A Typing System for an Optimizing Multiple-Backend Tcl Compiler

Forest Rouse and Wayne Christopher
ICEM CFD Engineering
Berkeley, CA

A Typing System for an Optimizing Multiple-Backend Tcl Compiler

Forest Rouse

ICEM CFD Engineering,
Berkeley, CA 94705.

Wayne Christopher

ICEM CFD Engineering,
Berkeley, CA 94705.

Abstract

This paper describes the typing system used by the ICE 2.0 Tcl compiler[Rouse]. The typing system tracks the usage of variables and allows the compiler to reduce the number of instructions required to carry out Tcl instructions. It will also allow future improvements in code emission by making it possible to carry out standard compiler analyses and optimizations[Aho].

1 Introduction

Tcl has become a popular scripting language[Ousterhout]. In connection with the rise of the internet as a popular medium for information transfer, there is a high probability that it will continue to grow in popularity. One of the advantages of Tcl as a scripting language is the lack of explicit typing. Everything in Tcl is a string, therefore the complexity of both the programs written in Tcl and learning the language itself are reduced dramatically. Unfortunately, the simplicity of "everything is a string" makes optimizing compilation extremely difficult.
Additionally, the semantics of the language are not formally defined but rather given by the implentations of the various commands. Hence, different commands can and do treat language elements differently. For instance, the command

```
set a(1) 3
```

treats the variable `a(1)` differently than the command

```
foo $a(1)
```

The variable `a(1)` is parsed differently in the two cases. There is no difference between the two methods of parsing the variable name most of the time, but in cases like

```
foo $a(this is an example)
```

it does.

Finally, Tcl commands can cause non-local side-effects on variables. The act of placing a trace on a variable and the actions on upvar and uplevel variables can cause any cached knowledge about the variable's value type to become invalid. Any typing system must be able to handle all possible non-local effects.

Early tests with the ICE 1.0 Tcl to C compiler reveals some interesting statistics. We use a simple ASCII file format converter as a benchmark example. This converter has about 430 lines of commented Tcl code. The code has a high proportion of string manipulation and relatively few arithmetic manipulations.

The act of compiling control statements, recognizing Tcl words at compile time, and not copying argument strings to invoked procedures results in a factor of 3 speed-up. The final version of the 1.0 compiler has a factor of 7.5 speed-up on the same benchmark. The 8.0a1 version of the Sun byte compiler[Lewis] results in a factor of 4 speed-up. Hence, the current compilers (the Sun on-the-fly and the ICE 1.0-1.2 Tcl to C) to within a factor of two rely on parsing the statements once (recognition of Tcl words), and not copying strings when invoking procedures. We must go to other optimizing techniques to get larger speed-ups.

Placing constraints on the variable type and possible side-effects allows the compiler to avoid emitting code to carry out traces or unnecessary type conversions. It will allow the compiler to safely remove unnecessary code in the near future. This paper describes the typing system, the related system of "promises", and future optimizations based on the typing system.

1.1 Overview of Compiler

The ICE Tcl compiler is an ongoing commercial project started in 1994. It was first designed to emit only C code. The Tcl to C translation has proved useful to our customers because of the improved performance of the Tcl code along with the ability to hide their intellectual property. Obviously, shipping just the Tcl code opens the program up to both customers or anyone else with access to a computer system.

However, most Tcl users like the convience of rapid turn-around available using the Tcl interpreter. The ICE 2.0 compiler has been redesigned so that instead of emitting just C code, the compiler emits an intermediate language that can be translated to multiple backend languages. The first language that will be available will an enhanced Sun 8.0 byte code. This version is entering tests as of this writing. We expect that both Java and C backends will be available by September of this year.

Users can "mix–and–match" emitters to tailor their application. That is, users can have specific procs compiled to "C" while others compile to any other possible backend including bytecode.

