TS: An Optimizing Compiler for Smalltalk

Ralph E. Johnson
Justin O. Graver
Lawrence W. Zurawski

Department of Computer Science
University of Illinois, Urbana-Champaign

Abstract

TS (Typed Smalltalk) is a portable optimizing compiler that produces native machine code for a typed variant of Smalltalk, making Smalltalk programs much faster. This paper describes the structure of TS, the kinds of optimizations that it performs, the constraints that it places upon Smalltalk, the constraints placed upon it by an interactive programming environment, and its performance.

1 Introduction

A number of recent Smalltalk implementations have acceptable performance[DS83][CW86][SUH86]. This has been achieved as much by the increase in processor speed as by improvements in software technology. However, all current implementations of Smalltalk are slower than those of languages like C. Smalltalk's poor performance is usually ascribed to late-binding of procedure calls, heavy use of closures, and dynamic memory management. However, Smalltalk implementations such as PS and SOAR reduce the cost of these language features. SOAR spends two-thirds of its time executing primitives. Thus, even if procedure calling and memory management overhead were completely removed, SOAR could be no more than a third faster. TS (Typed Smalltalk) is an optimizing compiler for Smalltalk that results in a large speedup over interpreters. The performance of TS indicates that the real reason that Smalltalk is inefficient is information-hiding provided by object-oriented programming.

Most attempts to speed-up Smalltalk have focused on optimizing the interpreter instead of building an optimizing compiler. The reason is easy to see. In the absence of type information, a compiler cannot determine which method (procedure) is being invoked by a message send (procedure call). Since control structures, field selection, arithmetic operations, and array accesses are all accomplished by sending messages, the compiler has virtually no information on which to make optimizations.

The importance of a type system is illustrated by two compilers that compile subsets of Smalltalk to efficient machine code: Hurricane[Atk86] and Quicktalk[BMW86]. Both compilers use a type system to determine the methods that would be invoked by a particular message send. Quicktalk produces the fastest code but has the most restrictive type system. Quicktalk produces speed-ups of over 20 using only modest optimizations, but few large Smalltalk methods satisfy the restrictions of the compiler. While neither Quicktalk nor Hurricane accept full Smalltalk, they show the potential for code optimization to improve the speed of Smalltalk.

The type system of TS[Joh86][JG87] is more powerful than the type systems of Hurricane or Quicktalk. It type-checks most large Smalltalk methods, ensures that each variable always contains an object whose class is compatible with the type of the variable, and allows the type of a variable or expression to be expressed precisely enough that the compiler can deduce the set of methods that can be invoked by a particular message. Neither of the earlier type systems is sufficient for type-checking all Smalltalk methods, and neither ensures that the contents of a variable is always an object in a particular set of

Authors' address: Department of Computer Science, 1304 West Springfield Ave., Urbana IL 61801
Telephone: (217) 244-0093
E-mail: {johnson,graver,zurawski}@cs.uiuc.edu

classes. Quicktalk treats compiled methods as "user primitives", and if the classes of the arguments differ from their types then the primitives fail, just like normal Smalltalk primitives. Hurricane treats types as hints to the compiler, and produces code to handle the cases where the classes of objects are different than what was expected.

TS uses a set of very general optimizations, most notably early binding of procedure calls (message sends) and in-line substitution of calls to user-defined and primitive methods. The result is that Smalltalk programs are converted into a form similar to C or Pascal programs, permitting standard code optimization techniques to be used. TS has been designed so that it can be easily ported to any machine with 32 bit integers and pointers. It currently runs on a Tektronix 4405 and produces Motorola 68020 machine language.

2 An Overview of TS

TS reuses many of the components of the Smalltalk-80 programming environment, including most of the original compiler. The reused classes served as an interface between the various people working on the project. This reuse of code and interfaces let six inexperienced Smalltalk programmers build the compiler in a relatively short time.

The standard Smalltalk parser produces a parse tree consisting of instances of subclasses of ParseNode. We changed the parser to allow the types of variables to be declared. Since the Smalltalk-80 compiler uses a recursive descent parser, it was easy to build the the parser for TS by subclassing the Smalltalk-80 Parser class.

