Optimizing Tcl Bytecode

Donal Fellows
The University of Manchester / Tcl Core Team
donal.k.fellows@manchester.ac.uk

1. Abstract

The Tcl interpreter has an evaluation strategy of parsing a script into a sequence of commands, and compiling each of those commands into a sequence of bytecodes that will produce the result of the command. I have made a number of extensions to the scope of commands that are handled this way over the years, but in 2012 I started looking at a new way to do the compilation, with an aim to eventually creating an "interpreter" suitable for Tcl 9. This paper looks at the changes made (some of which are present in 8.6.0, and the rest of which will appear in 8.6.1) and the prospects for future directions.

1. Introduction

During the development of Tcl 8.6, Kevin Kenny and Ozgur Dogan Ugurlu demonstrated[1] (through the implementation of the command `::tcl::unsupported::assemble`) that it was possible to create an assembler for Tcl bytecodes that was sufficiently safe that it was suitable for exposure in a Safe Interpreter. In particular, it became clear that there were a set of constraints that could be applied that would ensure that a Tcl assembler would never generate code that could crash; access to parts of the stack not “owned” by the code was prohibited. Though infinite loops were still possible — excluding infinite loops requires either totally emasculating the capabilities of Tcl or the possibility of proving exceptionally complex mathematical theorems — those loops would never exceed properly calculable stack bounds.

I found this absolutely fascinating, as it showed that the bytecode that we had been generating was not nearly as unruly as I had previously feared; my problems with writing compilation commands such as for `switch` and `dict` had been more due to my not being aware of those implicit constraints, rather than their absence. It also allowed us to quantify exactly what was wrong with the compilation of `break` and `continue`, both of which had long been known to be problematic in cases previously only known in an operational sense.

When we considered the implications of this, we realized that it also made it substantially easier to consider compilation of Tcl to actual native code. Previous attempts[2] had focused on a simplistic transformation of the existing bytecodes to their machine-code equivalents, but that is a strategy that is unlikely to yield significant benefits for several reasons:

1. The bytecode that they are starting from is significantly non-optimal in the first place.
2. Tcl commands are potentially highly dynamic, with the option for their implementations to be changed substantially as a script executes.
3. The value-model of Tcl is very strongly rooted in the concept of an immutable reference with typed views, which is substantially different to that of machine code (mutable references to machine words) or languages like C (mutable references to typed variables).

The combination of these issues means that compilation of Tcl to machine code is a significant challenge. This paper will be primarily looking at dealing with the first part of this problem, so that the bytecode that the compilation starts from is at least a stronger foundation. The advantage of working on this part is that the results of doing this can be made available to the community more rapidly than the other parts; full compilation will require a lot more work to support than tweaking things to work better within current constraints.

In this paper, I present a summary of Tcl's bytecode system in Section 2. In Section 3, I describe how I have been improving the coverage of commands that bytecode is generated for. In Section 4, I describe the improvements I have made to the bytecode generated for a number of existing Tcl commands. In Section 5, I talk about an improvement to the introspection tools for Tcl bytecode so as to more simply expose the information that is there to scripts. In Section 6, I describe the simple optimizer that I have created for Tcl bytecode, and in Section 7 I present some performance measurements to examine whether I am making any progress on the performance front. Finally, I examine possible future directions in Section 8.

2. About Bytecode

Tcl's current system of compilation uses a custom target called Bytecode[3], designed primarily to be an in-memory and on-disk data structure[1] that has minimal space consumption as a primary goal.

The fundamental model of bytecode execution is that there is a stack of values that represent intermediate working values, arguments and results. Every command becomes a sequence of instructions that ends up with the result of the command being pushed on the stack; in the simplest case, a command is compiled into a push of all the argument words and an invoke via Tcl's basic command dispatch, which will in turn replace those argument words with the single result value. The pushing of the argument words may be non-trivial if the words are complex compounds of various substitutions, but the overall model is comparatively simple.

When this simple universal execution strategy is used, the command in question is referred to as *uncompiled*. This is obviously not actually true — we have just discussed what the compilation is! — but the key is that the actual embodiment of the semantics of the command is still the standard implementation function; all that is bytecode-compiled is the assembly of the arguments and the lookup and invocation of that function. With a "compiled" command, the semantics of the command are embodied by a direct sequence of bytecode instructions. Those instructions will produce the result of the command without (typically) going through command dispatch.

Command compilation

Each compiled command has its own strategy for producing instructions, the command compiler, which is asked to consider how to do the compilation in a particular case. Any failure of the command compiler will cause the standard compilation to be used, which is also used when the command name itself is dynamically generated (*e.g.*, the value of a variable or the result of a command), as at that point Tcl is unable to statically determine the command compiler to use.[2]

[1] Support for what became the TclPro compiler and the `tbcload` extension was part of the original mandate, though it is not part that is officially supported for general free use. However, the consequences of that support are subtly scattered through the code.

[2] It is a consequence of this that TclOO instance dispatch is unlikely to ever be compiled; a key usage pattern of TclOO is to hold the name of the instance to invoke in a variable, clearly a case that cannot ever be compiled to anything substantially better than the current dispatch mechanism.

Command compilers can fail for many reasons. One of the main reasons is if one of their arguments is not a literal despite the compiler requiring it to be; this is what happens when any of the arguments to while is not a literal. The other two common reasons for failure are if there is a lack of a local variable table (LVT) in the compilation context, or if given the wrong number of arguments, though this is very much not an exhaustive set.

The lack of an LVT case requires some explanation. The local variable table is a *numerically indexed* collection of variables that is used to hold the formal arguments to a procedure, the local variables inside that procedure, and whatever extra information is necessary (local temporary variables that hold values in patterns that would interfere with the stack). Some instructions for compiling commands only exist in a form that accesses the local variable table, so compilations that necessarily use those instructions cannot be done without an LVT present: this is exactly why foreach is not efficient except when used in a procedure (or other procedure-like entity, such as a lambda term or TclOO method).

Exposure of bytecode in Tcl scripts

There has been a disassembler for Tcl bytecodes since they were introduced in Tcl 8.0, but up to 8.4, this was one of the most hidden features (it required setting a magic variable and then reading standard output, itself tricky on Windows). In 8.5, this disassembler was exposed more cleanly via the disassemble command in the tcl::unsupported namespace.

In 8.6 this was joined by assemble, though the two commands shared very little in terms of actual syntax beyond the names of the instructions. In addition, the supported instruction sets were also subtly different, mainly due to it being hard to correctly and safely issue some instructions.

3. Improving Coverage

In order to improve the overall generation of bytecode by Tcl, it is necessary to increase the fraction of Tcl code that can be compiled to pure bytecode. That is, an instruction sequence is pure bytecode if it does not contain any of the instructions invokeStk1, invokeStk4 (commonly just referred to as invokeStk, as they are a linked pair) or invokeExpanded[3]. Script fragments that have their instruction sequences entirely free of those instructions are entirely predictable in their behaviour, at least at an operational/type-theoretic level.

