The State of TclQuadcode 2017

Kevin B. Kenny
Donal K. Fellows

The 'tclquadcode' system is a compiler, now under development for about three years, that translates a significant subset of Tcl to machine code. For the limited cases that it can handle, 'tclquadcode' produces significant gains with respect to the bytecode engine: 4-6-fold speedup in the typical case, and 30-100-fold speedup in the most advantageous cases. This talk presents recent work on speeding up the interface between compiled and interpreted code, avoiding memory allocation, and supporting non-local variable references. The speakers also intend to poll the audience informally on several topics where quadcode may drive developments in core Tcl.

1. Review: what is tclquadcode?

The ‘tclquadcode’ compiler is a *native-code* compiler for Tcl. That is, its output is executable machine code, not code for an abstract machine that is then processed by an interpreter. It is an *ahead-of-time* compiler, that is, it runs in advance of execution rather than generating code when a procedure is first executed. (At the current stage of development, it is too slow for just-in-time compilation.) It handles a limited subset of the language; for instance, at the moment it compiles procedures only (no TclOO methods, no λ terms, no global scripts). Nevertheless, the aspiration is that all language features will either be compilable or fall back on interpreted code. Similarly, the developers’ intent is that compiling code for performance should not depend on the user helping the compiler in ways like requiring variable type declarations. As far as possible, the compiler should deduce where it can generate efficient machine code while still preserving the ‘everything is a string’ convenience that Tcl programmers know and love.

The project is, at present, about forty-five thousand lines of Tcl code, augmented with just under 3000 lines of C++ code (plus about ten thousand lines of generated code), all of which layers atop the LLVM compiler infrastructure [LATT04]. It is, and will remain for some time, very much a work in progress. Nevertheless, the authors belive that at this point, significant programs can be ported over to it and gain the benefits of C-like execution speed while still retaining a Tcl flavour.

2. History of the project

Tcl’s performance has been known to be a problem for quite a few years, and the bytecode interpreter (which represented order-of-magnitude improvements over direct interpretation) was the headline feature of Tcl 8.0 in 1998. Bytecoding has improved much in recent years, but we are now clearly at a performance wall. The bytecode engine is a delicate piece of code, where any change tends to make programs go slower. It’s horrible to maintain (the code is a maze of GOTO statements), and it’s close to the achievable limit. Conceivably, a new interpreter (and perhaps a new bytecode language) could double or triple the speed, but the further order of magnitude that we would like to see really requires compilation down to machine code.

A number of events combined in the autumn of 2013 to trigger the start of the ‘tclquadcode’ project. A Google Summer of Code project had produced a Tcl bytecode assembler [UGUR10], which not only demonstrated that it was possible to work with bytecode at a high level, but more importantly, showed that it was possible to analyze bytecode and ensure execution safety. The assembler does not yield code that crashes the interpreter; all errors are reported as Tcl errors. Nevertheless, hand tuning can get 30-40% speed improvements relative to the code that the Tcl front end generates. At the same time, several developers had been experimenting with adding Tcl front ends to such embeddable compilers as tcc [TCC17] and llvm [DECO17]. It was therefore possible both to analyze bytecode, and to produce code for a compiler backend, without needing to leave the Tcl programming environment or hack the Tcl code. Finally, Karl Lehenbauer of FlightAware had in 2012 proposed a series of bounties for improving Tcl/Tk, one of which was a

substantial sum of money for a tenfold performance improvement on his benchmarks. While the authors' chief motivation is the improvement of Tcl/Tk for our own use, the bounty proved that there is enough community interest that the project is worth pursuing.

Accordingly, discussion started in earnest at the 2013 Tcl conference and in online media shortly thereafter, with key ideas coming from the authors, Andreas Kupries, Miguel Sofer, Don Porter, and Jos Decoster. Through the rest of 2013, and on into 2014, several preliminary studies took place. One of us (Kenny) developed a translation of Tcl bytecode into an intermediate language of our on invention (called quadcode, because initially, all the instructions were four-element lists) and an embeddable compiler for the Datalog language in Tcl, used to prototype the complex analyses required to analyze Tcl well enough to compile it. Working to the evolving definition of quadcode, Fellows was able to translate it into the LLVM intermediate language.