A TS parse tree can type-check itself, resulting in a parse tree decorated with type information[Gra86]. In particular, each parse-node representing a message send knows the type of the receiver. There are many different kinds of types, and some of them specify a set of classes.

Once a parse tree is type-checked, the type information can be used to perform high-level optimizations[Loy88]. The ones that are currently implemented are:

- conversion of a message send into a case statement with procedure calls (when the receiver is known to be in a small set of classes),
- in-line substitution of procedure calls,
- tail recursion elimination, and
- beta reduction, i.e. elimination of block creation and evaluation.

These are only a small fraction of the optimizations that are potentially useful. However, they are sufficient to provide a dramatic performance increase.

TS converts optimized parse trees into programs written in a machine-independent register transfer language, which is also used to write Smalltalk primitives. This increases portability and is useful for optimizations because it allows uniform in-line substitution of primitives and of methods written in Smalltalk. The code generation algorithm performs several optimizations on the register transfer instructions, often eliminating entire uses of the primitives. A single representation for code simplifies the compiler, as well.

The code generator is modeled after PO, by Davidson and Fraser[DF80]. PO takes a stream of assembly language instructions, converts each instruction into a sequence of register transfer instructions, optimizes the resulting sequence, and then converts it back to assembly language. The main optimization is common subexpression elimination. The conversion back to assembly language selects the best instruction for a sequence of register transfer instructions. The result is that peephole optimization is performed automatically by a machine independent algorithm.

We modified the algorithm of PO slightly[Wie87]. Assembly language is never generated. Instead, parse trees are converted directly to register transfer instructions. Register transfer instructions are also used to describe the Smalltalk primitives, which are usually written in native machine language and embedded in the interpreter[GR83]. In TS, a primitive method is defined using the register transfer language, and can be read and written from the browser. Thus, there are no black-box primitives in TS; *all* code can be inspected by the programmer. Of course, a programmer must know the register transfer language for this to be useful.

A message send can be implemented in one of three ways. The most efficient is in-line substitution. This can only be done when the class (or set of potential classes) of the receiver is known and should only be done when the method being invoked is small. If the class of the receiver is known but the method being invoked is large then a message send is implemented as a procedure call using native machine instructions. Finally, if the class of the receiver is not known, in-line caching[DS83] is used for method-lookup. In-line caching is implemented by procedure calls. Thus, there is no need for a "method lookup" primitive in the code generator, though there is a routine in the in-line caching system for that purpose.

All knowledge about run-time structures is isolated to the translation from parse trees to register transfer instructions. In particular, the translator from register transfer instructions to native machine instructions knows nothing about method-lookup or the format of method contexts. This probably makes the compiler less portable, since it means that primitives need to know the exact format of objects, whether the stack grows up or down, and where to find arguments. This problem has been partly overcome by adding macros to the register transfer language for accessing method arguments and returning results. We plan to try porting our compiler to another platform soon; this will undoubtedly reveal many more machine dependencies.

3 Types

Types in Typed Smalltalk are similar to sets of classes. To be precise:

> An *object type* is a class name together with a possibly empty list of types.
> A *type* is a non-empty set of object types.

An object type describes the type of a single object. The class name in the object type is the class of the object, while the list of types describes the types of the components of the object. For example, the type of an array whose elements are integers is *Array of: Integer.*[1] Array is the class name and the list of types consists of the single type *Integer.* The type *Integer* is a single object type with a class name Integer and an empty list of types.

The type of a variable or expression describes the objects that are possible values of the variable or expression. An example is *(Array of: Integer) + (Array of: Character) + (Array of: (Integer + Character)).* Here, the plus operator is read as 'or', so a variable of the above type could contain an Array whose elements are all Integers, all Characters, or a mixture of the two.

