Perl as an Embedded Language

Dominik Brettnacher

August 28, 2004

Abstract

The second part of the seminar on Configurable Systems dealt with programming languages that were specifically designed for embedding into applications.

Unlike these languages, Perl was primarily designed as a language for extracting and processing information, while the idea of embedding Perl only came up later.

In the following, I will compare the Perl language as well as its API with Tcl, Scheme and Lua. Furthermore, I will present a Perl-enhanced Network Monitor as an example for an application that benefits from a scripting language.

1 Motivation

During the second part of the seminar we had a detailed look into several embedded languages and compared their properties:

- Tcl is a language with a shell-like syntax. The interface for embedding is clean and simple. However, Tcl merely supports strings as the only data type, which makes it difficult to handle complex data structures.

- libscheme introduces more advanced features such as lexical scoping and complex data types (e.g. lists and first-class functions). The API for embedding is slightly more complex, compared to Tcl. But main drawback of Scheme is its fully-parenthesized syntax, which makes it difficult to use, as least for a typical end user.

- Lua, a language designed to be simple and flexible aims to provide the advantages of both Tcl and Scheme. Lua's features such as fallbacks and tables are extensible, while both syntax and the API remain relatively small.

In contrast to this, the Perl language was originally not designed to be embedded into host applications: while the roots of Perl date back to the year 1987[2], the first application that actually embedded Perl[1] was developed in 1996. It will be interesting to see how the Perl interface compares to that of the languages already known. What has become apparent from the languages already considered is that the complexity of the API grows together with the complexity of the language itself.

2 The Perl Language

As an introduction, I am going to present some properties of the Perl language. I have chosen them either because they are important with respect to the embedding API or because they are interesting in comparison with the languages discussed earlier.

2.1 Data Types

Perl distinguishes three data types: first, there are scalar values (SV) which are used to represent numbers and strings. A number is transparently converted into a string and vice versa if needed. A reference (RV) is another type of a scalar value and can be thought of as a pointer to another value (or a subroutine).

The second type is the array (AV). An array consists of several scalar values indexed by number (like in C). The most important type however is the associative array, better known as hash (HV). Hashes are indexed by string.

Each data type has its own namespace, that is, a scalar `$address` can coexist with an array of the same name (`@address`). A variable of a certain type is referred to by using the prefix for that type. Scalar values are prefixed with `$`, array and hash values are prefixed with `@` and `%`, respectively.

As stated above, it is possible to create references to any data type. The predominant data type used in Lua is the table. The properties and the behaviour of a table can be compared to a hash reference in Perl. Consequently, Perl objects are usually represented by hash references.

2.2 Subroutines

Perl's subroutines are defined with the `sub` keyword. It is possible to define an anonymous subroutine using a reference and pass it as a parameter or assign it to a variable.

The function call arguments are passed as a list and can be accessed through the special array `@_`. Similarly, the return value is also a list. It is important to note that this interface does not allow to pass complex values. Any hash or array value would be degraded to a list if passed to or returned from a function. However it is possible to pass references to any complex value.

3 The Perl Interface

3.1 Embedding

The process of embedding a Perl interpreter into an application written in C is similar to the other languages. It is documented in [5], a part of the Perl core documentation. The interpreter itself is available as a library that has to be linked to the host program. A special Perl module provides the compiler options needed to accomplish this task.

```
#include <EXTERN.h>
#include <perl.h>

int main(int argc, char *argv[])
{
  static PerlInterpreter *my_perl;

  my_perl = perl_alloc();
  perl_construct(my_perl);

  perl_parse(my_perl, NULL, argc, argv,
      NULL);
  perl_run(my_perl);

  perl_destruct(my_perl);
  perl_free(my_perl);

  return 0;
}
```

Listing 1: A Perl interpreter embedded into C

Listing 1 shows the function calls needed to embed an interpreter. The most important functions are `perl_parse()` and `perl_run()`.

`perl_parse()` tells the interpreter to parse a chunk of code. This function expects command line arguments (i.e. `argc` and `argv`). This approach makes the embedded interpreter work like the stand-alone Perl interpreter. It is either possible to supply Perl code directly (using the `-e` flag from the command line) or a file name. If this is not the desired behaviour, one has to supply "dummy" arguments, as there is no alternative function which does not expect `argc` and `argv`.

```
#include <EXTERN.h>
#include <perl.h>

int main(int argc, char *argv[])
{
  static PerlInterpreter *my_perl;

  my_perl = perl_alloc();
  perl_construct(my_perl);

  perl_parse(my_perl, NULL, argc, argv,
      NULL);

  call_pv("example2", G_DISCARD |
      G_NOARGS);

  perl_destruct(my_perl);
  perl_free(my_perl);

  return 0;
}
```

Listing 2: Calling a subroutine explicitly

If the code is syntactically correct, `perl_run()` can then execute the parsed code. Apart from `perl_run()`, which will start at the first statement supplied, it is possible to call a subroutine explicitly. The example in listing 2 shows how a subroutine named `example2` is called.