But what was the status of Tcl's compilation of commands back in mid-2012? Well, there had been some adjustments done during the development of 8.6, but they had been rather piecemeal. After the introduction of compilation strategies for subst (in 2009), unset (in 2010) and dict with (in 2011), not much had really changed; the set of scripts that would be likely to become pure bytecode was indeed very small.

Improving key loop types

To change this, I instead took a different tack and looked at scripts where I wanted them to *become* pure bytecode. An example of the sort of script that I wanted to be pure was an inner loop of a simple value-generating coroutine. Such a coroutine is this one, which yields first its own name (often a useful thing to do), then each of its arguments, and finally it causes the receiving loop to break.

```
proc all args {
   yield [info coroutine]
   foreach item $args {
      yield $item
   }
   return -code break
}
```

[3] The other instruction that should not be present is invokeReplace, which I will discuss later in this paper.

This coroutine body procedure would be created and its values consumed something like this:

```
set c [coroutine X all "foo" "bar"]
while 1 {
   puts "X\[[incr i]\] = [$c]"
}
```

As you can see, all the commands in all are part of Tcl itself, and ones that are reasonably likely to occur in an inner loop. Furthermore, all the operations involve data that is available locally; it is all either immediately present on the stack, in the stack frame, or in the interpreter.

An alternative mechanism for doing such a simple yielding loop is this one:

```
proc all args {
   yield [info coroutine]
   foreach x [lrange $args 0 end-1] {
      yield $x
   }
   return [lindex $args end]
}
```

This has a different pattern of usage, or rather two slightly different patterns, one of which is done with info commands:

```
set c [coroutine X all "foo" "bar"]
while {[llength [info commands $c]]} {
   puts "X\[[incr i]\] = [$c]"
}
```

And the other with namespace which (equivalent to having the -command option specified):

```
set c [coroutine X all "foo" "bar"]
while {[namespace which $c] ne ""} {
   puts "X\[[incr i]\] = [$c]"
}
```

Making these as pure as practical (*i.e.*, the generating and receiving loops except for the necessary call to the coroutine itself, and — in this illustrative example — the call to puts to print the values) required being able to bytecode-compile both a way to actually produce values, yield, and a way to detect whether the coroutine had terminated from both within and outside the coroutine. Within the coroutine, it was a matter of making info coroutine be a compiled operation (interior termination detection is really just a matter of whether a non-yielding exit from the coroutine is performed, such as a return) and from outside the coroutine it was a matter of allowing for a way to query whether a particular command existed, which was done in different ways by different people: some used info commands with a literal non-pattern argument, and others used namespace which -command.

Similar concerns with determining whether a procedure call actually provided a value for some optional argument encouraged me to add info level to the compilation list. I also did namespace current and self object, as these are very common in some coding styles, while representing information that is readily available.

The compilations of all of these introspection commands are to straightforward instructions that implement the functionality; thus, the coroName instruction implements the functionality of info coroutine, resolveCmd implements namespace which, the yield instruction (unsurprisingly) implements the core of the yield command, etc.

Examining the Teapot

Yet for all that, I did not feel that I had made much of an impact in the coverage. I needed a different technique for selecting what commands would attract improvements. So I turned to the Tcl packages that I had installed in my local Teapot repository.

The vast majority of packages in the Teapot are either wholly or partially scripts, so this forms a substantial body of Tcl code that is in current use. By looking at this and finding what commands were common but uncompiled, I would at least establish a set of commands that *should* be compiled if possible and reasonable. Not all of them actually ought to be compiled (for example, there is no real benefit to compiling commands that do I/O) but it would at least establish some priority; very rarely used commands clearly need not attract significant effort.

In particular, I studied the core ensemble commands `array`, `dict`, `namespace` and `string`. (The vast majority of `info` and `interp` is only used on code that does not need to be fast, and `chan` is almost entirely focused on OS access.)

As can be seen in Table 1 and in more depth in Appendix 1 (the frequency counts date from October 2012), just how frequent the various subcommands are varies widely; `array set` is nearly 70 times more common than `array size`. In addition, some of the subcommands are largely impractical to implement: the operations that *read* the state of an array (`size`, `get` and `names`) have really rather strange trace behaviour, using privileged access to the implementation of the arrays to operate (the existence of elements has to be checked during the processing of those subcommands). But it does indicate that `array set` is a strong candidate for compilation.

Command	Incidence	Practical
`array size`	37	No
`array exists`	56	Yes
`array unset`	191	Yes
`array get`	479	No
`array names`	1085	No
`array set`	2511	Yes

Table 1: Incidence of array subcommands

Similarly, I have identified that the `first`, `last`, `map`, and `range` operations of `string` are practical, as are the `code`, `qualifiers` and `tail` operations of `namespace`, and the `merge` operation of `dict`. (Strictly, `string last` is not actually sufficiently frequent to be worthwhile, but its functionality is required to implement the more frequent `namespace qualifiers` operation.) Some of the *impractical* operations were `namespace eval` (very common, but crosses stack frame boundaries) and `string is` (potentially very relevant, but very complex).

Generating values

In addition, I looked for commands that could be used to produce literals. With the introduction of `lmap` and `dict map` in 8.6, we have increased the need to be able to produce a literal value as the result of evaluating a Tcl script. A few techniques were in common use:

- `expr` *`123`* — Fine for numbers, but poor where the value is non-numeric or (even worse) looks like a number but isn't.
- `subst` *`abc`* — A reasonable way of producing a simple value, but not actually commonly chosen.
- `format` *`abc`* — A way that works provided the literal being produced has no %-substitutions in it, but otherwise problematic. Also slow.
- `format "%s"` *`abc`* — Lacks the formal problems of the option immediately above, but still very slow.
- `list` *`abc`* — Not correct at all except for literal lists! This is particularly obvious when the literal being produced contains spaces.
- `return -level 0` *`abc`* — The "official" method, hardly used by anyone due to it being so thoroughly unobvious.

It was clear that it was desirable to make more of these operations efficient. In particular, wherever these commands produce a constant value or a value determined entirely by arguments, there is an excellent opportunity for generating the operation via efficient bytecodes. Since some of these were already bytecoded (`expr`, `subst`, `list` and that complex `return` form) what I was seeking to do was to ensure that the other potential common forms were efficient: in particular, where `format` has constant arguments or is only using simple string concatenation (literal pieces plus exactly `%s`) then it is entirely practical to bytecode them. The other forms of format remain uncompiled.

4. Improving Generation

But it is not just that the set of commands for which instructions are issued has been extended; I have also looked at improving the generation of bytecode for existing commands.