One feature of this type system is that type inclusion (i.e. type specificity) is exactly the same as the subset relationship on the sets of object types making up a type. Thus, an object of type *Integer* is a member of type *Integer + Character.*

This definition of type permits subtle distinctions between types. Consider the two types *Array of: (Integer + Character)* and *(Array of: Integer) + (Array of: Character).* Neither is a subset of the other—the first has one object type and the second has two, but neither of the two object types in the second type are equal to the object type in the first. Although these types seem similar, they are completely different. An object of type *Array of: Character* is not of type *Array of: (Integer + Character)* since it cannot have an Integer stored in it. A variable of type *Array of: (Integer + Character)* is not of type *(Array of: Integer) + (Array of: Character)* because it cannot be assigned something of type *(Array of: Integer).*

3.1 Type Declarations

As our examples of types illustrate, a class may be regarded as having certain *type parameters.* The list of types that an object type associates with a given class name C will contain one type for each type parameter of class C. These type parameters are declared when the class is created, and can be referred to in the methods of the class as if they were types. The following class definition declares OrderedCollection to be a subclass of SequenceableCollection with a type parameter *ElementType.*

```
SequenceableCollection subclass: #OrderedCollection
    instanceVariables: 'firstIndex ⟨SmallInteger⟩
                        lastIndex ⟨SmallInteger⟩'
    classVariables: ''
    typeParameters: 'ElementType'
    poolDictionaries: ''
    category: 'Sequenceable-Collections'
```

Note that the type of each instance variable is declared when the variable is declared and that angle brackets are used to set off types from the regular Smalltalk code. Type parameters like ElementType and classes like SmallInteger can be used as types. Other classes, such as SequenceableCollection, return types when sent the of: message, the of:of: message, and so on.

The range of a type parameter may be restricted by including an optional range declaration. The range declaration of a type parameter is syntactically identical to the type declaration for a variable. In the above example, if we wished to restrict the elements of the OrderedCollection to be characters, the typeParameters: field would be declared as 'ElementType ⟨Character⟩'.

Class variables and instance variables have their types declared when their defining class is created. However, method arguments, temporary variables, and block arguments all have their types declared in a method, as will be seen later.

A *signature type* is a type (i.e. a set of object types) specified by a set of message types. A message type is the type of a message, not an object, so it is not

[1] Types will be in italics. Class names and Smalltalk code will be in a sans serif font.

really a type. It consists of the name of the message, the type of the message's arguments, and the type of the value that the message returns. A object type T is in the signature type if, for each message type m in the specification of the signature type, a message of type m sent to an object of type T is type-correct.

A signature type can be thought to include object types belonging to classes not yet created. Under this interpretation, signature types contain an infinite number of object types. In fact, the type specified by an empty signature contains every possible object type. Since signature types specify the largest possible types, procedures that use them exhibit the most polymorphism and so are the most flexible. However, use of signature types prevents the compiler from performing some important kinds of optimizations, since they do not provide enough information about the class of the receiver to allow compile-time binding of message sends.

3.2 Type-checking

Type-checking infers the types of expressions and ensures that each statement is type-correct. An assignment statement is type-correct if the type of the expression on the right-hand side (viewed as a set of object types) is a subset of the type of the variable on the left-hand side. A return statement is type-correct if the type of its expression is a subset of the return type of the method. A statement sequence is type-correct if each statement in the sequence is type-correct.

Due to the simplicity of Smalltalk, the only remaining language construct is the message send. A message send is type-correct if it is type-correct for each possible class of the receiver. Let method(C, msg) denote the method invoked when message msg is sent to an object in class C. The sending of msg to an object in class C is type-correct if the message type describing the send *includes* the message type of method(C, msg). This will be true if there exists an assignment of types to type parameters (type variables) such that the type of each argument of the message send is a subset of the type of the corresponding argument of the message type of method(C, msg). Finding this assignment requires unification. The return type of a message send to an object in a particular object type is the return type of the corresponding typed message with each type parameter replaced by the type that was assigned to it in the previous matching. The return type of a message send is the union of the return types of the message sends for each component object type of the receiver.

Type-checking with signature types is easy. No finite union of object types contains a includes type, while a signature type includes a type T if every message in the signature is type correct for each object type in T. A message is type correct for a signature type if it matches the corresponding method in the signature.