The compiler builds a parse tree. The nodes of the parse tree represent Tcl language elements. During synthesis, code representing the language element is emitted. The synthesis of both typing and overall script attributes are also directed by the node. This design makes it considerably easier to maintain and extend the compiler than the 1.0 version.

The principle upgrade to the 1.0 system has been the implementation of a C++ node to represent variables. Variables are tricky because different statements parse variables differently. This design allows all language elements to treat variables in the same way.

2 The Typing System

We can describe the typing system to first order as a set of attributes whose values at any point in the program is the current Tcl type that is associated with every recognized Tcl variable. This description is incomplete only because we must also follow all allowable side-effects. Hence, the type attribute includes whether or not the variable has a known type, locality (global, local, upvar, uplevel, or argument), whether or not the variable has been unset, whether or not the variable is possibly being traced, whether or not the variable is in a loop, and whether or not the variable is an array indexed with a constant string, an array indexed with a variable string, or is a run–time computed variable name, which we call a *dynamic* variable (as in `set $x 3`).

Essentially the action of the typing system is to properly synthesize these attributes after every state change of the system. We can avoid emitting type conversion statements if the variable is known and is of the correct type. Other optimizing actions can also be taken on some known variables. This reduces the number of statements required in a compiled Tcl program thereby reducing the execution time.

The difficulty comes from the fact that every Tcl statement must be analyzed to determine the possible side effects it might have.

Consider, for example, the following sequence:

```
proc a {} {
    set c 3
    unset c
    set x [list a b c d e]
    set y [list set unset]
    uplevel 1 {
        unset a
        [lindex $y 1] [lindex $x 1]
    }
}
```

This sequence unsets variables "a", and "b" in the scope of the caller, and the variable c in the scope of proc a. Trival analysis is required for understanding the effects on the variable c. Calling procedures must analyze the effect of the proc to determine the effect of unsetting "a". Finally, to determine that the variable "b" is unset requires constant folding along with the calling procedure determining the effect of unsetting "b".

The current type analysis system in the ICE 2.0 compiler currently carries out analyses in the local scope of the procedures. Recursive typing, or the ability of the caller to analyze the effects of a procedure on variables in its own local scope and constant folding through variables are currently being tested. In lieu of recursive typing and constant folding, the current type system states whether a given procedure unsets remote variables. This invalidates the variable cache in all calling procedures. Any read or write of the variable subsequent to the invalidation of the cache will require that the variable will be read via the standard procedure "LookupVar".

All Tcl statements that have an unanalyzable effect invalidate the cache in similar way. All commands of the form "$x a b c..." (both on command lines and in control statements), eval statements, and variable traces can cause the variable cache to be invalidated. Users can make "promises" – compiler directives that limit the effect of unanalyzable statements. These compiler directives are not suitable for every Tcl program. For example, a user cannot make a promise about statements computed at run–time in a Tcl script that calls the following procedure:

```
proc a {x} {
    uplevel {
        set c "unset $x"
        eval $c
    }
}
```

The reason is that the "eval" statement has the side effect of invalidating the variables in the calling scope.

2.1 Promises

The typing system is conservative. The worst case scenario is used to synthesize attributes for unanalyzable statements. Users can constrain the effect of unanalyzable statements thorough the use of "promises". Promises can be made about the possible side effects of

- catch statements with variable bodies,
- dynamic statements (`$x a b c ..`),
- eval statements and control statements with variable bodies,
- trace procedures (effect of variable traces), and
- uplevel statements with variable bodies.

Statements can be deemed "safe", "nounset", and/or "notrace". Users invoke compiler directives on the command line as in

```
Tcl_compiler -filescript -safe trace foo.Tcl
```

If the compiler directive is invoked in this manner, the directive applies to the entire Tcl script in the file foo.Tcl. Additionally users can also direct that a specific procedure scope be compiled with a specific promise as in

```
proc -safe trace a {a b c ...} {
  <proc body>
}
```

In this case, the "safe trace" directive applies to procedure a and all procedures defined within its scope.

