smalltalk programmers would like to write their own primitive methods 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***

```
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 1: A System Primitive-Calling Method

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

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.

Many methods send messages to only the existing compiled PC methods, and none to regular compiled methods. A large portion of methods have arguments and results of the same class for nearly every call of the method. Thus, many methods could have their message selectors bound to methods during compilation. QUICKTALK is designed to handle such methods that call only PC methods and whose arguments 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.

The QUICKTALK dialect adds type declarations for method arguments as well as instance variable and class variables used in the method.[3] It restricts the use of block expressions to a set of control structures. The selectors that can be used in QUICKTALK also are restricted.

The problems of adding staticly 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, only those components actually used in the method should be type-checked. For example, a UPC method that expects an array of integers and is looking for the index of the first element equal to zero 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 as array of character objects, but as an array of bytes. Some UPC methods might be able to ignore the character objects and operate directly on the byte representation. QUICKTALK must provide a way for a UPC methods to declare its intention to operate on the representation of an object. Thus, a particular string object could be treated as an array of bytes, so that **at:** would return a 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 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, type checking can not detect that the sum of two SmallIntegers will not 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 [MOP85] . 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. Consistent with typing methods rather than the instance variables of a class, the object-accessing selectors are typed.

[3]The current implementation does not handle class variables.

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 can not 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 incremental typing of 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 while the semantic action requires only 1 microsecond [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 types 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, although types can be parameterized; e.g. "Collection of: X". In their proposal, they add to the Smalltalk language explicit type declarations to method arguments and returned values. 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. 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 parameterized types. 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 parameterized types.

4.2. Compiling Smalltalk

Hagmann [Hag83], adds a class declaration to method arguments; the class that is expected in the majority of method activations. Thus, his types are "hints" or "preferences". 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 encounter a value that does not match the preferred class, then the execution must be continued in the standard compiled method. He must deal 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, foregoing 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. Second, he allocated method contexts (activation records) on a linear stack, only promoting them to standard Smalltalk objects when necessary. Third, he suggested that the bytecodes could be dynamically expanded 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 UPC method format, the typing discipline, the control structures, and the message selectors that are permitted in the dialect.

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 2 for an example UPC method with the sections annotated.

Notice that the type declaration statements (defined 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.

```
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 2: Example UPC Method

5.2. Typing

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:

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

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. The types of temporary variables are inferred when assigned the value of an expression that can be typed.

Types of identifiers (method arguments and class variables), and object-accessing selectors are declared by the messages in Figure 3. The declaration messages may appear as statements anywhere in the user-primitive section before the use of the declared symbol. The message **declare:** declares an identifier that will denote only objects of the given class. (Subclasses are disallowed, except for Object). For example,

x declare: Point.

declares the identifier *x* will denote an object of class Point. The exception for class Object allows a method to accept an arbitrary object when its type is not needed by any operations.

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. 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.

The message **declareArrayOf:** is used to declare that an identifier denotes an Array whose elements are of a single class. 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 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

---

<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 3: Type-declaring and Object-accessing Selectors

---

when applied to any object of the class Rectangle. When the message name is the same as that of the instance variable name, the declaration above 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

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, to have an internal representation, or to be Array are checked upon entry. *Elements* of Array's are checked upon extraction, so elements not extracted will never be checked. Object-accessing selectors declared with either **declareAccess:** message invoke methods that 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.

Only the pseudo-variables, *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.

5.3. Blocks

A block expressions in Smalltalk describes an object representing a deferred sequence of actions. A QUICKTALK method may use blocks only with the selectors identified in Figure 4. These blocks and selectors supply the Smalltalk programmer with the standard conditional and looping control structures.