Type-checking is actually more complicated than this. TS uses case analysis to type-check some methods, and the programmer can use type-coercion when necessary. Details can be found in [JG87].

4 A Simple Example

We will illustrate the way the compiler works by showing how the method for max: in class Magnitude is compiled. The code for this method is

```
{ arguments: aMagnitude ⟨MagnitudeType⟩
  returnType: ⟨MagnitudeType⟩}

max: aMagnitude
    ↑ self < aMagnitude ifTrue: [aMagnitude]
                        ifFalse: [self]
```

This is identical to the original Smalltalk-80 method except for the type information prefix.

Method < in class Magnitude is known to return an object of type *True + False* (also called *BooleanType*), but it is actually implemented by subclasses of Magnitude, so its exact definition is not known. However, the ifTrue:ifFalse: message to its result can be converted into a case statement with procedure calls, as follows: [2]

```
t ← self < aMagnitude.
case
  t = true
    → ↑call True::ifTrue:ifFalse:([aMagnitude],[self]).
  t = false
    → ↑call False::ifTrue:ifFalse:([aMagnitude],[self]).
```

Each of the two procedure calls can be replaced by its definition. The ifTrue:ifFalse: message for class True evaluates its first argument, while the ifTrue:ifFalse: message for class False evaluates its second argument. Thus, the result is

```
t ← self < aMagnitude.
case
  t = true → ↑[aMagnitude] value.
  t = false → ↑[self] value.
```

Evaluating a constant block can be reduced at compile time to the expression in the block, so the method will end up as:

[2] The notation here is not legal Smalltalk, but a printable representation of the parse tree.

```
t ← self < aMagnitude.
case
  t = true → ↑aMagnitude.
  t = false → ↑self.
```

A particular use of max: is likely to be optimized by in-line substitution, providing further opportunities for optimization. For example, suppose x and y are declared to be of type *SmallInteger*. The expression x max: y would be converted into the expression

```
t ← x < y.
case
  t = true →y.
  t = false →x.
```

In this case, the < message can be further optimized because the class of the sender is known to be SmallInteger. The < method for class SmallInteger is a primitive method. Primitive methods are defined in TS by a register transfer language program. TS extends the Smalltalk-80 description of a primitive to <primitive: n <type> 'code'>, where n is the integer assigned to the primitive by Smalltalk-80, <type> is the type of the object that the primitive returns if it succeeds[3] and 'code' is the register transfer language program. Figure 1 shows the TS definition of the < method for SmallInteger.

The register transfer language uses the standard Smalltalk-80 syntax for variables, constants, and binary operators. Variables come in four flavors: address valued registers (a1, a2, ...), integer valued data registers (d1, d2, ...), byte valued data registers (b1, b2, ...), and special registers like the stack pointer (SP), a register holding the constant nil (NIL), and the condition code registers. The only constants are integers, though there are ways to access global variables and other objects using the current method's literal frame.

Expressions usually contain only one operator. These operators include the usual arithmetic and logical operators. The C dereference operator * is used to dereference a pointer. The instruction $\uparrow L$ jumps to label L and the instruction $z \uparrow L$ jumps to label L if the z bit is set, which normally means that the previous operation resulted in a zero.

The arguments to a method are given special names. Register $0 refers to the receiver. The n arguments to a method are named $1 through $\$n$. Register $\$n+1$ refers to the result to be returned to the sender. Using these special names makes it easier for the compiler to substitute a primitive in-line.