In particular, the very complex commands switch, try and dict with have had a lot of attention from me, as have the built-in ensembles *as generic features*. I have also worked on changing the break and continue compilers so that they jump to their target instruction locations (after doing appropriate clean-up) rather going to the expense of throwing exceptions.

Improving ensembles

One of the key features of Tcl's ensembles is that they can be bytecode compiled into. This is a key feature that distinguishes them from objects; they are command groupings, but the name of the command and its subcommand are expected to normally be literals in scripts. This is particularly true for those ensembles defined by Tcl itself. (Enabling compilation for all ensembles would have the unfortunate side effect of making the Snit extension enormously more expensive.)

The ensemble dispatch mechanism in Tcl 8.5 normally uses a cache in the subcommand's Tcl_Obj to hold a reference to the implementation command to dispatch to (building that cache from the ensemble's internal table if it does not exist or has been modified) and dispatches via Tcl_EvalObjv. This mechanism is fairly quick, but has some overhead relative to directly invoking the command. Where the ensemble is marked for compilation (via an internal flag not exposed to third-party code, used for string, info, etc.), the subcommand is a known literal, and the implementation command has a compilation implementation of its own, the compiler for that implementation is called with the rewritten argument list so that bytecode is generated. This means that we can use this mechanism for well known core Tcl ensembles with no degradation in performance.

However, the mechanism is not *that* fast where specialist bytecode compilers are not available. In particular, there are a number of steps that are relatively costly, such as the rewrite of the argument lists and the double dispatch (once to process the overall ensemble command, and a second time for the dispatch to the implementation), and a fair number of the subcommands that it is used with are relevant for use in high-performance code (*e.g.*, the string tolower command is really rather common when cleaning up external data).

To improve this situation, I have done two things. The first is to embed a new mechanism in Tcl (via the new bytecode instruction invokeReplace) to perform the efficient dispatch of ensemble subcommands. This embeds part of the mechanism described above; it reduces the overhead of the dispatch mechanism to the minimum (*i.e.*, a single call to Tcl_EvalObjv) while preserving the exact externally visible semantics that already existed. The second improvement is to add very simple command compilers to many (but not all) of the existing subcommand implementations; these simple compilers just check if the correct number of arguments is supplied (*i.e.*, that there will be no call to Tcl_WrongNumArgs at runtime, as that is one of the few functions that can observe the differences due to ensemble dispatch) and if the right number of arguments is present, issues a direct invokeStk instruction to call the relevant implementation command.

Following this change, we can now observe four different ways that ensemble subcommands can be dispatched:

1. Directly compiled subcommands just generate normal bytecode:

```
string range foo 2 3
```

Compiles to:

```
push "foo"
strrangeImm 2 3
```

2. Simple subcommands become invocations of the underlying implementation:

```
string tolower foo
```

Compiles to:

```
push "::tcl::string::tolower"
push "foo"
invokeStk1 2
```

3. Complex subcommands become the *replacing* invoke of the subcommand:

```
namespace eval ::foo { bar }
```

Compiles to:

```
push "namespace"
push "eval"
push "::foo"
push " bar "
push "::tcl::namespace::eval"
invokeReplace 4 2
```

4. Invocations of uncompiled ensembles use the old mechanism:

```
userEnsembleExample x y
```

Compiles to:

```
push "userEnsembleExample"
push "x"
push "y"
invokeStk1 3
```

The criteria for whether a subcommand is deemed to be "complex" are whether it evaluates a script during it's processing (as those scripts can contain a call to `info level` or `info frame`, both of which can observe the difference), or whether the subcommand has a non-trivial mechanism for determining if the correct number of arguments has been supplied (the `chan copy` command is one of these).

As previously noted, this mechanism is not enabled for ensembles created outside the core of Tcl. This is because the cost of deleting or doing an update of a compiled ensemble is substantial: it triggers the re-compilation of all bytecode in the interpreter. The impact on Snit in particular would be enormous, given that it is constructing an object system on top of the ensemble mechanism.

Improving switch

The **switch** command has a number of main modes of operation from the perspective of bytecode generation. There are three principal ones:

- *Jump table.* This is generated when doing exact matching of the argument, and is fast especially when the number of things to compare against is large (such as in the implementation of the `clock` command's format parser). This was already substantially correct, as it did not need to retain a copy of the value to test against for any length of time.
- *Sequence of conditions.* This is what is normally generated, and is what is used for case-insensitive exact matches, glob matches and simple regular expression matches. This is the area that had a substantive amount of work applied to it, mainly to ensure that the computed stack depth used during the compilation of the body scripts was actually correct so that any break or continue across the `switch` command would function correctly instead of causing one of a whole variety of crashes.

  Note that this did not substantially change the actual code generated; this is much more about getting correct metadata about the generated code, so that *other* code could be generated correctly.

- *Uncompiled.* This represents the degenerate case, and is used in awkward situations such as when a script is not a literal, or when a particularly complex regular expression match is used (particularly with capturing of the subexpressions). This case remains in need of substantial work in the future in order to reduce the number of variations of switch that are not compiled.

Improving try

For the try command, the key to understanding its fundamental complexity is that there are two major conditions to consider when doing code generation: the correct way to create code depends strongly on whether or not there are any on or trap clauses[4], and whether or not there is a finally clause. If there are neither, the try is little more than an eval variant (without the concatenation).

The aim of any code generation plan for a construct like try has to be to keep the amount of overhead down. Additionally, care has to be taken when an error occurs during the processing of any on, trap or finally clauses, as the original exception state has to be embedded in the option dictionary's -during option. The major opportunity for optimization is when there is *no* finally clause and *no* trapping of a TCL_OK result, when it is possible to allow the exiting of the context without having to do a full trap and reissue of all exceptions; only the actual exceptional cases need special extra processing.

If we look at this code:

```
try {
   puts "foo bar"
} on error msg {
   puts "bad stuff: $msg"
}
```

[4] The only difference between a trap clause and an on error clause is that the former also checks whether the given words match a prefix of the -errorcode list.