At the 2014 Tcl/Tk conference, we were able to report on the Datalog language. Several examples in the talk served as a "back door" announcement of the 'quadcode' project, and the two of us spent the weekend after the conference in a room together, integrating the Tcl/Datalog front end with the LLVM back end. We were able at the end to demonstrate our first trivial test procedure: a simple loop that given N, computes the Nth Fibonacci number.

Most of 2015 passed in adding language features one by one to this code base. While this was going on, we also were working on speeding up the front end by going to purpose-built data structures and eliminating Datalog, and on interprocedural data type analysis. The last is needed because we can generate much better code if we know the types of procedure arguments, and in Tcl the types are not necessarily the same at all places in the program where a procedure is called. We were able to present this work in the autumn of 2015 at the Tcl/Tk conference [FELL15], and thereby to announce the project formally.

2016 was, alas, a slow year for the project, spent chiefly in consolidating the gains already achieved (while both developers were busy with other work). In January of 2017, however, the pace picked up again. Gains have been achieved this year in node splitting (which will be discussed in the next section), in the handling of non-local variables (global and namespace variables, and variables imported via `[upvar]`– to be discussed in Section 4), and in additional language feature support and performance gains. The remainder of this paper discusses these recent developments.

3. Node splitting

In the course of attempting to achieve a goal of tenfold performance improvement that FlightAware had set, we discovered a significant performance issue with the first FlightAware benchmark, which tested numeric operations. The problem was with the simple forward type analysis that we perform (in which we tag variables with a given type when they are either constants of that type, or have been checked at runtime to be of that type). This worked remarkably well for simple examples such as:

```
proc x {} {
    set y 0
    for {set i 0} {$i <= 10} {incr i} {
        incr y $i
    }
    return $y
}
```

The control flow and type analysis for this procedure is shown in Figure 1.

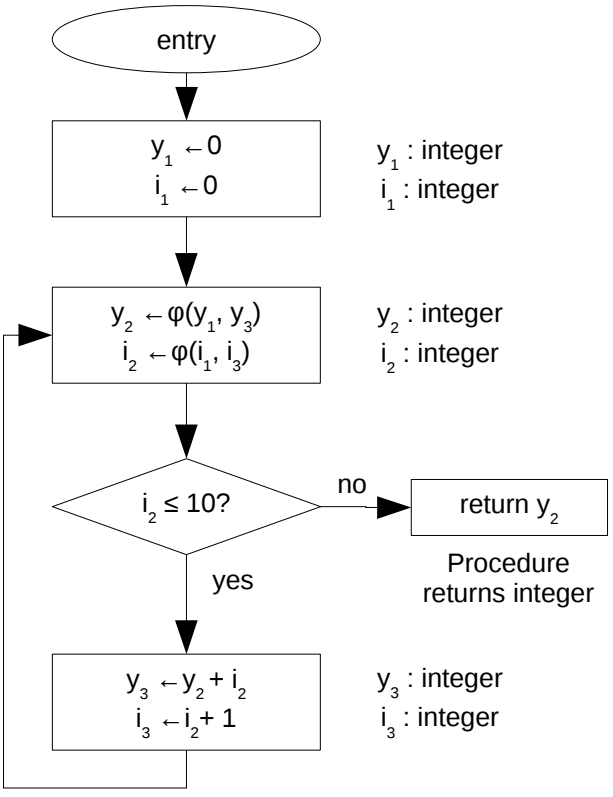

Figure 1. Simple type analysis

In the figure, the code has been converted to Static Single Assignment (SSA) form [CYTR91]. The φ is a pseudo-function that selects a value according to the code path by which a program reaches a junction point.

Unfortunately, when dealing of different types, for instance, the strings passed in from interpreted code, the type inference is not nearly so fortunate. Let's try replacing the loop starting value with a procedure parameter:

```
proc x {a} {
  set y 0
  for {set i $a} {$i <= 10} {incr i} {
    incr y $i
  }
  return $y
}
```

The generated code explodes to the structure shown in Figure 2. Since the procedure is called from interpreted code, both parameters are strings. This is not ordinarily a problem, since we are capable of generating a data type 'impure integer' that will hold both a string representation and an integer internal representation, but this is done only after a check is made.