5.4. Selectors

Figures 5, 6, and 7 (and Figure 3 on type-declaring and object-accessing selectors) contain the set of all selectors that can be used in UPC methods.[4] 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.

```
<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 4: Primitive Blocks

[4]Floating point selectors have not been implemented.

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

Figure 5: Compiler-Known Selectors — Arithmetic selectors

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

Figure 6: Compiler-Known Selectors — Non-Arithmetic

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

Figure 7: Compiler-Known Selectors — Additional

Figure 7 lists messages that are novel with QUICKTALK in addition to those in Figure 3. 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 a PC method must execute in an environment as if the primitive routine had not been tried. Responsibility for insuring that the primitive leaves its environment unchanged upon failure rests solely with the programmer.

6. SYSTEM DESIGN

This section describes design decisions and changes made to the Smalltalk-80 system to support UPC methods.

6.1. Interfaces

The user defines his UPC methods through the Smalltalk system browser, the standard interface to class and message definitions. The compiler is invoked on the UPC method by the same mechanism as for a regular source method. 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 can not be typed, a temporary variable is assigned with conflicting types, or a message selector appears that is not among the ones allowed for QUICKTALK.

6.2. Smalltalk Compiler

6.2.1. Storing Primitive Compiled Methods

A new dictionary, called the *primitive method dictionary* (PMD), has keys that are selectors of the messages available in QUICKTALK. Since the same selector can refer to different methods based on the types of its arguments, the dictionary's values are sets 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 QUICKTALK compiler allows plus the descriptions of any declared object-accessing selectors.

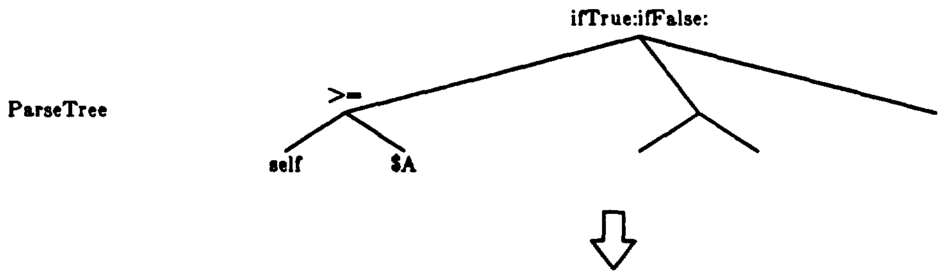
6.2.2. Changes to System Parser

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). A modified parser handles the user-primitive section. That 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, called the *method root*, holds the number of a system primitive, if the method being parsed is a PC method. In the case of a UPC method, the new parser generalizes this instance variable to be a *primitive-method root* that heads the primitive parse tree. Each node of a primitive parse tree has an additional instance variable for its type, which 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 8.)

In a standard Smalltalk compiled method, send bytecodes reference their selectors as symbols stored in the *literal frame*. In a compiled UPC method, for the QUICKTALK section, send bytecodes reference their selectors stored as primitive-method descriptions. For example, in Figure 8, the send bytecode (numbered 17) references the primitive method description for a character comparison. Each bytecode in the compiled method is expanded



```
                   ...
            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 8: Code Generation

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 evaluation stack, and is used to access the receiver and method arguments. Register *myHeader* points to the head of the primitive routine being executed, providing 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 machine-code version of the primitive section and is referenced from the compiled method.

6.4. Interpreter

The interpreter has a new Smalltalk primitive (137) that knows how to find the object reference to a user-defined primitive routine stored in a compiled method. The primitive finds the offset in the user-defined primitive routine where the native code begins, and begins executing there, passing (a) 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. Upon completion, the primitive routine returns control to primitive 137, passing back a return code indicating failure or the number of object pointers to pop from the interpreter stack, and a return value for primitive 137 to push on the interpreter stack. Upon failure of the primitive routine, primitive 137 jumps to the part of the interpreter that knows how to start the failure section.

6.5. Debugger

The normal Smalltalk debugger was not 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.

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[5] that knows about UPC methods. The source methods for the experiment may be found in Figure 9 and in the Appendix. QUICKTALK methods decrease execution time but increase the amount of space needed to represent compiled methods. Figures 10 and 11 at the end of this section quantify the tradeoff for the example methods. The source for the methods may be found in the Appendix. 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 executes repeatedly the expression of interest it surrounds in order to get a valid measurement. This **to:do:** 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. 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. The *speedup factor* is the time required to execute the regular Smalltalk method divided by the time 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 near the asymptotic speedup.