This is only compiled when placed in a context with a local variable table (*i.e.*, in a procedure, lambda term or method) and it produces this rather long bytecode sequence in Tcl 8.6.0 (for clarity, the parts that are *not* actually executed normally are indented, and the parts that *are* normally executed are in bold):

```
beginCatch4 0
push "puts"
push "foo bar"
invokeStk1 2
push "0"
reverse 2
jump1 +4                    # → pc 22
 pushReturnCode
 pushResult
pushReturnOpts              # = pc 22
endCatch
storeScalar1 %2
pop
storeScalar1 %1
pop
dup
push "1"
eq
jumpFalse4 +26              # → pc 60
 pop
 loadScalar1 %1
 storeScalar1 %msg
 pop
 push "puts"
 push "bad stuff: "
 loadScalar1 %msg
 concat1 2
 invokeStk1 2
 jump4 +11                  # → pc 66
pop                         # = pc 60
loadScalar1 %2
loadScalar1 %1
returnStk
done                        # = pc 66
```

With the changes, this is now rather longer (alas) because of the need to handle the -during exception logging, but also correct and faster when no errors occur:

```
beginCatch4 0
push "puts"
push "foo bar"
invokeStk1 2
endCatch
jump4 +103                  # → pc 115
 pushReturnCode
 pushResult
 pushReturnOpts
 endCatch
 storeScalar1 %2
 pop
 storeScalar1 %1
 pop
 dup
 push "1"
 eq
 jumpFalse4 +78           # → pc 109
 pop
 loadScalar1 %1
 storeScalar1 %msg
 pop
 beginCatch4 1
 push "puts"
 push "bad stuff: "
 loadScalar1 %msg
 concat1 2
 invokeStk1 2
 endCatch
 jump4 +57                # → pc 115
 pushResult
 pushReturnOpts
 pushReturnCode
 endCatch
 push "1"
 eq
 jumpFalse1 +28           # → pc 98
 loadScalar1 %2
 reverse 2
 storeScalar1 %2
 pop
 push "-during"
 reverse 2
 dictSet 1 %2
 reverse 2                # = pc 98
 returnStk
 jump4 +11                # → pc 115
 pop                      # = pc 109
 loadScalar1 %2
 loadScalar1 %1
 returnStk
done                      # = pc 115
```

In particular, I have highlighted in bold the instructions taken when no error occurs in the body; as can be seen, the number of instructions processed in this, the expected case, is now far smaller; the *execution overhead* of the try command is demonstrably reduced despite the increase in length of bytecode created. This difference is only exacerbated when the try command has a sequence of trap clauses instead of a single simple on error clause.

Improving dict with

Sometimes, the Tcl community find things to do with Tcl commands that I never anticipated. So it was with dict with, where one of the key use-cases has turned out to be converting a dictionary into a group of local variables *without* maintaining the binding to the mapping in the dictionary. The common idiom for this technique is to use an empty body script, and that has the advantage that it is a case that can be simply detected during compilation.

When I detect this case, I am able to determine precisely that there can be no exceptions arising from the evaluation of the body of the dict with — no commands, no substitutions, therefore no errors — so I can omit the (complex!) code stanza to manage exceptions arising from the body script. Indeed, I can actually omit virtually everything from the implementation of the command other than the operation to expand the dictionary into variables and the operation to write the values back. The latter has to remain as I cannot prove at the time of issuing the code that none of the variables have a trace set on them that modifies the values.

Improving break and continue

Traditionally, the break and continue commands are implemented as commands that produce the specialized result codes TCL_BREAK (defined to be 3 in tcl.h) and TCL_CONTINUE (4) respectively. In the bytecode-compiled era, these have been translated into instructions (with the same names as the commands) that generate the relevant exception conditions. However, this is actually not very efficient, as it requires the code processing these conditions to jump outside the main bytecode evaluation loop to the separate code that finds the exception target in the two cases.

Instead, I track exactly what exception trapping ranges are present at the point where the break/continue is issued. By finding the innermost active range, I can directly determine where the exception would end up branching to, and directly use a jump instruction instead. This is a significantly more optimal instruction sequence.

However, there are some *significant* wrinkles to this. In particular, it is possible for the stack depth to be different between the place where the break/continue command is being compiled and the place where we want to jump to; this is actually a bug in Tcl's bytecode generation of long standing. The problem is not (usually) the nesting of commands that you see with most Tcl scripts, but rather more esoteric scripts such as:

```
while 1 {
   puts "foo,[continue]"
}
```

This is problematic because words for the inner invocation of puts are being placed on the evaluation stack when the continue is processed, and a skip back to the start of the loop without resetting (whether done via an exception or via a direct jump) results in the overall stack depth growing without reasonable bound. Preventing this requires additional pop instructions to be done before the jump so that the stack depth is correct for the target instruction. This does somewhat decrease the efficiency in this case, but since it is rare and previously a critical error, the cost is entirely justifiable. (The complexity of tracking the stack depths and jump targets is only borne during compilation, not execution.)

A variation on this (also handled) is where there are expansions being processed on the stack at the same time, because expansions have their own tracking stack.

Improving expanding list

The final major improvement to code generation that I have made was to the list command when some of its arguments are derived from expansion. While it is not generally possible to handle expansion which produces the first word of a command, as it easily becomes impractical to figure out what command is being compiled, when that command is capable of processing any number of arguments and producing a result, it should be possible to make a compiled version of that command. The only command for which I have done this is list, where an all expanding arguments version:

```
list {*}$foo {*}$bar
```

is actually semantically equivalent to this:

```
concat [lrange $foo 0 end] \
       [lrange $bar 0 end]
```

However, it can be implemented in a considerably more efficient manner, as there is no need to consider what is going on with string interpretations; it can be a pure list operation.

In terms of how the code generated changes, this:

```
list {*}$foo {*}$bar
```

used to compile to this[5]:

```
expandStart
push "list"
loadScalar %foo
expandStkTop 2
loadScalar %bar
expandStkTop 3
invokeExpanded
```

Following this change, we instead generate this sequence of instructions:

```
loadScalar %foo
loadScalar %bar
listConcat
```

This sequence with three arguments to list, one of which is expanded:

```
list $foo $bar {*}$grill
```

Used to compile to this:

[5] It turns out that the numeric parameter to expandStkTop is not actually necessary for the instruction to function correctly, not now that stack depth calculations are correct.

```
expandStart
push "list"
loadScalar %foo
loadScalar %bar
loadScalar %grill
expandStkTop 4
invokeExpanded
```

But now becomes this:

```
loadScalar %foo
loadScalar %bar
list 2
loadScalar %grill
listConcat
```

The listConcat instruction used is a simple binary operation to concatenate two lists.

It should be noted that the mechanism for allowing a command to take charge of what instructions it is compiled when expansion is present is generic. The potential exists to increase the number of commands that produce efficient code, but most commands have the issue that they support additional options or fixed-position arguments, so compilers have to be relatively careful; the list command is a special case in that it is a simple command that treats all arguments (after the initial command name) exactly the same.

5. Improving Inspection

As part of this work, and with a little minor prodding from Colin McCormack, I found that there were some fairly severe limitations on the built-in disassembler that needed to be addressed.