If we look at what happened in Figure 2, we see that the comparison is now a complicated one, because we're comparing a string value against the constant `"10"`. As experienced Tcl programmers know, this may be a string or numeric comparison, depending on the content of i, which is initially unknown. The comparison requires two Tcl_Obj's as input.

Next, i is checked again for whether it is numeric, and this time an error is thrown if it is not. Finally, it is promoted to a native numeric value, accumulated into y, and incremented. The result of the increment is an integer. But this integer must immediately be discarded! It is about to rejoin a code path (represented by the φ operation) where a string is required, so the result object has to be repacked into a Tcl_Obj.

The result is that the code for the procedure is only a tiny bit faster than interpreted code; in the worst case, it may be slower because the memory management of the intermediate values is not as tightly optimized. Clearly, something better is needed.

The solution that we choose involves a technique that often goes by the name of *jump threading* or *loop peeling* [SONG02]. The idea is that the first iteration of a loop, which contains code that is problematic in some way, will be peeled off from the rest of the loop by splitting successive nodes in the control flow. In the case of type inefficiencies, a problem loop always seems to be flagged by the fact that one of its variables must be demoted to a weaker type (often a Tcl_Obj) at the bottom of the loop, as in the instruction shown in **bold text**. It is guaranteed that if the loop that requires weakening is split, more information about data types will be available to one of the copies.