The "safe" promise directs the compiler to assume that no statement in that category unsets a variable, changes the type of the variable, or establishes an "unsafe" trace on a variable. The "nounset" promise just directs the compiler to assume that no statement in that category unsets a variable. The "notrace" promise directs the compiler to assume that no statement in that category establishes a trace on the variable.

The promise that has the largest effect on any program is the promise that variable traces are "safe". Since every variable (including variables local to a proc), the compiler must assume the worst about every variable reference. The compiler analyzes if a variable local to a proc has a trace on it. But all non-local variables potentially have a trace placed on the variable.

Consider the following code:

```
proc a {} {
    trace variable a r trace_a
    proc trace_a {name element op} {
        set $name "foo"
    }
    b
}
proc b {} {
    uplevel 1 {
        set a 3
        incr a
    }
}
```

When compiling procedure b, seemingly the variable "a" in procedure b could be typed as an integer and a string after the `set a 3` statement. Unfortunately, the following incr statement will invoke a read trace that will cause the variable "a" to be changed to the string "foo". The type of "a" as an integer must instead be changed to unknown before we compile `incr a`. The following is the code emitted for the incr statement in procedure b:

```
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "read");
if (__tempReturn_i != TCL_OK)
                    goto __tempLabel_1;
__tempReturn_i = Tclc_CheckVar(
    interp, a, "a", NULL);
if (__tempReturn_i != TCL_OK)
                    goto __tempLabel_1;
__tempDummy_i =
    Tcl_ConvertUnknownToNumeric(
        interp, &a);
if (!(a->varAttr.typeAttr&TCL_TYPE_INTEGER)) {
    __tempReturn_i =
        Tcl_ConvertFromUnknownToDString(
```

```
            interp, &a);
    if (__tempReturn_i != TCL_OK)
                    goto __tempLabel_1;
    Tclc_ErrIncrVar(interp,
        a->value.allValue.dString.string);
    __tempReturn_i = TCL_ERROR;
    goto __tempLabel_1;
}
TCL_ONLYVALIDVAR(a, 2);
a->value.allValue.integer += 1;
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "set");
```

All of the code prior to `a->value.allValue.integer += 1;` is to assure the validity of "a" and that the variable can be promoted to an integer.
However, if the read or write trace does not modify the type of the variable as in

```
proc a {} {
    trace variable a r trace_a
    proc trace_a {name element op} {
        global c
        lappend c "$name $element $op"
    }
    b
}
proc b {} {
    uplevel 1 {
        set a 3
        incr a
    }
}
```

the script can be compiled with the promise of "safe trace". The compiler will instead know the type of "a" as an integer is still valid. The following code is emitted for the incr statement in proc b:

```
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "read");
if (__tempReturn_i != TCL_OK)
    goto __tempLabel_1;
TCL_ONLYVALIDVAR(a, 2);
a->value.allValue.integer += 1;
__tempReturn_i = Tclc_InvokeTraces(
    interp, a, "a", NULL, 0, "set");
```

A factor of three fewer lines of C code are emitted in this case. We eliminate the check on the validity of the variable "a", and the promotion of "a" to an integer and the checks that both operations succeed. That is, since we know any possible trace on the variable "a" cannot have a side effect that modifies the type or validity of "a", we can avoid the checks that assure the state of variable "a".

Test	ICE 1.0	ICE 2.0	Speedup(%)
Loop	102	68	50
10! (using loop)	98	76	28
Sum	1958	648	200
List	17651	10669	53
Reverse list	28443	21663	31

Table 1: Comparison of the execution speeds of C code produced by the Tcl 1.0 compiler and the Tcl 2.0 compiler on a variety of benchmarks. Times are in μ sec.