[3] Primitives can fail if the types of their arguments are incorrect—the Smalltalk code in the method is then executed.

```
{ arguments: aNumber <IntegerType>
  returnType: <BooleanType>}

< aNumber
    < primitive: 3 <BooleanType>
    'd1 ←$0.
    d2 ←$1.
    z ←(d2 & 16r80000000) = 0.  "check class of arg."
    ~z ↑l2                      "fail if wrong."
    d1 ←d1 {1:31}               "Convert self."
    d2 ←d2 {1:31}               "Convert aNumber."
    n ←(d1 − d2) < 0 .          "Test for less-than."
    n ↑l1.
    a1 ←NIL.
    a1 ←a1 + 12.                "Generate a false."
    $2 ←a1.                     "Return a false."
    @.
l1  a2 ←NIL.
    a2 ←a2 + 24.                "Generate a true."
    $2 ←a2.                     "Return a true."
    @.
l2'>
↑super < aNumber
```

Figure 1: Definition of primitive for SmallInteger <

The register transfer program defining SmallInteger < (Figure 1) first fetches the receiver and operand from the stack and stores them in d1 and d2. It then tests to see if the second argument is a SmallInteger. As in other Smalltalk implementations, SmallIntegers are flagged by having certain bits set. In this case, the high-order bit of a SmallInteger is 0. The operands are then converted to the machine representation for an integer and compared. The last half of the method returns a boolean object as a result. It depends on the fact that the objects nil, false, and true are located next to each other, so the addresses of the last two objects can be calculated from that of the first. This fact, like the representation of small integers, is specific to the Tektronix implementation of Smalltalk.

When the parse tree for x max: y is converted into a register transfer program, the primitive definition of < will be substituted in-line. The registers $0 and $1 in the primitive definition will be replaced by references to the receiver and argument of the method, in this case x and y. The resulting register transfer program will be optimized and converted into a machine language program similar to the one in Figure 2.

Note that the quality of the resulting code is pretty poor. In particular, there is no reason to generate a true or false object and then discard it immediately

```
00      move.l  a5,a0
02      move.l  (a5),d0             ;receiver in d0
04      move.l  (-8,a5),d1          ;argument in d1
08      move.l  d1,d2
0A      and.l   # 16r80000000,d2 ;Test if SmallInteger.
10      tst.l   d2
12      bne     3A
16      bfext   d0{1:31},d0
1A      bfext   d1{1:31},d1
1E      cmp.l   d1,d0
20      bmi     32                  ;test x<y
24      move.l  d5,a2               ;d5 + 12 is false
26      add.l   # 12,a2             ;return false
2C      move.l  a2,a1
2E      bra     3A
32      move.l  d5,a1               ;d5 + 24 is true
34      add.l   # 24,a1             ;return true
3A      move.l  d5,d0
3C      add.l   # 12,d0
42      cmp.l   d0,a1               ;test if true
44      bne     52
48      move.l  (a0),a2
4C      move.l  a2,a6               ;a6 will hold result
4E      bra     5A
52      move.l  (-8,a0),a3
56      move.l  a3,a6
5A
```

Figure 2: Assembly language version of x max: y

thereafter. Also, the argument is declared to be a small integer, so the first test is unnecessary. There are many optimizations of this kind that are needed. In spite of lacking many optimizations, the resulting code is much faster than the best interpreters, as will be seen in the next section.

5 Performance Evaluation

A compiler needs to produce fast code and to produce it quickly. TS is currently speeding up small examples by a factor of 5 to 10 over the interpreter, but it takes 15 to 30 seconds to compile them. Both of these figures are certain to improve. The previous example shows that better optimizations should be able to make the resulting code several times faster than it is now. The code generator has several bottlenecks. Rewriting them should make TS two or three times faster and compiling TS should then make it about as fast as the current Smalltalk-80 compiler.

The following benchmarks compare TS with Quicktalk. It is impossible to compare absolute times because the machines we used are several times faster than the ones used in [CW86]. Even the speedups are hard to compare, since the Tektronix interpreter has been rewritten and made faster since the earlier paper. However, the results show that TS produces code that is about as fast as that of Quicktalk.

The first example is the sumFrom:to: method of SmallInteger. Since TS is integrated within the standard Smalltalk-80 programming environment, we used the same source code for the compiled and the interpreted time. 0 sumFrom: 1 to: 10000 took 62 milliseconds in TS and 829 milliseconds in the interpreter, for a total speedup of 13. Quicktalk achieved a speedup of 22 on this example.