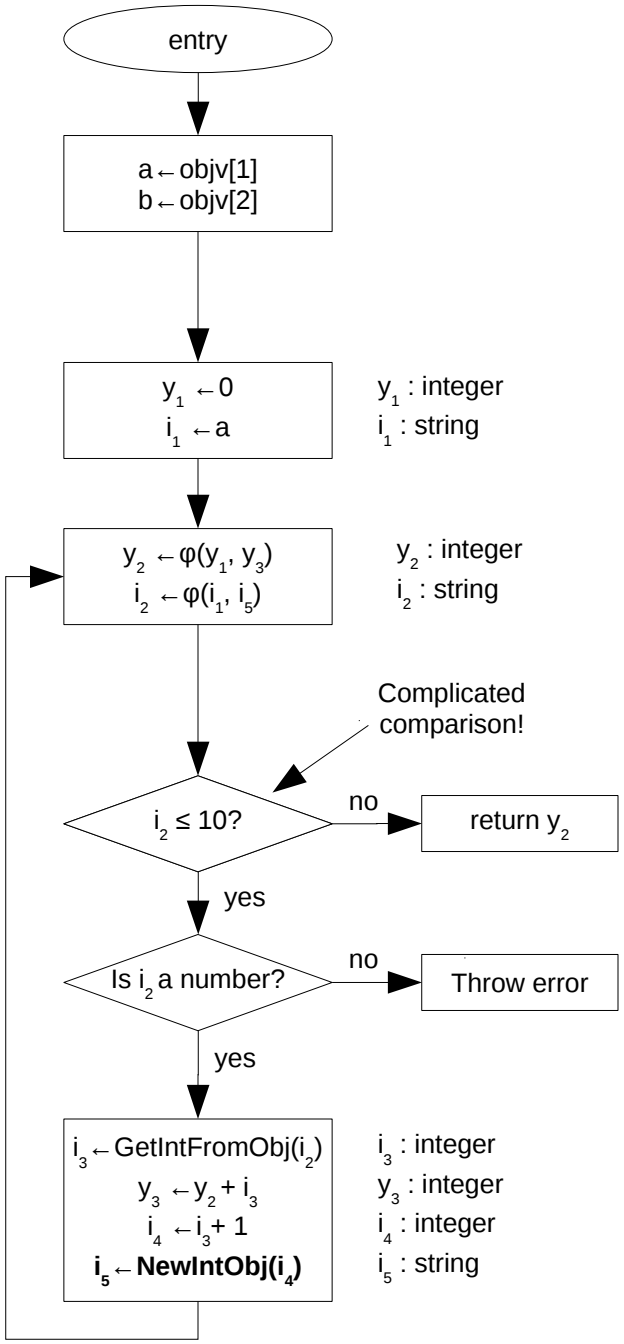

Figure 2. Inefficient type analysis

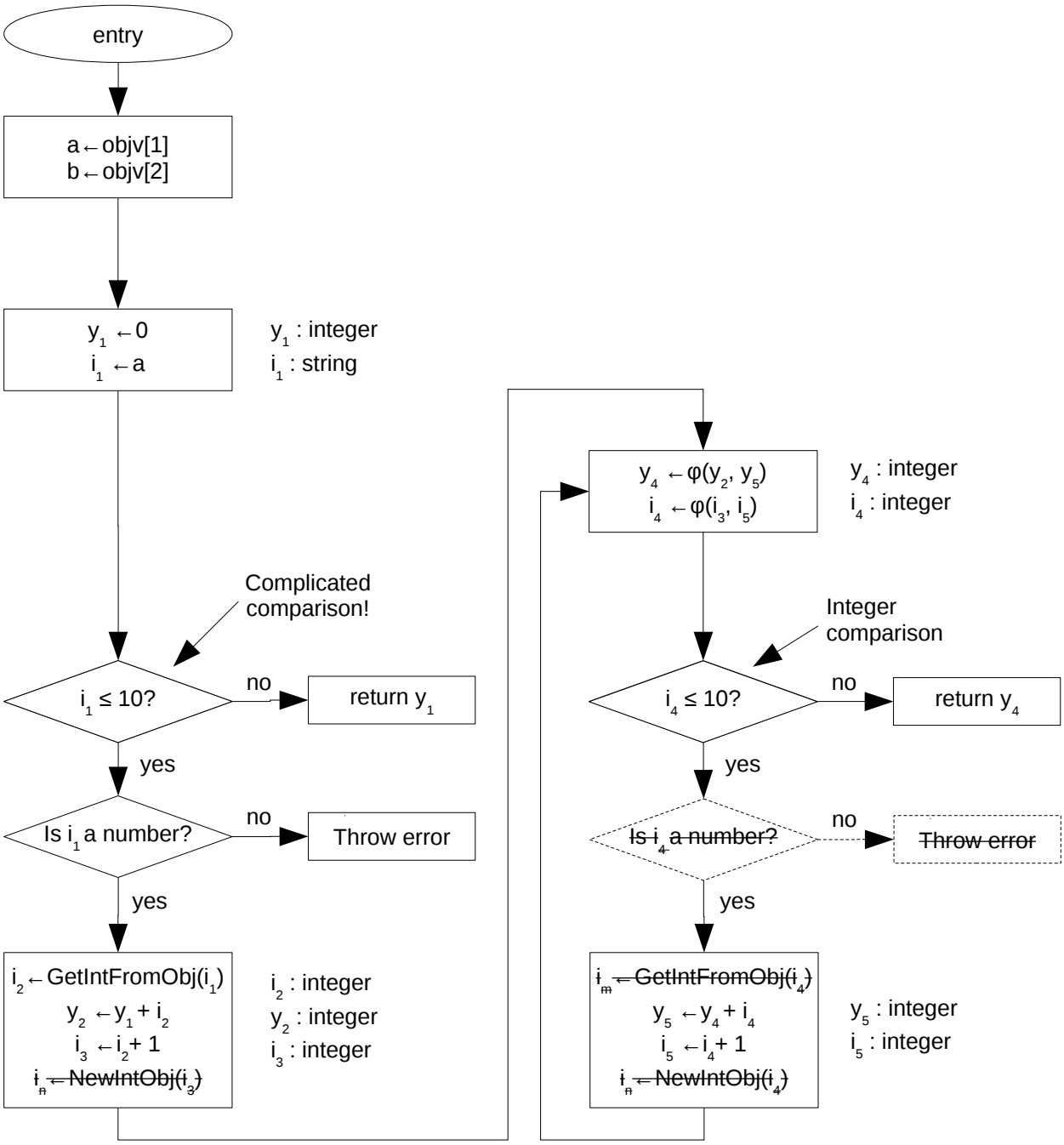

Figure 3. Result of node splitting on the problem loop of Figure 2.

The first iteration of the loop turns into straight-line code, and hence the φ operations may be removed. The remaining operations all start off knowing that i is an integer, and therefore there is no type checking or type coercion involved. The loop becomes a tight loop in machine code, performing only native integer arithmetic. The performance gain is phenomenal. One of FlightAware's numeric benchmarks, a simple loop solving equations in spherical trigonometry, increased in speed by a factor of 15-20 by this change alone.