2.2 Tcl Lint

The in depth analysis of Tcl programs has led us to release "Tcl Lint". This program is the ICE 2.0 compiler with no code generator. The compiler with its ability to determine the possible side-effects of statements allows us to create error or warning messages when a variable is used before it is set. When there is no possibility for a variable to be set, an error is reported. If an unanalyzable statement is executed prior to variable usage, a warning message is generated if the proper user modifiable warning level is set.
Additionally, the compiler checks that usage of core functions and user defined procedures matches what is expected. Currently, we check that the number of arguments matches the number of arguments given, and for core functions, the usage of the function matches what is expected. Extensive analyses of catch, for, foreach, if, regexp, regsub, switch, and uplevel are carried out.
Tcl Lint can warn of unanalyzable statements. Statements like `uplevel $a` and `switch $a $b $c $d ...` can be pointed out to the user. Tcl Lint can also warn of procedures that use the implicit return mechanism that are expected to actually return a value. The compiler can slightly improve the code emitted if it knows that a procedure does not actually return a value. Finally, control commands with empty bodies are pointed out.

3 Current Status

Table 1 shows the speedups of the 2.0 compiler over the 1.0 compiler. The benchmarks were carried out on a Solaris 5.0 machine. We see that over a number of different types of benchmarks, the effect of

having a typing system is relatively modest. The speedups range from 25% to 50%. The benchmark of summing numbers between 1 to 1000 achieved factor of 3 speedup over the ICE 1.0 compiler is the exception to this rule.
There is also a decrease in the amount of emitted C code. We were able to emit a factor of 3.5 less C code, compile to a factor of 2.5 less object code and get a factor of 1.5 reduction in the executable size with our standard benchmark. This reduction occured in part to the fact we could use the promise of "safe trace" in the benchmark.

4 Future Plans

The typing system is a prerequisite to all forms of standard compiler optimizations. We must be able to analyze the side effects of moving or eliminating code before we can carry out the optimizations. Hence, we are now in the position to implement the following standard compiler optimizations:

- recursive typing,
- constant folding,
- common subexpression elimination,
- loop strength reduction, and
- code motion.

The effect of these optimizations will be script dependent.
We expect to have an early version of the 2.0 compiler with a bytecode emitter ready by the time of the conference. It will emit C code and an extended set of Sun bytecodes. The extended bytecode set will take advantage of our typing information. We hope to be able to present preliminary timing studies of emitted bytecode at the conference.
Finally, we are seriously thinking of using the compiler to emit Java bytecodes. One can regard the "C" code emitter as a model for all other static language emitters. The only real technical challange to emit Java, therefore, is a reasonable user usage model. We most certainly can emit bytecodes for the joint Java–Tcl interpreter and we believe that a suitably constrained pure Java port of many Tcl core functions could be written in a reasonable length of time.

5 Conclusion

We have described a multiple backend compiler for use in compiling Tcl. Additionally, it has a typing system that allows the compiler to infer all possible side effects of Tcl statements. This improves the code emission, speeds up the target code, and makes it possible to include standard compiler optimization techniques in near future to automatically improve current Tcl scripts.

6 Acknowledgments

Our thanks to OpenMarket Corporation who partially funded this work.

7 Availability

Both the ICE compiler and Tcl Lint are available for electronic downloading at

`ftp.dnai.com/users/i/icemcfd/tcl`

or visit our web site at

`http://www.icemcfd.com/tcl/ice.html`

Users can request a demo license by downloading, uncompressing, and untarring the software. Then just execute the command

`tcl_compiler -hostid`

and send the resulting information to

`tcl@icemcfd.com`

References

[Rouse] Forest Rouse and Wayne Christopher, "A Tcl To C Compiler", Proceedings of the 1995 Tcl/Tk Workshop (1995), Toronto, Ontario, Canada., July 1995

[Aho] Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman, *Compiler Principles, Techniques and Tools*, Addison-Wesley, 1988

[Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).

[Lewis] Brian T. Lewis, "An On-the-fly Bytecode Compiler for Tcl", Proceedings of the 1996 Tcl/Tk Workshop, Monterey, CA., July 1996