3.2 Converting and Passing Values

Like Lua, Perl uses a stack in order to pass function arguments back and forth. Usually, each argument value needs to be converted into a Perl value. It is also possible to use an array of C strings as the arguments for a subroutine call, resembling the Tcl interface. The `perl_call_argv()` will convert the strings into Perl SVs and call the subroutine which was given as a parameter.

```
int call_example3(int a, int b, int c)
{
  int count, result;

  dSP;
  ENTER; SAVETMPS;

  PUSHMARK(SP);
  XPUSHs(sv_2mortal(newSViv(a)));
  XPUSHs(sv_2mortal(newSViv(b)));
  XPUSHs(sv_2mortal(newSViv(c)));
  PUTBACK;

  count = call_pv("example3", G_SCALAR);
  SPAGAIN;

  result = POPi;
  PUTBACK;

  FREETMPS; LEAVE;

  return result;
}
```

Listing 3: Calling a subroutine which takes three integers and returns one

For more complex function calls, the `argv` method is no longer sufficient, because it only converts flat values. In order to push references, arrays and hashes on the stack, they have to be created manually. The Perl API provides a number of functions in order to create and change scalar, array and hash values.

Listing 3 shows how three integers are converted and passed to a subroutine. After calling the subroutine, the return value is popped from the stack and converted to an integer. The glue code needed to call a function consists of several statements (shown in italics) which are needed in order to manage the stack. In contrast to this, the Lua API does not need

Figure 1: Perl API functions

Function	Purpose
*SV newSViv(int)	Creates a new scalar value from an integer
*SV newSVnv(double)	Creates a new scalar value from an float
SV newSVpv(char, int)	Creates a new scalar value from a string
sv_setiv (SV*, int)	Sets the value of a scalar value to an integer
sv_setnv(SV*, double)	Sets the value of a scalar value to a float
sv_setpv(SV*, char*)	Sets the value of a scalar value to a string
*AV newAV()	Creates an empty array
void av_push(AV*, SV*)	Adds a scalar value to the end of an array
SV av_pop(AV)	Pops a scalar value from the end of an array
SV av_shift(AV)	Removes a value from the beginning of an array
void av_unshift(AV*, int n)	Adds `n` empty values to the beginning of an array
**SV av_fetch(AV*, int key, int lval)	Fetches the element at position `key`
**SV av_store(AV*, int key, SV* val)	Stores a scalar value at position `key`
*HV newHV()	Creates an empty hash
**SV hv_store()	Stores a scalar value with `key` as the key
**SV hv_fetch()	Fetches the value stored for `key`
*SV hv_delete()	Deletes the value stored for `key`

this kind of explicit stack management, the glue code only consists of the push/pop operations as well as the conversion functions.

Similar to the Lua API, Perl values are represented by pointers to opaque data structures. The API provides a number of functions to create and change these values (see figure 1). A documentation of all API functions can be found in [4] and [3].

After being created, Perl values usually have to be pushed onto the stack. The `XPUSH` macros extend the stack if needed and put the supplied values on it. Similar, the `POP` macros take a value from the stack and convert it back into a primitive C value.

The environment of the Perl interpreter can be accessed with `get_sv()`, `get_av()` and `get_hv()` for the respective data types.

3.3 Memory Management

The Perl interpreter does garbage collection using reference counts, therefore memory management works automatically most of the time, although there are three situations in which the counter has to be dealt with manually.

If a reference (RV) is created, the developer must decide if the counter of the referenced value has to be incremented or not. For example if a hash reference is created in the host application in order to pass it to a Perl subroutine, the counter of the hash value usually has not to be incremented. Having returned the reference to the subroutine, it is usually no longer needed at the host application and would result in a memory leak. Because of this, the API provides two functions to influence the counter on creation of a reference: `newRV_noinc()` does not increment it while `newRV_inc()` would do so.

The second case are operations on arrays and hashes. The counter of a scalar that is added to an array or a hash is usually not incremented. This is convenient for the usual case of a scalar being created only to insert it into a complex data structure.

In order to ease the management of reference counters, Perl provides a concept called "mortality". The counter of a value marked as mortal will be decremented at "a short time later"[4]. In listing 3 for example, `sv_2mortal()` is used in order to mark the scalars as mortal. As a result, they will automatically be destructed as part of the `FREETMPS` statement.

3.4 Pattern matching

Perl's pattern matching capabilities and its regular expression engine are one of its greatest strengths. The most frequently used idioms are `$string =~ m/pattern/` which tests if a string matches a regular expression and `$string =~ s/pattern/replacement/` which substitutes the occurences of a pattern in a string with a replacement. The latter is very powerful, because `replacement` can be a complex expression, too.