Disassembler history

The disassembler in Tcl 8.5 was originally a part of Tcl 8.0 as a debugging tool. It was enabled by setting the variable tcl_traceCompile to 2 and causing the code in question to be compiled, when it would print the disassembled bytecode directly to standard out. (In situations without a real C-level stdout, such as in *wish* or *tkcon* on Windows, no output would be produced at all.) As you can no doubt guess, this was highly tricky to use; you had to wrap it in something like this, assuming an argument to the procedure was required for it to work:

```
proc disasProc {procedureName} {
   global tcl_traceCompile
   rename set SET
   rename SET set
   set tcl_traceCompile 2
   catch {$procedureName}
   set tcl_traceCompile 0
   return
}
```

This was horrible (especially the need to flush the bytecode cache by bumping the compilation epoch) so in Tcl 8.5 I altered the code to do the direct printing to instead dump its results out to a string buffer instead of printing directly, and wrapped this into the disassemble command (which was put in the tcl::unsupported namespace to signify that we were not guaranteeing the interface) together with a bit of code to allow the control of exactly what was being printed. This allowed the equivalent of the above to be done with:

```
proc disasProc {procedureName} {
   puts [disassemble proc \
            $procedureName]
}
```

This is an enormous improvement in usability, and works correctly on all platforms. (The parameter passed as proc above is used to disambiguate between procedures, scripts, lambda/apply terms, etc.)

However, the results of the disassembly (used in abbreviated form earlier in this paper) are difficult to consume in a script, as they are designed purely to be a basic debugging tool. In particular, they intermix metadata and the disassembly information. There is also quite a bit of information in the bytecode itself that is not exposed in the disassembled code, mostly relating to the parsed script's commands.

The new disassembler

To address this, I have created an additional disassembler[6] that takes the same arguments as the existing disassembler, but instead of producing its results as a complex string, it generates a dictionary that describes the bytecode. The description dictionary contains these keys:

- literals — contains a list of all literal values used in push instructions in the bytecode.
- variables — contains a list of information about all variables defined in the bytecode. Each variable is represented by a list where the first word is a set of flags (*e.g.*, "scalar" to indicate that the variable is a simple variable) and the second word is the name of the variable; temporary variables don't have the second word as they are unnamed.
- exception — contains a list of dictionaries that describe the *exception ranges* in the bytecode. Each of these dictionaries states what type it is (catch ranges trap all non-TCL_OK exceptions, loop ranges only trap TCL_BREAK and TCL_CONTINUE), what range of instructions are covered, and where to jump to when an exception occurs.
- instructions — contains a *dictionary* of the disassembled code, with the keys of the (inner) dictionary being the numeric addresses of the instructions in order and the values of the dictionary each being a list that is the disassembly of the instruction. The first value in each list is the instruction name, and the subsequent values are the arguments. Each argument may be an integer, a reference to a literal (an index into the literals list preceded by "@"), a jump target (an address preceded by "pc "), a variable index (a "%" followed by the index into the variables list), an immediate index literal (starts with a period, ".", followed by a list index which may be either start- or end-relative), or an auxiliary index (a "?" followed by an index into the auxiliary list).
- auxiliary — contains a list of auxiliary information descriptors. This is used to encode three key things: the description of what variables are used by the foreach-related instructions, the description of relative jumps to use in a jumpTable instruction, and the description of how to map variables in the instructions used to compile dict update. Each descriptor is a dictionary where the only guaranteed key is name, which holds the name of the type of auxiliary information encoded in the particular record.
- commands — contains a list of dictionaries that describe what commands were detected in the code that was compiled to produce the bytecode. The order of the list of dictionaries is the order in which the commands start within the script. For each command, the dictionary contains the range of instructions generated from that command (as addresses, in the codefrom and codeto members), the range of the source code that the command occupied (as offsets from the beginning of the script, in the scriptfrom and scripto members), and in the script element, the full text of the command that was compiled (which can include subcommands).

[6] This is ::tcl::unsupported::getbytecode at the time of writing. This is not a name that I consider to be a long-term name, as it is not *getting* the bytecode so much as *describing* it.

- `script` — contains the full text of the script that was actually compiled to produce the bytecode.
- `namespace` — contains the fully qualified name of the namespace used as context to resolve commands in the bytecode.
- `stackdepth` — contains the maximum stack depth (approximately the maximum number of arguments that need to be on the stack at once, plus the space for evaluating expressions).
- `exceptdepth` — contains the maximum depth of nested exception ranges.

The general principle of how the information is encoded in the result of `getbytecode` is to ensure that the maximum amount of information from the low-level bytecode is present *without* exposing any of the complex encodings that are used there or requiring consumers of the result to know how each of the instructions actually treats its result. In fact, this isn't *quite* all the information in the bytecode, but it is exceptionally difficult to make use of the rest (such as the interpreter reference and the compilation epoch) from scripts.

Using the disassembly

Though the output of `getbytecode` is not easy to read as a person, for scripts it is exceptionally easy to process. This makes it easy to do things like analysing the flow of control in the program, detecting automatically where loops are and allowing the visualization of how exception ranges are used.

An example of what can be done with this is shown in Appendix 2, where the `controlflow` command (based on a heavily-modified version of a x86 instruction renderer[4] originally written in Python) prints out the address of each instruction followed by the instruction itself, with arguments converted to an easier-to-read form (*e.g.*, variable references are converted to %*name*, or %%%*index* if they are nameless temporaries). Arrows are added as a prefix to indicate jumps, the different colour applied to the instructions in the middle of the output indicates that a (`loop`) exception range is in force, and the two coloured addresses are the targets for the exception range, one for `TCL_CONTINUE` (being where to go to start the next loop iteration) and the other for `TCL_BREAK` (for finishing the loop). The output part of the code renders to a normal Unix terminal.

It should be noted that the technique I used is not infallible when it comes to display. When asked to display a moderately complex procedure, such as those present in the implementation of Tcl's `clock` command (*e.g.*, `ParseClockFormatFormat2`), that has a number of substantial `switch` statements that compile into jump tables, the number of indents becomes larger than any reasonable width of terminal.

6. Optimization

Given all the work described above, it has become clear that it is necessary to generate improved bytecode. There are a number of places where simply generating better code *within* the implementation of a command was not generating good code within the wider stream of bytecode.

For example, it is the fundamental structure of Tcl command compilation that they overall push a single word onto the bytecode execution engine's stack. Then, in the common case where the result of the command is not needed, that word is then immediately popped off the stack again. It is therefore closer to optimal to not push the value in the first place, if it is possible to avoid doing so. Some cases are particularly easy to determine, such as where the commands being compiled always produce an empty (or other constant) value as their results; the last step of the compilation of both `for` and `foreach` is the push of an empty value, and it is very rare to actually use the results of those commands precisely because it is always the empty string.

Yet to *safely* avoid doing that extra work at runtime, you have to be cautious during compilation. In particular, suppose the generation of the pointless result is followed by a `pop` of the value, but the `pop` can also be jumped to from elsewhere (a pattern easily generated when using the `if` command) then might well be wrong to just remove the `push`/`pop` sequence as that will cause other code paths to create an unbalanced stack.