The second example is the substring replacement method replaceFrom:to:with:startingAt: in class String. TS achieved a speedup of 5.5 over the interpreter when replacing substrings of length 1000. This compares favorably with Quicktalk, which had a speedup of 3.3, but is still much slower than a handwritten machine language program, for the same reasons given in [CW86].

The third example is dot product. TS provided a speedup of 6.7 whereas Quicktalk provided a speedup of 5.0.

All of these example are small. The largest example that TS has been able to compile so far is addAll:. We compiled a test method that added one set of integers to another. The addAll: method was substituted inline and specialized for sets and for integers. The resulting program was from 5.7 to 7.2 times faster than

the interpreter, with the greatest speedup occuring with the fewest collisions in the hash table of the set. Sets and dictionaries are used a lot in Smalltalk, and both are based on hashing. Thus, this example indicates that TS should be able to make large programs quite a bit faster.

6 Programming Environment Concerns

Smalltalk-80 provides an extremely attractive programming environment [Gol84]. The user can incrementally construct and test a program, changing even the classes that provide the programming environment. An optimized method has assumptions about other methods encoded within it, so changes to the other methods can make the optimized method incorrect. Thus, optimization does not integrate easily into the kind of programming environment provided by Smalltalk-80.

The most important optimization, in-line substitution, is also the optimization that causes the most problems. If method *A* is substituted into method *B* then any change to *A* will require the recompilation of *B*. Thus, in-line substitution creates dependencies between methods.

There are several ways to handle these dependencies. One way is to keep track of the dependents of each method and to recompile a method's dependents when it is changed. However, nearly every method depends on some other method, so it will take a great deal of space to store all these dependencies. Most of the dependencies will be on standard methods with little chance of changing, such as ifTrue:ifFalse:. Therefore, this approach will waste a lot of space storing dependency information that is never used.

Another solution is that recompiling a method causes all other methods to be checked to see whether they need to be recompiled. This will waste a lot of time, since most methods being recompiled have no dependents.

Our solution is to have several kinds of methods. A *fixed* method is not expected to change, so no dependency information is kept for it. Before it can be changed it must be converted to a *changeable* method. Converting a fixed method to be changeable requires scanning all other methods to see which depend on it. Converting a method from changeable to fixed will delete the dependency information kept for it. Thus, we can make recompilation fast and minimize the dependency information required.

It is very important that the Smalltalk compiler be fast. Smalltalk programmers are used to the compiler taking no more than a second or two per method. Moreover, the compiler is used as a command interpreter. The compiler can be a fast command interpreter by having it do few optimizations. However, changing one method can cause an arbitrary number of other methods to be recompiled, so even a fast compiler will take a long time in some cases.

We make changing a method fast by keeping an unoptimized version of every method that depends on some other method. If the dependent method is told that its optimizations are invalid then it will use the unoptimized version of itself until it can be reoptimized. Instead of optimizing a method immediately, all optimization takes place in the background. Thus, when a method's optimizations are invalidated, it will place itself on the optimizer's queue. Invalidating a method is 10 to 100 times as fast as reoptimizing it, so this technique greatly improves the response time of the compiler.[Whi87]

One of the problems with performing in-line substitution is that the compiler needs to be able to determine which methods should be substituted. In general, rarely executed methods do not need to be optimized at all. Short methods should usually be substituted in-line, while long methods should not be. We have not yet tried to automate the decision of which methods should be called by in-line substitution and which by a procedure call. Instead, the programmer marks methods as substitutable or nonsubstitutable, and the compiler follows the advice.

7 Project Organization

Proponents of object-oriented programming claim it greatly improves reuse of code and decreases the amount of time needed to develop software. This project is evidence in support of these claims. The compiler was written by six people. Although the first code for the type system was written in December of 1985, most of the work on the compiler took place after September of 1986. Indeed, four people on the project did not even know Smalltalk at the beginning of September 1986. Somewhere between 2 and 3 man-years have been spent on this project, mostly by students who did not previously know Smalltalk nor had done much work on compilers.