4. Non-local variables

The next project that was taken on was access to variables that are not local to the procedure. There are essentially three ways that such a variable can appear:

1. It can be brought in as a global variable, using `[global]`, `[variable]`, `[namespace upvar]` or `[upvar #0]`.
2. It can be brought in from a caller's context with `[upvar]`.
3. It can be referenced directly using a qualified name (one that contains `::` namespace delimiters).

For the initial implementation, the compiler sacrifices some performance by declaring that all nonlocal variables that are aliased into the current procedure will be kept up to date in the call frame at all times. This decision means that the compiler will not have to worry about when to store a value, as opposed to simply keeping it in a register: nonlocal values are always stored. As time goes on and we have better understanding of code safety, we intend to revisit this decision.

The chief consequence of this decision on user code is that it becomes advantageous to defer writes to global variables whenever possible. The procedure:

```
proc accum {args} {
    global n; global s; global ss
    foreach a $args {
        incr n
        set s [expr {$s + $a}]
        set ss [expr {$ss + $a*$a}]
    }
}
```

will incur a performance penalty relative to a version that defers global variable updates to the very end:

```
proc accum {args} {
    global n; global s; global ss
    set n_ $n; set s_ $s; set ss_ $ss
    foreach a $args {
        incr n_
        set s_ [expr {$s_ + $a}]
        set ss_ [expr {$ss_ + $a*$a}]
    }
    set n $n_
    set s $s_
    set ss $ss_
}
```

There are two reasons that the second version is faster.

1. Nonlocal variables are Tcl_Obj's at all times, so in the first version, every assignment to one of the variables has to pack the value in a Tcl_Obj, incurring memory management overhead.
2. Nonlocal variables are all presumed possibly to be aliases of one another: we will discuss this problem below. The implication is that the compiler does not know, when it does `[incr n]`, that the statement will not affect the value of `s`. Consequently, it must generate code to re-fetch the value of `$s`, unpacking it from a Tcl_Obj and incurring type conversion overhead, prior to evaluating `[expr {$s + $a}]`.

Local variables are considerably less problematic. The only time that they must be refreshed is after invoking another command that may set their values. There is more detail available for invoked commands. The compiler at present keeps track of the following things that an invoked command might do:

- Read or write a variable in the calling procedure whose name is constant: `[upvar 1 a x]`
- Read or write a variable in the calling procedure whose name is passed on the command line: `[upvar 1 $argName x]`, with optimization for the common case where the caller passes a constant as the variable name.
- Read or write a variable in the calling procedure whose name is neither of the above cases (requires that all variables be in the callframe at the time the called command is invoked)
- Read or write a variable whose scope is not known to be in the calling procedure (`[upvar]` to an outer level, or to an unknown level)
- Read or write a global or namespace variable.

In addition, the analysis keeps track of whether a procedure is free of side effects (in which case, an invocation can be removed if the results are unused), and whether the procedure is also independent of the global state when invoked (in which case, multiple invocations with the same parameters can be collapsed into a single invocation).

5. Alias analysis – or the lack thereof

In all cases, the hard part of working with these variables is aliasing – the possibility that two different names might refer to the same variable.

In Tcl, aliasing among global variables is an intractable problem. Unless we have perfect knowledge of the entire program, it's possible for some [eval] somewhere to execute something along the lines of

```
uplevel #0 {upvar 0 a b}
```

and all of a sudden, `$a` and `$b` are the same variable. This possibility means that in uncontrolled code, any assignment to a global variable must be presumed to change any other global variables. The result is that type information for globals is lost – all must be presumed to be strings – and that global variable access entails a good deal of superfluous data motion, type checking and type conversion.

Similar problems exist with direct variable access, `$::path::to::variable`. In addition, direct variable access to non-fully-qualified names is not supported at the present time, because it is another thing that is impossible to analyze. The meaning of a partially qualified name depends on whether the given variable exists in both the procedure's own namespace and the global namespace at the time that the reference is used. Since code far remote from the procedure can create a new global variable at any time, even what variable the name refers to is not computable in advance.

The authors hope that this particular problem will be rectified in Tcl 9 by defining it away. In particular, TIP #278: "Fix Variable Name Resolution Quirks" [SOFE06] has languished long enough and deserves consideration if backward-incompatible changes are to be made to Tcl.