The simplest way of preventing such problems with code removal is to determine if there are any jumps (of any kind, including conditionals, result-branch operations, jump tables, exception targets, etc.) and to only do the removal when the `pop` can only be reached by that one `push`. This safety requirement reduces the number of optimizations done, but ensures that those that are done are correct.

Optimizations performed

There are a number of optimizations that are done now that were not part of Tcl 8.6.0. These are:

- Removal of `push`/`pop` sequences, as described earlier.
- Folding logical `not` into a following `branch` by inverting the `branch` instruction's condition.
- Removing `tryCvtNumeric` (part of the compilation of expressions) when the subsequent instruction will perform the numeric conversion anyway.
- Advancing jumps to their ultimate target, instead of having them pass through a chain of jumps and `nop`s to get there. This was a relatively common pattern, especially given that jumps are now being generated from `break` and `continue` commands.
- Removing `startCommand` instructions[7] where the code is found to be suitably "well-behaved", such as compiling to bytecode without any invoking of external commands or being located within the implementation of Tcl. Unlike the other optimizations described here, this one is done by rerunning the compiler in a special mode where it simply does not issue the instruction we want to exclude.
- Removing outright unreachable code at the end of a bytecode compilation. Where there is code after a `done` instruction that is not jumped to, it is possible to determine exactly that the following code cannot possibly be executed, and so makes it a good candidate for removal. However, this is an exceptionally minor optimization, as unreachable code is not executed by virtue of its very unreachability.

These optimizations are supported by a new peephole optimization within the execution engine. I added a special case to the processing of `pop` instructions so that a sequence of `pop`s would be handled more efficiently. The optimizer generates such sequences at this point because it does not move any pointers into the bytecode other than those held by simple `jump` and `branch` instructions. In particular, command boundaries are not modified; they have a singularly complex encoding that it is non-trivial to work with.

[7] The `startCommand` instruction is used to skip the rest of the bytecode of a command when the compilation epoch of the interpreter has been updated since the start of processing of the bytecode in question, typically in response to a rename or deletion of a command with a bytecode compiler attached to it. It is a common instruction to deal with a rare case so as to ensure official semantic correctness, and unfortunately is relatively expensive to process.

7. Performance Measurements

In order to compare the performance of Tcl between different versions, it is necessary to be very specific about what is actually being tested. In particular, it is easy to measure something completely different to what you *expected* to measure. To that end, the code used to perform the performance measurements is included in Appendix 3; the timings in Table 2 are rounded to 4 significant figures and are in microseconds.

Program	8.5.14	8.6b1	8.6b2	8.6.0+
ListConcat	0.418	1.564	0.544	0.493
Fibonacci	1.266	1.722	1.441	1.441
ListIterate	3.077	3.315	2.091	2.176
ProcCall	0.863	1.449	1.310	1.264
LoopCB	1.074	1.714	1.404	1.617
EnsDispatch1	0.975	1.944	1.400	0.888
EnsDispatch2	0.489	1.385	0.990	0.393
EnsDispatch3	0.520	1.598	1.261	1.112
EnsDispatch4	0.448	1.311	0.803	0.804
DictWith	2.634	4.072	1.895	1.289
TryNormal	*N/A*	26.221	1.385	0.522
TryError	*N/A*	39.313	3.804	3.931
TryNested	*N/A*	57.236	7.727	11.788
TryOver	*N/A*	39.230	4.120	4.290

Table 2: Times to execute key programs

The *ListConcat* program tests the performance of the new way of handling expansion in the `list` command. The *Fibonacci* program tests general bytecode and integer operation handling, the *ListIterate* program tests general bytecode and list operation handling, and the *ProcCall* program tests the costs of calling a procedure. The *LoopCB* program tests the costs of the `break` and `continue` commands. The *EnsDispatch** programs test the performance of ensembles: *EnsDispatch1* tests the costs of doing ensemble dispatch for two cases where we can convert to a direct invocation of the implementation command, *EnsDispatch2* tests the costs where we can fully compile to bytecode, *EnsDispatch3* tests a case that still has to use the full ensemble dispatch mechanism, and *EnsDispatch4* tests the costs for user-defined ensembles, verifying that no unreasonable costs have been introduced inadvertently. The *DictWith* program illustrates the handling of the empty-body special case in `dict with`. The *Try** programs illustrate the change of profile of costs associated with `try`: *TryNormal* shows what happens when no error is trapped, *TryError* shows a trapped error, *TryNested* shows the relative costs of throwing an error in a handler script (note that none of these execute on Tcl 8.5; the `try` command was new in 8.6, and was not compiled fully in 8.6b1), and *TryOver* shows the overhead associated with the evidence collection technique used in *TryNested* (using the interior workload associated with *TryNormal*).

Performance tests were done on a MacBook Pro with a 2.7GHz Intel Core i7 processor running OS X 10.8.4. The tests were deliberately not disk intensive, and the amount of memory used was much smaller than the free memory available. The executables used for these performance tests were compiled from clean checkouts of the source tree for the release versions associated with each tag from fossil, except for 8.6.0+, which corresponds to the commit labelled `bc57d06610b7`. All were compiled with exactly the same version of the compiler, and using the default, optimizing configuration. The benchmark driver script forces compilation of the code being benchmarked by running it once, then runs it for 100k iterations each of 20 times, taking the minimum (so as to try to avoid any potential problems with jitter due to the OS).

The overall evidence[8] is that some of the optimizations are definitely valuable; for example, the optimizations to core ensemble dispatch restore or even improve on the speed that was in Tcl 8.5 for the majority of ensem-

[8] It makes no sense to combine the performance figures, as the benchmarks are not chosen at all to represent realistic code or to give equal weight to all operations. They are best regarded as *samplings* of the performance of *small parts* of the implementation of Tcl.

ble subcommands. Similarly, some other operations (e.g., `dict with`, `try` in the non-error case) are clearly much cheaper now.

However, not everything is faster; command dispatch is definitely slower in 8.6 than in 8.5 and that has an impact on many of these benchmarks (most notably *ProcCall* and *EnsDispatch4*) and I have no idea why *ListIterate* is so much faster in 8.6 and *LoopCB* so much slower; further examination of the situation will be required. The increase in execution time of *TryNested* is expected due to the change of semantics of option dictionary handling in the error-in-handler case; *TryOver* confirms that this is the cause, and not the additional overhead of the error trapping used in our little benchmarking framework.

8. Future Considerations

This document represents on-going work; many things remain to be done in each of the areas described. For example, in the area of language coverage, we still need to analyse what is the actual set of commands required to allow the majority of Tcl scripts to be virtually entirely compiled to bytecode without the use of the generic dispatch sequence. There are a number of commands that are fairly common, relatively simple, but which are not compiled (*e.g.*, `string trim`). Which ones can we change that status on? This remains to be determined.