It is not possible, however, to use the regular expression directly from C. The Perl documentation[5] recommends to use glue functions which create and execute the Perl code shown above at run-time. Obviously this is error-prone and needs to be done very carefully, because special characters have to be quoted appropriately.

3.5 Error handling

In order to catch run-time errors, the usual approach in Perl is to enclose a block in an `eval {}` bracket (it is important to say that this is not the same as passing a string to `eval` in order to parse and execute it). This approach prevents the interpreter from exiting after fatal run-time errors.

Instead of using `eval {}`, the `call_*` API functions support the `G_EVAL` flag, which has the same effect.

3.6 Embedding C into Perl

Using C code from Perl (that is, the opposite direction than the rest of this document describes) is generally more convenient than embedding an interpreter. In order to make a C library accessible from Perl, the necessary glue code can be written in a meta-language (XS). It is even possible to automatically generate parts of the glue code from header files using the `h2xs` tool. XS helps to convert C data structures into Perl values using a concept of typemaps. A lot of the modules found in CPAN[1] were built using this technique.

4 Network Monitoring

As a case study of a Perl interpreter embedded into an application, I decided to take a software ("AMON") which does network monitoring and make it extendable through the scripting language. AMON is written in C and works by checking network services like HTTP, SMTP and others periodically. It can be configured at run-time by a database interface. Each service type is handled by a module, which is written in C. They are given a description of the service to be checked in a URL-like format as well as additional details such as an ID for logging purposes. The handler then takes this information, does the actual check and then returns its results. The result contains a status flag and two integer values which can contain information about packet loss, latency time or the amount of data transferred.

The case study consists of Perl meta-module which makes it possible to write AMON modules in Perl. Doing this, AMON can benefit from the advantages of the scripting language: modules can be written in less time, they don't need to be compiled, they can be plugged in

[1]Comprehensive Perl Archive Network, http://www.cpan.org

at run-time and they can profit from the enormous amount of Perl modules already available at CPAN (more than 6800 at the time of writing). This is especially useful if complex applications should be monitored, because then it is no longer sufficient to check if a server responds or not - a complex module can monitor an application in a more detailed way.

4.1 Implementation

The implementation of the meta-module follows the principle described above step by step. After creating an interpreter instance, the parameters, are converted: for each record of the C structures, the module creates a new hash entry using `hv_store()`. After that, it creates hash references and pushes them on the stack. As described above, the reference counter of the hash is not implemented and the references themselves are marked as mortal.

In order to support several Perl modules, the following convention is applied: the module name `perl_`*type* leads to a call of the Perl subroutine named `handler_`*type*. The resulting function name is called in a `G_EVAL` context as explained in order to recover from run-time errors gracefully (e.g. if the subroutine does not exist).

The called subroutine is expected to return a hash reference. The meta-module checks if an error has occured and if the return value is indeed a hash reference. It then fetches the result values from the hash and returns them to the host application.

5 Conclusion

The embedded Perl interpreter works flawlessly in practice, though its API is, as expected, more complex than that of the languages discussed earlier. There are mainly two reasons for this: first, the language itself is more complex than Tcl or Lua with respect to both syntax and feature set. Second, embedding was not the primary goal when Perl was developed. The sheer number of API functions (more than 400[3]) alone is very high, compared to Lua (about 100 only[2]). But what makes the interface especially complicated is awkward interface of `perl_run()` together with the numerous stack operations that have to be done manually.

Apart from the documentation that is part of the Perl distribution, chapters 18 and 19 of [6] are a valuable source of information.

Even if the interface is not as clean as that of Lua and much more complex as that of Tcl, I think that Perl provides a viable alternative in practice because of the CPAN and the widespread distribution of the Perl language.

References

[1] mod_perl. http://perl.apache.org.

[2] Jarkko Hietaniemi et al. *perlhist - the Perl history records.* The Perl core documentation.

[3] Jeff Okamoto et al. *perlapi - autogenerated documentation for the perl public API.* The Perl core documentation.

[4] Jeff Okamoto et al. *perlguts - Introduction to the Perl API.* The Perl core documentation.

[5] Doug MacEachern Jon Orwant. *perlembed - how to embed perl in your C program.* The Perl core documentation.

[6] Sriram Srinivasan. *Advanced Perl programming: foundations and techniques for Perl application developers.* A Nutshell handbook. O'Reilly & Associates, Inc., Cambridge, CA, 1st edition, 1997.