Because of potential aliasing issues, the performance of non-local variables is unacceptable to the authors of this paper at present, and will have to be addressed moving forward. There are a couple of options under consideration.

The first is to have the compiler generate two versions of the code, one with the maximally optimistic assumption that no two variables of distinct names are the same variable, and one with the current pessimistic assumption that all nonlocal variables are potential aliases of one another. An enumerated set of variable references in play in a given context will also be generated. At runtime, on entry into a context, the references will be checked to see whether all are distinct, and the appropriate code will be selected.

This first scheme allows for the common case where nothing is an alias, but does not give the programmer fine-grained control if the aliasing is more complicated. For this reason, another possibility is to introduce an aliasing assertion, using some command like:

```
tcl::pragma::noalias \
    {w x y} {x z} a b c
```

This command would assert that the variables `a`, `b`, `c`, `w`, `x`, `y`, `z` are all distinct, except that any of `w`, `x`, `y` may alias each other, or `x` and `z` may alias each other. In general, the command would take lists of variables, and assert that any two variables mentioned on the command line are distinct unless they are both members of at least one of the lists.

This assertion would actually be valuable to have in interpreted code as well. A procedure like the `[accum]` procedure above would do well to assert that `n`, `s`, `ss` are all distinct – while it will not crash if they are not, it surely will not generate the intended result of keeping the count, sum, and sum-of-squares of a variable.

It is also possible to imagine that the check could be enabled or disabled on a per-interpreter or per-namespace basis. This would allow code to proceed with unchecked alias assumptions. Violating the assumptions would not crash at runtime in the sense of nasty effects like segmentation faults, but merely generate incorrect answers owing to stale values being used for variables.[1]

The aliasing assertion has yet to be proposed as a formal Tcl Improvement Proposal because the design is still incomplete. In particular, we have not yet decided what the correct assumptions are with respect to the system variables `$::errorInfo` and `$::errorCode`, which are commonly set 'behind the program's back' when errors are caught. Surely there are very few Tcl programs out there that would survive having their globals aliased to one of these!

6. Summary of project status

The compiler is starting to be in a condition where it can compile a significant, albeit mostly static, subset of the Tcl language. It can invoke most builtin Tcl commands, can

1 The authors tend to avoid disabling assertions, believing Brian Kernighan's maxim that enabling assertions while developing and then disabling them in production is akin to wearing a parachute when a plane is on the ground and taking it off when the plane is in the air.

deal with variable references and `[upvar]`, and can provide significant speedups (well over tenfold in numeric code, a factor of 2-3 in 'typical' code, and perhaps none at all in string processing code, where Tcl has always excelled.)

A few significant deficiencies remain, and some of these will always be there.

First, there is no possibility of dynamic evaluation (using `[eval]`, `[uplevel]` or substitution on the first word of a command). Evaluating unknown code has unknown side effects, any of which might invalidate the compilation of already-compiled procedures. Moreover, this sort of restriction 'poisons' all the callers – once a procedure has unknown side effects, anything that calls it, anything that calls them, and so on, also has unknown side effects.

Fortunately, dynamic evaluation is generally at an outer level: inner loops that actually do the computational work, and are performance critical, generally don't use it. One important exception to this rule is user-defined control structures that use `[upvar 1]`. Work is in progress to handle this case, at least for the case where the script to be evaluated is a constant on the command line that invoked the procedure.

Similarly, traces are not yet supported, again owing to uncontrolled side effects. This is a potentially greater problem, and will probably need to be addressed by defining the semantics of compiled code in a restrictive way. If traces have side effects that would change the semantics of the compiled code, the side effects will not be honored. (That is to say, if your trace does something like redefine `[::set]`, you deserve whatever happens to you!)

Code containers other than procedures, such as the λ-forms used in `[apply]` and TclOO methods, are also not yet supported. The chief obstacles to this work are recognizing what strings are λ-forms (since the construction of the form is often remote from its application), and handling the custom variable resolver required in TclOO methods. The authors are optimistic that the necessary analyses are feasible, but this extension will represent a fair amount of work.

At present, array variables are not supported; when array references appear in code, they are replaced with dictionaries. This has been quite successful in practice – few programs that we've tried actually notice the difference – but will not be interoperable with legacy uncompiled code, and will have to be addressed.