On the other hand, we also know that the large majority of Tcl scripts are going to be continuing to call commands even after conversion to bytecode. This is because there is an on-going need to invoke user-defined commands, whether they are procedures, objects or functionality defined in an extension written in a foreign language. What can we do to make that step more efficient? Can we support anything like inlining of procedures? (It is my theory that the last question can be definitely answered affirmatively with relative ease provided the local variable table has no named entries in it, but that's an incredibly restrictive condition; the real question is whether it is possible to do so with fewer restrictions, and to what extent the results change the visible semantics.)

When it comes to the quality of the generated code, more can be done. In particular, the current mechanisms for exception handling are especially complex, and have a far-reaching impact on code generation. Perhaps adding a mechanism such as perhaps internal subroutines *à la* classic BASIC would enable at least some reduction in complexity.

There is also the fact that we currently need to explicitly push exception depths on the stack; it would be far nicer (from the perspective of code *generation* at least) if the target stack depth information were encoded in the exception range record. After all, we now have that information accurately during code generation.

For the disassembly side of things, the obvious thing to do now would be to move to allowing the assembler to be able to handle what the disassembler produces in some way, possibly with syntactic changes, so that we can perform a full round-trip from Tcl code to bytecode to disassembled bytecode to code that is executable again. Currently there are a few key things missing, most notably including the ability to issue the `foreach`-related instructions. The aim would be to enable the writing of more of the optimizer in Tcl itself (ignoring for now the problems associated with optimizing the optimizer's own code).

However, for all the above, the major challenge for the future of bytecodes has to be to improve the optimizer. The next step has to be to actually remove irrelevant and unreachable code and other miscellaneous related structures (*e.g.*, exception ranges). This would let the code issued be made quite a bit more compact. This might in turn require substantial reconfiguration of the in-memory representation of bytecode.

Longer term

The real goal is meeting the Lehenbauer Challenges and achieving speedups of between 2× (*i.e.*, code that executes in half the time) and 10× (*i.e.*, code that executes in a tenth of the time) as these will improve a great many Tcl scripts instead of specific code. The optimization strategies described within this document may go some way towards addressing the lower end of that range of improvements depending on the exact profile of code to be improved (indeed, the *DictWith* micro-benchmark in Section 7 already achieves a better-than-double speedup), but the higher end will definitely require native code generation.

The aim of this work is therefore to provide an improved basis for generating code where it is possible to more easily analyse the code to be native-compiled and determine its real type behaviour. Proving micro-theorems about the types of variables is the key to determining how to generate *good* native code from Tcl programs, and it is conjectured that bytecode has a key advantage over straight Tcl code as a starting point in that the type logic of bytecode is more static. It remains to be seen if this conjecture is actually a true one.

9. References

[1] Ugurlu, O.D., Kenny K., *A bytecode assembler for Tcl*, in Proceedings of the 17th Annual Tcl/Tk Conference (Tcl'2010), Tcl Community Association, Whitmore Lake, MI, 2010.

[2] Vitale, B., Abdelrahman, T.S., *Catenation and specialization for Tcl virtual machine performance*, in Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators (IVME'04), pages 42–50, ACM, Washington, DC, 2004. doi:10.1145/1059579.1059591

[3] Lewis, B., *An on-the-fly bytecode compiler for Tcl*, in Proceedings of the 4th Annual USENIX Tcl/Tk Workshop, USENIX, Berkeley, CA, 1996.

[4] Fairchild, D., *Automagical ASM Control Flow Arrow-Annotation*, weblog entry at http://blog.fairchild.dk/2013/06/automagical-asm-control-flow-arrow-annotation/, June 11, 2013, accessed June 19, 2013.

Appendix 1: Teapot Ensemble Subcommand Frequency Tables

In the tables below, the assessment of whether a command is practical to bytecode depends mainly on the internal complexity of the command; commands that create or destroy commands are *always* considered to be impractical as they potentially modify the interpreter epoch. The final column states whether the subcommand was bytecode-compiled in Tcl 8.6b2, *i.e.*, prior to the work in this paper.

The shell script used to extract the information was (with the environment variables `$TEAPOT_REPOSITORY` and `$TCL_CMD` supplying the place to look for the sources and the command to look for respectively):

```
find $TEAPOT_REPOSITORY -name "*.tm" -print0 | xargs -0 grep -lw $TCL_CMD \
      | xargs cat | grep -w $TCL_CMD | sed -nE "
            /$TCL_CMD +\[a-z\]/ {
                  s/^.*$TCL_CMD (\[a-z\]*).*\$/\\1/
                  p
            }
      " | sort | uniq -c | sort -n
```

The result of that script was then hand-filtered to remove irrelevant values (such as words in free-text that just happen to mention the major command) and to coalesce abbreviations onto their main subcommand.

For the `string` command (subcommands not mentioned did not occur):

Command	# Uses	Practical to Bytecode	Bytecoded in 8.6b2
`string totitle`	1	Possibly	No
`string replace`	2	Possibly	No
`string trimleft`	8	Yes	No
`string last`	28	Yes	No
`string trimright`	33	Yes	No
`string repeat`	34	Possibly	No
`string toupper`	145	Possibly	No
`string trim`	147	Yes	No
`string index`	245	Yes	Yes
`string tolower`	248	Possibly	No
`string is`	424	Possibly	No
`string map`	569	Possibly	No
`string first`	674	Yes	No
`string range`	892	Yes	No
`string match`	898	Yes	Yes
`string length`	1100	Yes	Yes
`string equal`	2129	Yes	Yes
`string compare`	5971	Yes	Yes

For the `dict` command (subcommands not mentioned did not occur; note that `dict map` was only introduced after this survey was done):

Command	# Uses	Practical to Bytecode	Bytecoded in 8.6b2
dict keys	1	Yes	No
dict with	1	Yes	Yes
dict unset	2	Yes	No
dict lappend	3	Yes	Yes
dict values	8	Yes	No
dict for	8	Yes	Yes
dict merge	15	Yes	No
dict incr	18	Yes	Yes
dict create	22	Yes	No
dict append	28	Yes	Yes
dict exists	34	Yes	No
dict get	297	Yes	Yes
dict set	347	Yes	Yes

For the namespace command (subcommands not mentioned did not occur):

Command	# Uses	Practical to Bytecode	Bytecoded in 8.6b2
namespace forget	3	No	No
namespace inscope	6	Possibly	No
namespace parent	7	Yes	No
namespace children	17	Possibly	No
namespace qualifiers	30	Yes	No
namespace exists	50	Yes	No
namespace which	56	Yes	No
namespace delete	77	No	No
namespace code	116	Yes	No
namespace import	130	No	No
namespace upvar	132	Yes	Yes
namespace origin	153	Possibly	No
namespace tail	206	Yes	No
namespace ensemble	269	No	No
namespace current	272	Yes	No
namespace export	757	Possibly	No
namespace eval	2681	No	No

