The TyCL compiler internals and preliminary performance analysis

Andres Buss
Otlet Technologies Ltda
aabuss@otlettech.com

Abstract

Performance was the main reason that drove the inception of the TyCL compiler, to have the possibility to run Tcl programs faster and with an smaller memory footprint. In order to try to have those possibilities, a series of trade-offs[1] at the syntax and functionality levels had to be made besides the inclusion of new syntax and features. The question now is: how can TyCL try to improve the performance of such programs? … or in other words: how it does it? … and: how well or bad it does it?. This paper describes TyCL's internal processes and data-structures and provides a basic and preliminary performance analysis of the compiled programs that it generates, compared against the Tcl VM/Interpreter.

Introduction

Like any other compiler for any other language, TyCL has a large number of ways to try to improve the performance of the programs that it compiles, from paying close attention to the types of the data that the program is handling to paying close attention to the processes that the program is performing on such data, and even taking into account the platform (at the hardware and software levels) in which the final compiled program is going to be executed. Having the perfect compiler is obviously a complete myth, but having a compiler that could get better at its job over time is a very possible concept, it just needs to keep evolving. Right now there are a few practices/strategies implemented as an starting point from where it can start evolving, they are not perfect at the time, but they are effective at improving the performance of TyCL's compiled programs. So, what comes above is the description of such practices/strategies currently implemented inside the compiler.

Practices/strategies

- **Parsing/processing the source code at compile time**

  This is a big deviation from Tcl's way of doing things, where the source code is parsed and processed at run-time and by demand. TyCL parses the hole source code completely at the beginning, including going deep into descriptions of list's elements and the procedure's (procs from now on) bodies of code. By doing this, it releases the run-time VM of this job, effectively decreasing the amount of time needed to start running the actual program.

- **Tracking the data-types**

  Types are the key components of TyCL programs, every data-value has to have a type (even if the program doesn't explicitly declare one). By knowing the data-types in advance, the compiler has the chance to think a little bit harder and generate specific targeted code for each situation in particular, or in other words: it can avoid the time expended by the run-time VM where it has to figure the type of the data out before it can start processing it and go directly to the processing part of the program instead, all in all knowing also the platform where the program is going to be executed.

  This is the most important way of increasing the performance of the compiled programs, and is also the most difficult to achieve, given the dynamic and string-based nature of Tcl. At this moment, TyCL can identify explicit type declarations but it can't relate those declarations through the hole program, it can handle local variable's types but can't go any deeper, for those cases where the compiler is unable to identify the proper types, a generic-discovery solution is generated as a fallback, thus leaving the "problem" to be solved at run-time. Again, this is very difficult to achieve in highly dynamic languages, but it could get better with time.

- **Internal memory model of the data-types**

  As previously stated, every data-value must have a type related to it. TyCL has a memory-model/structure (see figure 1) that allows each type to have its own internal description of any form or size (it just grows the structure accordingly) which is known explicitly by such type in particular, this "additional" description could be composed of additional sub-type declarations, data-sizes, etc.

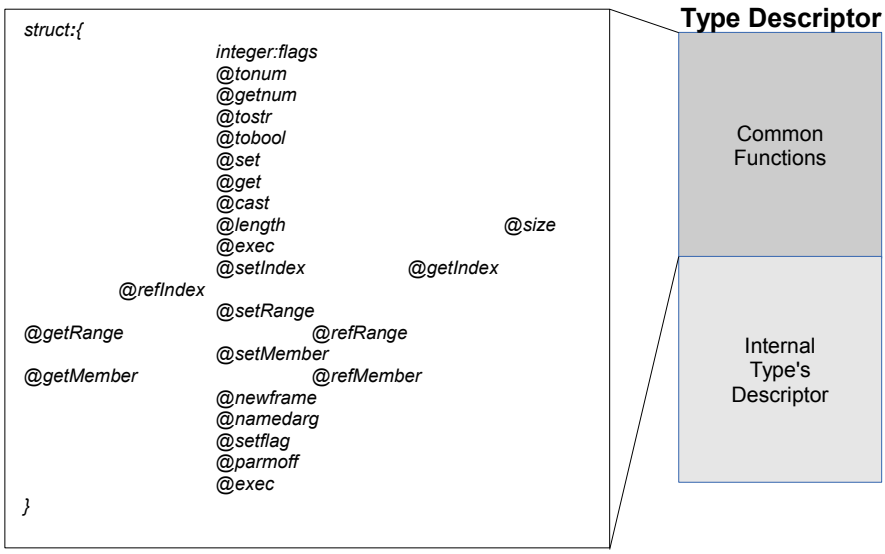

Figure 1. The memory description of types

Thus every type have a common known and expected part composed of a series of pointers to functions meant to manipulate data of such type and a unique (possibly non-existent) part that only each type knows how to handle.

All type descriptions are stored together in memory, composing a type-table where each type is referenced by an offset into the table.