The generated code is not yet aware of non-recursive evaluation (NRE), and hence cannot handle `[yield]` or `[yieldto]`. (It can appear in a coroutine, but an attempt to yield from it will give the dreaded 'C stack busy' error.) We have notes on how NRE might be handled, but implementation work has not progressed far.

7. Next steps

The relative priority to give to the issues raised in the last section is something of an open question, and we hope to gain some insight by interacting with the conference attendees. There are nevertheless some issues that are obviously immediate.

First, we need to tidy the code – notably its programming interface – so as to make it ready for an alpha release. At this point, we believe that even the limited implementation we have can be useful, and by actual use we'll gain experience in which deficiencies are most critical. In addition, the code needs to be reconfigured to produce a loadable DLL rather than loading into a running process. The result could conceivably replace the aging TDK Compiler, and would be a better approach to the problem. The machine code is considerably more obscure than Tcl bytecode (and the code that we generate doesn't much resemble the output of existing compilers for languages like C, so decompiling it would be rather a headache as well.)

We also need to experiment ourselves with how much of some large extant code base, perhaps Tcllib, the code is capable of handling, and the performance that it achieves. (The hope would be that the C extensions to Tcllib could eventually wither away in favor of compiled Tcl.)

Native Tcl array support, and the limited support for control structures based on `[uplevel]`, are also obvious next steps to expand quadcode's repertoire.

Beyond the issues mentioned above, there are also various minor improvements that we expect to make. Introducing more of Tcl's own data structures (such as lists, dicts, and bignums) into the compiled code would likely be a performance gain. Optimizing the compiler itself is also a priority, since at this point, the process of translating Tcl to machine code is painfully slow. Finally, as we carry out this work, there will likely be "spin-off" technology where

the Tcl Core could benefit from the compiler development. (`tcl::pragma`, alluded to in Section 5, would be one trivial example.)

Finally, the greatly improved numeric performance of the compiled code suggests that we should also investigate the incorporation of numeric extensions such as VecTcl [GOLL14] as a long-term effort.

References

[CYTR91] Cytron, Ron; Jeanne Ferrante; Barry K. Rosen; Mark N. Wegman; F. Kenneth Zadeck. “Efficiently computing static single assignment form and the control dependence graph.” *ACM Trans. on Programming Languages and Systems* **13**:4 (October, 1991) pp. 451-490.

[DECO17] Decoster, Jos. “llvmtcl” https://github.com/jdc8/llvmtcl, downloaded 23 September 2017.

[GOLL14] Gollwitzer, Christian. “EIAS and numerical math – introducing VecTcl” *Proc. 2014 European Tcl Symp. (EuroTcl 2014)*.

[FELL15] Fellows, Donal K, and Kevin B. Kenny. “The TclQuadcode Compiler” P*roc 22nd Annual Tcl/Tk Conf.*, Manassas, Va.: Tcl Community Association, 19-23 October 2015.

[KENN14] Kenny, Kevin. “Binary decision diagrams, relational algebra, and Datalog: deductive reasoning for Tcl.” *Proc. 21st Annual Tcl/Tk Conf.* Portland, Ore., Tcl Community Association, 10-14 November 2014.

[LATT04] Lattner, C., & Adve, V. (2004, March). “LLVM: A compilation framework for lifelong program analysis & transformation.” In *CGO 2004. International Symposium on Code Generation and Optimization, 2004.* (pp. 75-86). IEEE.

[SOFE06] Sofer, Miguel. “Fix variable name resolution quirks.” Tcl Improvement Proposal #278. Tcl Core Team: 2006, https://core.tcl.tk/tips/doc/trunk/tip/278.md.

[SONG02] “A technique for variable dependence loop peeling.” In *Proc. 5th Intl. Conf. on Algorithms and Architectures for Parallel Processing*. IEEE, Beijing, 2002. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.4406

[UGUR10] Ugurlu, Ozgur Dogan, and Kevin B. Kenny. “A bytecode assembler for Tcl.” *Proc. 17th Annual Tcl/Tk Conf.*, Oak Brook Terrace, Ill., Tcl Community Association, 11-15 October, 2010. https://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2010/.../dogeen.pdf.

[TCC17] “tcc4tcl” https://chiselapp.com/user/rkeene/repository/tcc4tcl/index, downloaded 23 September 2017.