For the array command (subcommands not mentioned did not occur):

Command	# Uses	Practical to Bytecode	Bytecoded in 8.6b2
array size	37	Possibly	No
array exists	56	Yes	No
array unset	191	Yes	No
array get	479	Possibly	No
array names	1085	Possibly	No
array set	2511	Yes	No

For the info command (subcommands not mentioned did not occur; note that `info class` and `info object` are themselves ensembles, and that `info patchlevel` is impractical due to the "interesting" failure mode behaviour):

Command	#Uses	Practical To Bytecode	Bytecoded in 8.6b2
info class	1	Yes (parts)	No
info default	1	No	No
info loaded	1	No	No
info frame	2	Possibly	No
info sharedlibextension	4	No	No
info nameofexecutable	8	No	No
info object	8	Yes (parts)	No
info patchlevel	11	No	No
info complete	12	Possibly	No
info body	29	No	No
info vars	33	Possibly	No
info args	42	No	No
info procs	50	No	No
info hostname	54	No	No
info script	67	Possibly	No
info commands	274	Possibly	No
info level	587	Yes	No
info exists	3822	Yes	Yes

For other ensembles, they are typically wholly impractical to bytecode other than through generic mechanisms (*e.g.*, the chan ensemble, which is thoroughly entangled with the I/O subsystem and so likely to encounter dominating OS-related delays, but which still benefits from the generic improvements to the ensemble mechanism).

Appendix 2: Example of controlflow Output

Sample code used to create the output:

```
controlflow lambda {{a b c} {
    set sum 0
    foreach x [list $a $b $c] {
        incr sum [expr {$x**2}]
    }
    puts "sum of squares: $sum"
}}
```

Output from the above code:

```
          0 push1 "0"
          2 storeScalar1 %sum
          4 pop
   ┌─     5 startCommand ➡ 61 2
   │     14 loadScalar1 %a
   │     16 loadScalar1 %b
   │     18 loadScalar1 %c
   │     20 list 3
   │     25 storeScalar1 %%%4
   │     27 pop
   │     28 foreach_start4 {data %%%4 loop %%%5 assign %x}
┌──┼►    33   foreach_step4 {data %%%4 loop %%%5 assign %x}
│ ┌┼─    38   jumpFalse1 ➡ 59
│┌┼┼─    40   startCommand ➡ 56 2
││││     49   loadScalar1 %x
││││     51   push1 "2"
││││     53   expon
││││     54   incrScalar1 %sum
│└┼┼►    56   pop
└─┼┼─    57 jump1 ➡ 33
  └┼►    59 push1 ""
   └►    61 pop
         62 push1 "puts"
         64 push1 "sum\ of\ squares:\ "
         66 loadScalar1 %sum
         68 concat1 2
         70 invokeStk1 2
         72 done
```

As you can see, the implementation of the loop body has been indented, and the (loop) exception range — the range of instructions where a TCL_BREAK or TCL_CONTINUE instruction will trigger a jump to a nominated target instruction — has been highlighted in red. The break-target for the exception range has its address highlighted in blue, the continue-target has its address highlighted in red.

The source to the controlflow command is too long to include in this document. It can be downloaded from DropBox: https://dl.dropboxusercontent.com/u/19238925/Tcl/2013/controlflow.tcl

Appendix 3: Performance Measurement Script

The performance measurements of Section 7 were done with this consolidated script.

```
package require Tcl 8.5

proc listConcat {a b c} {
    list $a $b {*}$c
}

proc Fibonacci {n} {
    set a 0
    set b 1
    for {set i 2} {$i <= $n} {incr i} {
        set b [expr {$a + [set a $b]}]
    }
    return $b
}

proc iter {param} {
    set result {}
    foreach x $param {
        lappend result [string length $param]
    }
    return $result
}

proc inner {} {
    return "ok"
}
proc outer {} {
    inner; inner; inner; inner; inner
}

proc loopcb {x} {
    for {set i 0} {$i < 10000} {incr i} {
        if {$i == $x} break
        continue
    }
    return "ok"
}

proc ensDispatch1 {} {
    info tclversion
    info patchlevel
}

proc ensDispatch2 {} {
    namespace current
    info level
}

proc ensDispatch3 {} {
    namespace inscope :: {return -level 0 "ok"}
    namespace inscope :: {return -level 0 "ok"}
}

proc ensDispatch4 {} {
    ens4 foo bar
```

```
}
namespace ensemble create -command ens4 -map {
    foo {::ens4core}
}
proc ens4core {msg} {
    return $msg
}

proc dictWithAdd {d} {
    dict with d {}
    return [expr {$a + $b}]
}

proc tryNormal {} {
    set d 1.875
    try {
        set x [expr {$d / $d}]
    } on error {} {
        set x "error happened"
    }
    return $x
}

proc tryError {} {
    # Zero divided by zero is an error (no NaN please!)
    set d 0.0
    try {
        set x [expr {$d / $d}]
    } on error {} {
        set x "error happened"
    }
    return $x
}

proc tryNested {} {
    set d 0.0
    catch {
        try {
            set x [expr {$d / $d}]
        } on error {} {
            error "error happened"
        }
    } msg opt
    return $opt
}

proc tryNestedOver {} {
    set d 0.0
    catch {
        try {
            set x [expr {$d / $d}]
        } on error {} {
            set x "error happened"
        }
    } msg opt
    return $opt
}

proc benchmark {title script {version 8.5}} {
    if {[package vsatisfies [info patchlevel] $version]} {
        eval $script
        for {set i 0} {$i < 20} {incr i} {
            lappend t [lindex [time $script 100000] 0]
        }
        puts [format "%s: %4f" $title [tcl::mathfunc::min {*}$t]]
    } else {
        puts [format "%s: N/A" $title]
    }
}

benchmark "ListConcat"    { listConcat {a b c} {d e f} {g h i} }
benchmark "Fibonacci"     { Fibonacci 10 }
benchmark "ListIterate"   { iter {a aaa aaaaa} }
benchmark "ProcCall"      { outer }
benchmark "LoopCB"        { loopcb 10}
benchmark "EnsDispatch1"  { ensDispatch1 }
benchmark "EnsDispatch2"  { ensDispatch2 }
benchmark "EnsDispatch3"  { ensDispatch3 }
benchmark "EnsDispatch4"  { ensDispatch4 }
benchmark "DictWith"      { dictWithAdd {a 1 b 2 c 4} }
benchmark "TryNormal"     { tryNormal }      8.6
benchmark "TryError"      { tryError }       8.6
benchmark "TryNested"     { tryNested }      8.6
benchmark "TryNestedOver" { tryNestedOver }  8.6
```

This script can be downloaded from DropBox:
https://dl.dropboxusercontent.com/u/19238925/Tcl/2013/optbench.tcl