7.1. Character Testing

Figure 9 compares a Smalltalk method and a QUICKTALK method for testing if a character is uppercase. The UPC method 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. The timing results reveal a 12.8 speedup factor. Running the

[5]The interpreter was Version X1.5e Experiment < Fri Sep 6 1985 > running on a Tektronix 4404 68010-based

```
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 9: isUppercase Methods

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. 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 truth-valued jump bytecodes. Without this optimization in the Smalltalk-80 compiler, the QUICKTALK method demonstrates a 21.4-fold speedup.

7.2. Iterative Sum

The **sumFrom:to:** method compares a Smalltalk method with a QUICKTALK method to add all the integers in an interval to the message receiver. 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.[6] The rest is due to eliminating bytecode decoding and simplifying the increment of the loop control variable.

7.3. Integer Point Addition

The next test compares a Smalltalk method with a QUICKTALK method + for Points. The Smalltalk method is 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. 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.

7.4. Dot Product

Next, we compare a Smalltalk method with a QUICKTALK method that answers the sum of the products of corresponding elements of two vectors with SmallInteger elements. 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:**.

---

workstation with two megabytes of memory.

[6]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:**.

7.5. Substring Search

Next we compare the standard Smalltalk system method for finding a substring of a given string with an equivalent QUICKTALK method. 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:**.

7.6. String Replacement

Our final experiment 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. The experiment demonstrates a 0.038-fold speedup of the QUICKTALK UPC method compared with the handcoded primitive, that is, about 26 times slower. The handcoded primitive takes advantage of knowing that Array elements are stored in sequential memory copying memory from one Array to the other. The QUICKTALK method accesses both Arrays one element at a time and checks bounds on each access. A 3.31-fold speedup results compared with the equivalent Smalltalk method.

name	UPC method	regular method	expansion factor
isUppercase	111 bytes	19 bytes	5.84
sumFrom:to:	133 bytes	27 bytes	4.93
intPlus:	511 bytes	20 bytes	25.55
myDot:	435 bytes	34 bytes	12.79
myFindString:	965 bytes	76 bytes	12.70
myReplaceFrom:	469 bytes	45 bytes	10.42
handcoded primitive		266 bytes	

Figure 10: Code Expansion

	iterations	UPC	regular	speed-up factor
isUppercase	1000	65	830	12.77
	10000	649	8314	12.81
sumFrom	100	81	1799	22.21
	1000	807	17573	21.78
intPlus	1000	211	291	1.38
	10000	2179	2999	1.38
myDot	1000	6836	34281	5.01
	10000	68356	342751	5.01
myFindString	1000	15187	77882	5.13

	iterations	UPC	primitive	regular	speed-up factor
myReplaceFrom	1000	9124	350		0.038
	10000	91227	3445		0.038
myReplaceFrom	1000	9124		30218	3.31
	10000	91241		302128	3.31

Figure 11: Timing

8. LIMITATIONS AND FUTURE WORK

The following sections summarize limitations in the design of QUICKTALK. We propose extensions that have been ordered beginning with those we feel most important. 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 can not 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. The compiler could suggest changes that would allow it to complete. 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.

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. New QUICKTALK methods should be able to call existing UPC methods. This ability requires the concept of an activation record for the primitive and there might be a different argument passing mechanism. The compiler could then make the space/time tradeoff of jumping to a subroutine or copying the subroutine inline. 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.

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 methods argument types must be maintained. A technique for lazy recompilation could be used so as not to degrade the interactive programming environment when a change to a method requires recompilation of its dependents.

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

For a more complete description of QUICKTALK, see the thesis by Ballard [Bal86].

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

[Bal86] Ballard, M. B., "QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods," Master's Thesis, Oregon Graduate Center, Dept. of CS&E, Beaverton, OR, Apr. 1986.

[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.

[Kra83] G. Krasner, ed., *Smalltalk-80 Bits of History, Words of Advice*, Addison-Wesley, Reading, MA, 1983.

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

[MOP85] Maier, D., Otis, A. and Purdy, A., "Object-Oriented Database Development at Servio Logic," *Database Engineering*, vol. 8(Dec. 1985), pp. 58-65.

[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).

APPENDIX

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'
```

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
```

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
```

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
```

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
```

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)]
```