[2]according to the `lua.h` header

A Source code

```
#include <sys/errno.h>
#include <syslog.h>
#include <EXTERN.h>
#include <perl.h>

#include "amon.h"

amon_value *handler_perl(amon_job *myjob, amon_job_identifier *identifier)
{
  PerlInterpreter *my_perl = perl_alloc();
  char *perl_argv[] = { NULL, "handler_perl.pl" };
  int count;
  char *perl_function;
  HV *perl_myjob, *perl_identifier;
  dSP;

  syslog(LOG_INFO,"queue_id %d: %s: %s",myjob->queue_id,__FUNCTION__,myjob->identifier);

  myjob->results.value1 = 0;
  myjob->results.value2 = 0;
  myjob->results.status = AMON_JOB_CRITICAL;
  myjob->results.size = AMON_VAR_INT_32;

  perl_construct(my_perl);
  perl_parse(my_perl, xs_init, 1, perl_argv, NULL);

  perl_myjob = newHV();
  hv_store(perl_myjob, "queue_id", strlen("queue_id"),
    newSViv(myjob->queue_id), 0);
  hv_store(perl_myjob, "id", strlen("id"),
    newSViv(myjob->id), 0);
  hv_store(perl_myjob, "type", strlen("type"),
    newSViv(myjob->type), 0);
  hv_store(perl_myjob, "identifier", strlen("identifier"),
    newSVpv(myjob->identifier, 0), 0);
  hv_store(perl_myjob, "valuetype", strlen("valuetype"),
    newSViv(myjob->valuetype), 0);
  hv_store(perl_myjob, "contract_id", strlen("contract_id"),
    newSViv(myjob->contract_id), 0);

  perl_identifier = newHV();
  if(identifier->protocol != NULL)
    hv_store(perl_identifier, "protocol", strlen("protocol"),
      newSVpv(identifier->protocol, 0), 0);
  if(identifier->username != NULL)
    hv_store(perl_identifier, "username", strlen("username"),
      newSVpv(identifier->username, 0), 0);
```

```
if(identifier->password != NULL)
  hv_store(perl_identifier, "password", strlen("password"),
    newSVpv(identifier->password, 0), 0);
if(identifier->host != NULL)
  hv_store(perl_identifier, "host", strlen("host"),
    newSVpv(identifier->host, 0), 0);
if(identifier->port != NULL)
  hv_store(perl_identifier, "port", strlen("port"),
    newSVpv(identifier->port, 0), 0);
if(identifier->path != NULL)
  hv_store(perl_identifier, "path", strlen("path"),
    newSVpv(identifier->path, 0), 0);
if(identifier->query != NULL)
  hv_store(perl_identifier, "query", strlen("query"),
    newSVpv(identifier->query, 0), 0);

ENTER;
SAVETMPS;

PUSHMARK(SP);
XPUSHs(sv_2mortal(newRV_noinc((SV*) perl_myjob)));
XPUSHs(sv_2mortal(newRV_noinc((SV*) perl_identifier)));
PUTBACK;

asprintf(&perl_function, "handler_%s", identifier->protocol+5);
count = perl_call_pv(perl_function, G_ARRAY | G_EVAL);
free(perl_function);

SPAGAIN;

if(SvTRUE(ERRSV))
{
  STRLEN len;

  syslog(LOG_ERR,"queue_id %d: %s: perl: %s",myjob->queue_id,__FUNCTION__,SvPV(ERRSV, len));
}
else
{
  SV* result = POPs;

  if(SvROK(result))
  {
    HV* hash = (HV*) SvRV(result);

    if(SvTYPE(hash) == SVt_PVHV)
    {
      SV** fetch;

      if((fetch = hv_fetch(hash, "value1", strlen("value1"), 0)) != NULL)
```

```
        myjob->results.value1 = SvUV(*fetch);

      if((fetch = hv_fetch(hash, "value2", strlen("value2"), 0)) != NULL)
        myjob->results.value2 = SvUV(*fetch);

      if((fetch = hv_fetch(hash, "status", strlen("status"), 0)) != NULL)
        myjob->results.status = SvIV(*fetch);

      if((fetch = hv_fetch(hash, "size", strlen("size"), 0)) != NULL)
        myjob->results.size = SvIV(*fetch);
    }
    else
    {
      syslog(LOG_ERR,"queue_id %d: %s: perl: return value is no hash reference",
          myjob->queue_id,__FUNCTION__);
    }
  }
  else
  {
    syslog(LOG_ERR,"queue_id %d: %s: perl: return value is no reference",
        myjob->queue_id,__FUNCTION__);
  }
}

PUTBACK;
FREETMPS;
LEAVE;

perl_destruct(my_perl);
perl_free(my_perl);

syslog(LOG_INFO,"queue_id %d: %s: value1: %llu, value2: %llu",
    myjob->queue_id, __FUNCTION__, myjob->results.value1, myjob->results.value2);

return &myjob->results;
}
```