- **Internal memory model of the data-values**

A data-value (see figure 2.) is an structure that holds a particular data in conjunction with a reference to its type, in consequence, all data must be enclosed within a data-value structure unless the compiler get to know the data's type in advance and then this "additional" information is handled at compile-time, and at run-time only the reference to the actual data is needed. Besides the reference to the data's type, this structure provides information about the way the data could be discarded, if it can be modified, if the data must be of the referenced type when is modified, or if the data is really at another place in memory (referenced data), also there is some space reserved to count how many times this data-value has been referenced by some variable, so the system don't discard the structure if a reference to it is still alive.

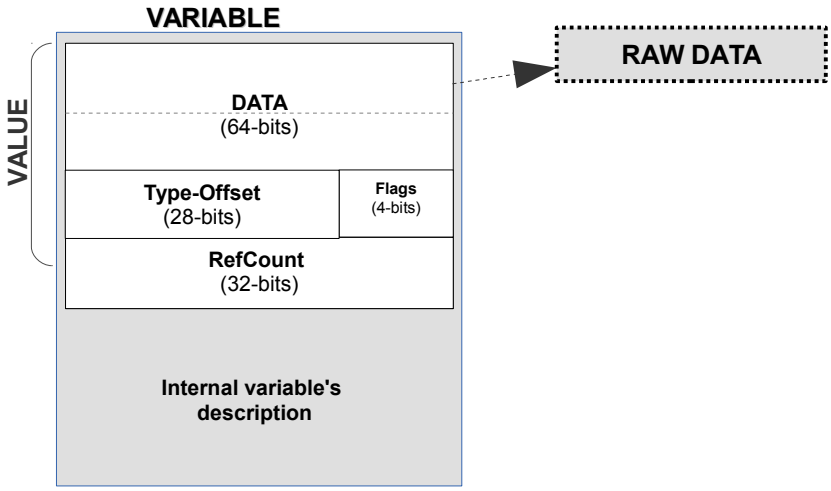

Figure 2. The data-value's memory model

The current implementation of TyCL follows a policy of no-copy-on-write, which means that when a value is assigned into a variable (or a member, index, etc) its content is actually copied/cloned into the variable instead of merely increasing a reference count and sharing a pointer to the real data (as is the case with Tcl). There is also another policy in place that suggests that the system have to provide enough space in memory to store every data whenever possible, all this to try to minimize the usage of dynamically-allocated memory,

which is expensive (in terms of time) to allocate and often to discard.

- **Memory model of composed-data types**

  Not every type of data can be contained inside the data-value structure, which allows a maximum size of 64-bit, there are types that will need more space, in those cases the data contained within the data-value structure points to the real data where the real data could be of any form and size (controlled by its type). For those types which are containers of values (lists, arrays, structures) there are to possible ways of storing those values: if the value's sub-type (the contained value-types) is not defined then the proper data-value structure should be stored along with the actual data, but if the value-types are defined, those definitions can be placed in the extra description space within the type descriptor and then only the actual raw-data is stored (see figure 3), thus reducing the memory requirement to store the same content.