We were able to reuse most of the parser, most of the parse-node class hierarchy, and the user interface tools. This not only saved an enormous amount of work, but provided a common framework for the project. The reused components provided standard interfaces between the people working on the project.

The project was divided into one person each working on

- the type system
- integrating the type system into the compiler and programming environment
- high level optimizations on parse trees
- optimizing register transfer instructions and producing machine code
- maintaining dependencies
- building an interface to the virtual machine.

The most important shared interface was that of the parse trees, which was reused from Smalltalk-80. The type system was also an important shared interface. Although the type checking algorithms were reimplemented several times, the interface remained stable. The register transfer language was also an important interface, and changes to it caused other code to be rewritten.

Since many of the students taking part in the project are doing so as part of a M.S. program, there is a lot of turnover of project members. In fact, three of the original students have been replaced by three others. We have stressed clean, understandable design to minimize the difficulty of learning the system, but a new generation of students quickly learns which parts of the system are well designed and which need more work.

8 Further Plans

Although the compiler is becoming more reliable, it has not yet compiled the entire Smalltalk-80 image. We plan to do that and quit using the interpreter. We are rewriting register allocation, adding support for foreign functions, making it possible to provide specialized method look-up routines, and providing support for compiling applications to run outside the Smalltalk programming environment. Long-range problems that are being investigated are that of providing the usual Smalltalk debugger for optimized code, type inference, and allowing typed and untyped code to coexist safely.

The compiler only performs a few optimizations, so it is just a skeleton of an optimizing compiler. However, because it converts Smalltalk programs into an intermediate form that is very similar to C or Pascal, it can be fleshed out to perform most standard optimizations. Smalltalk should eventually be as efficient as other languages.

A type system is a prerequisite for optimizing Smalltalk. Fortunately, it is possible to design a type system for Smalltalk that is flexible enough to allow most Smalltalk programs to be type correct, yet restrictive enough to allow the compiler to optimize the programs. While it is not yet clear that all the advantages of the Smalltalk environment can be preserved, it should be possible to make Smalltalk as fast as any other language.

Acknowledgements

The authors are grateful to Tektronix for the donation of equipment, financial support and advice, to AT&T for financial support under the Illinois Software Engineering Program, to NSF for support under grant CCR-8715752, and to Joe Loyall, John Wiegand, and James Whitledge for doing much of the programming.

References

[Atk86] Robert G. Atkinson. Hurricane: an optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 151–166, November 1986. printed as SIGPLAN Notices, 21(11).

[BMW86] Mark B. Ballard, David Maier, and Allen Wirfs-Brock. QUICKTALK: a Smalltalk-80 dialect for defining primitive methods. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 140–150, November 1986. printed as SIGPLAN Notices, 21(11).

[CW86] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 119–130, November 1986. printed as SIGPLAN Notices, 21(11).

[DF80] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[DS83] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record*

of the Tenth Annual ACM Symposium on Principles of Programming Languages, pages 297–302, 1983.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Gra86] Justin Graver. *Adding Type Specification and Type-Checking Capabilities to Smalltalk-80.* Master's thesis, University of Illinois at Urbana-Champaign, 1986.

[JG87] Ralph E. Johnson and Justin O. Graver. *A User's Guide to Typed Smalltalk.* Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield, Urbana, Illinois, 1987.

[Joh86] Ralph E. Johnson. Type-checking Smalltalk. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 315–321, November 1986. printed as SIGPLAN Notices, 21(11).

[Loy88] Joseph Loyall. *High-level Optimization in a Typed Smalltalk Compiler.* Master's thesis, University of Illinois at Urbana-Champaign, 1988.

[SUH86] A. Dain Samples, David Ungar, and Paul Hilfinger. SOAR: Smalltalk without bytecodes. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages and Applications*, pages 107–118, November 1986. printed as SIGPLAN Notices, 21(11).

[Whi87] James Robert Whitledge. *An Interface for an Optimizer in the Highly Interactive Environment of Smalltalk.* Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[Wie87] John David Wiegand. *An Object-oriented Code Optimizer and Generator.* Master's thesis, University of Illinois, Urbana-Champaign, 1987.