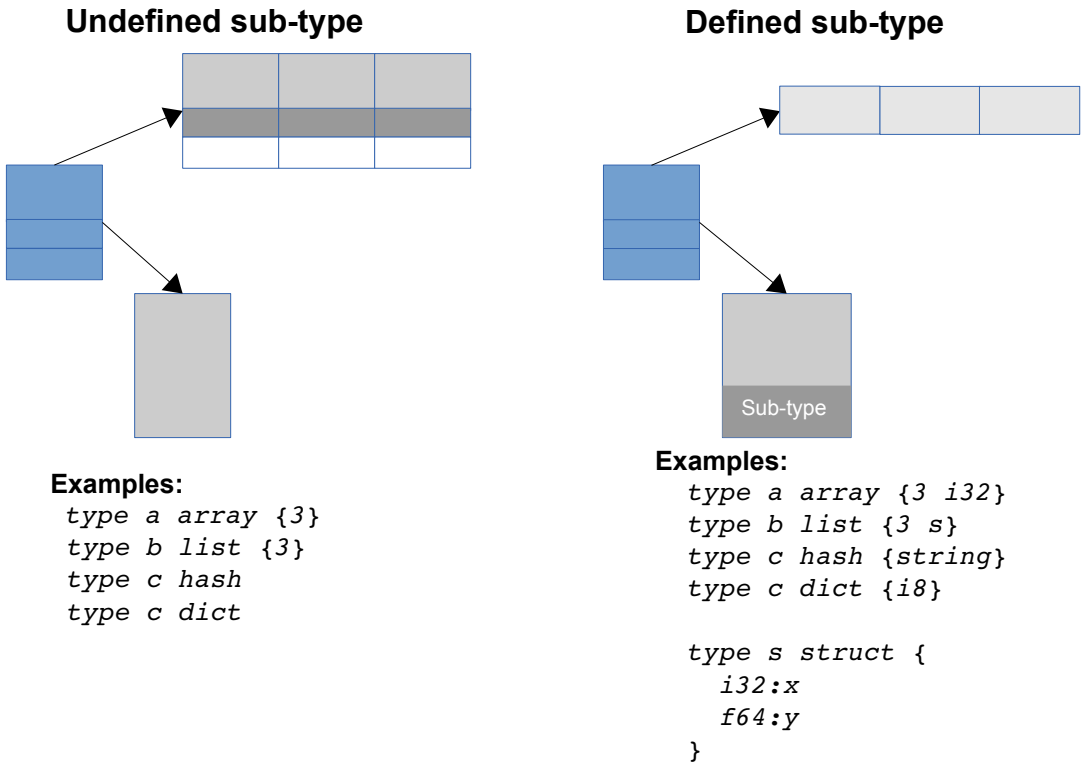

Figure 3. Suggested memory model of types composed by more than one value

- **Function creation**

  Every time a proc is defined, a new type is created with the description of its parameters as the internal descriptor of the type, the system keeps track of all created types and in the case of procs, it only stores a single definition for procs with the same description of parameters. This process is executed at compile-time thus releasing of such job off the run-time VM.
  Additionally, the body of the proc is compiled also and some information is reserved to be placed inside the data field of the value that is going to hold the processed proc's body, among this information is the pointer to where the body's code starts, and the size of the execution frame where the body is going to be executed.

- **Function calling**

  When a function call is going to be executed, the system gets the real address where the proc's body is located in memory and also gets the size of the execution frame used by the body to execute and then creates the frame in memory, this frame is used to pass the arguments of the function being called and to hold its local variables and registers (see figure 4). A proc's body should have anything it needs to execute within the boundaries of its own frame (without asking for more dynamically allocated memory,.. wich costs time), except for raw-data related to specific values.

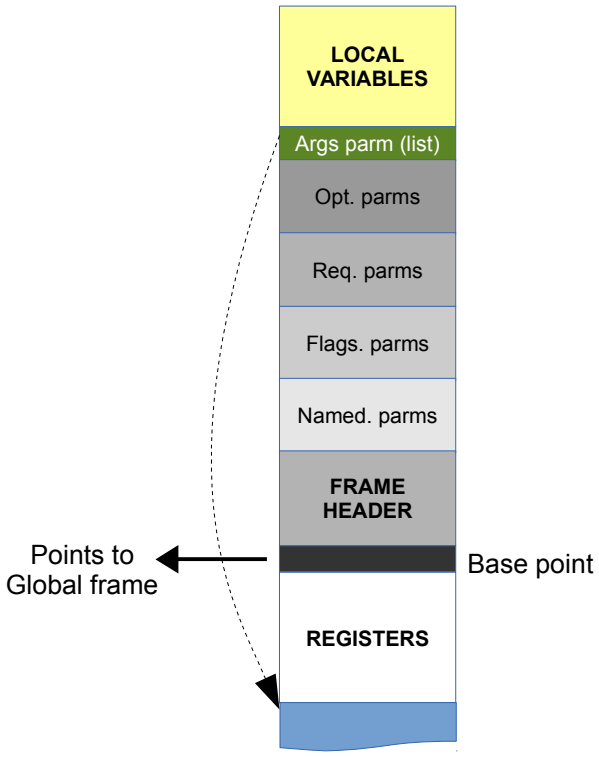

Figure 4. Model of a local frame

Basic performance comparisons

Generally speaking, TyCL produce binary programs that runs between 2 and 25 times faster than the Tcl VM/Interpreter, without taking into consideration the cases where the types of the variables/values are being defined inside the source code, in which cases the speed goes up considerably up to around 50 times faster. So to be fear with Tcl's VM, all the following numbers were taken using code that both Tcl and TyCL can understand without any modification whatsoever.

At the global frame execution, TyCL can access and modify variables 20 times faster than Tcl, and function calling works 22 times faster on average.

At local frame execution (code within a proc), as the Tcl's bytecode compiler kicks in, the difference is considerable lower: variable access and modifications drops to 4 times faster and function calling drops to 7 times faster on average.

In any form where a dynamic evaluation of code is required, as could be the case when the Tcl's VM encounters an expression not surrounded by braces, the TyCL's generated code executes more than 300 times faster.

Conclusions

Even though TyCL is still in an experimental state, its advantages can be noticed right away, the fact that it already generates faster versions of programs from plain Tcl code (without declaring explicit types for example) is a very good starting point. Once one start declaring types inside the code the performance begins approaching the level of c-compiled code.

There still a big space for improving performance, right now the compiler acts like a naive compiler, it doesn't modify or optimize the code in any way as other compilers do, thus when the time for such optimizations come, a new jump in performance could be expected.

References

[1] Andres Buss, “TyCL: an interpreter/compiler of a typed language implementation of Tcl/Tk ”, 18th Annual Tcl/Tk Conference, October 26, 2012