
GERMAN
PERL WORKSHOP
DRESDEN
MMXV

Auf geht‘s - zum 17. Deutschen Perl-Workshop

Am 06. Mai 2015 ist es soweit - der Deutsche Perl-Workshop kehrt nach 10 Jahren nach Dresden zurück. Die lokale Organisatoren der Dresdner Perlmongers haben ganze Arbeit geleistet und ein dreitägiges Programm zusammengestellt, das alle möglichen Ecken rund um Perl abdeckt.

An dieser Stelle schonmal ein dickes Dankeschön an alle Helfer und deren Familien, die in den vergangen Monaten immer wieder zurückstecken mussten. Und auch den Vortragenden gilt unser Dank, denn ohne Vorträge wäre es die nächsten drei Tage recht still ;-)

Jetzt soll es aber endlich losgehen...

Wir wünschen viel Spaß beim Deutschen Perl-Workshop 2015

Perl für die Hosentasche

Autor

Max Maischein `corion@corion.net`

Bio Max Maischein

Der Autor entwickelt seit 15 Jahren in Perl und hat einen Schwung Module auf dem CPAN veröffentlicht. Mit der Entwicklung von Anwendungen unter Android beschäftigt er sich seit 2014.

Perl für die Hosentasche

Die Miniaturisierung von Elektronikkomponenten schreitet immer weiter voran. In 2008 hatte ein Mobiltelefon etwa die Leistung eines PCs von 1998, passt aber in die Hosentasche. In 2015 hat sich die Leistungsfähigkeit von Mobiltelefonen und integrierten Kleinstrechnern ohne Mobilteil weiter gesteigert. Im Vergleich zu Systemen, die sich eher an Elektronikbastler richten wie zum Beispiel der Raspberry Pi oder Beaglebone Platinen, bieten fertige Android-Systeme dank der Massenproduktion ein deutlich besseres Preis/Leistungsverhältnis, wenn es um die „übliche“ Konnektivität geht, also Stromversorgung, Bluetooth, Wlan und Touchscreen. Da ich die erweiterten Ein- und Ausgabemöglichkeiten über herausgeführte I/O-Ports der Selbstbau-Platinen von Raspberry Pi etc. nicht benötige, liegt es also nahe, sich mit Android als Plattform zu beschäftigen.

Mein langfristiges Ziel der Entwicklung unter Android ist einerseits, Daten mit Perl von meinem Mobiltelefon auszulesen statt sie über das Internet mit einem Cloud-Dienst zu synchronisieren. Andererseits möchte ich eine Anwendung entwickeln, die Netzwerkverbindungen physisch greifbar macht. Statt alle (Medien-) Daten auf einen der vielen Cloud-Dienste hochzuladen und dann bei Freunden über den Heimrechner abzuspielen, soll ein kleines Gerät ins Heimnetzwerk eingebunden werden und über das Internet Verbindung mit meinem privaten Rechner aufnehmen.
Das Gerät soll dann meine Mediendaten ohne weitere Konfiguration im Netzwerk der Freunde verfügbar machen, zum Beispiel über den USB-Anschluss oder als Medienserver. Sobald das Gerät abgeschaltet ist oder von mir wieder mitgenommen wird, sollen auch die Mediendaten nicht mehr zur Verfügung gestellt werden.

Natürlich deckt bereits eine USB-Festplatte 90% der Aufgabe ab, da ich einfach nur alle Daten jeweils mit der USB-Festplatte synchronisieren müsste. Da ich aber vergesslich bin und im Zweifelsfall gerade die im Moment wichtigen Daten nicht synchronisiert sind, soll eben eine Online-Verbindung zu meinem Heimnetzwerk hergestellt werden.

Ausgangslage für Perl

Die Hautpentwicklungssprache für Android ist Java. Seit 2009 unterstützt Google für Android aber auch sogenannte „native“ Anwendungen, also Anwendungen, die direkt in den Maschinencode der verwendeten CPU kompiliert wurden. Da Perl im wesentlichen in C geschrieben ist, ist dies der Weg der Wahl um Perl in einer Android-Umgebung zu verwenden.

Mit Perl 5.20 hat Perl durch Brian Fraser viel Unterstützung für die Kompilation unter Android geschaffen. Neben den Anpassungen am Code von Perl gibt es hier insbesondere die Dokumentation `pod/perlandroid.pod`, welche die Android-spezifischen Schritte zur Kompilation von Perl beschreibt. Wichtigster Bestandteil für die Entwicklung von und mit Perl unter Android ist ein kompatibler C Compiler. Die Android-App „CCTools“ bringt einen angepassten `gcc` sowie die für Unix üblichen Anwendungen wie eine Shell, `vi`, aber auch nützliche Programme wie `wget` und `curl` mit, so daß man direkt Perl kompilieren kann.

Die CCTools funktionieren auf jedem Android-Gerät, welches einen ARM-7 kompatiblen Prozessor hat. Dies deckt viele Geräte ab, insbesondere auch günstige Geräte mit Mediatek Chipsatz.

Weiterhin hat CCTools bereits eine vorkompilierte Version von Perl dabei, allerdings ist die installierbare Version 5.18.2, welche nicht alle Anpassungen für Android beinhaltet. Um möglichst viel von der bereits von Brian Fraser für 5.20 geleisteten Arbeit zu profitieren, möchte ich also ein Perl 5.20, möglicherweise aber auch eine aktuelle Entwicklungsversion 5.21.x und ab Mai dann 5.22 kompilieren.

Das Rechtekonzept von Android sieht vor, daß es pro Anwendung einen eigenen Unix-User gibt. Je nach dem, aus welcher Terminalanwendung heraus wir Perl also starten, wird unser Prozeß unter verschiedenen Usern laufen. Eine gemeinsame Datenhaltung oder der Zugriff auf Daten anderer Android-Anwendungen benötigt also entweder den Root-Zugang oder aber Datenhaltung auf der Speicherkarte.

Kompilieren und Entwickeln unter Android

Die eigentliche Idee der Entwicklung für Android ist die der Cross-Kompilation auf einem Desktop-PC und des Testens im Emulator oder einem per USB-Kabel angeschlossenem Android-Gerät. Das Aufsetzen einer solchen Umgebung zusammen mit den Schwierigkeiten, Perl auf einer Plattform für eine zweite Plattform zu kompilieren und testen kombiniert mit der Leistungsfähigkeit und ständigen Verfügbarkeit meines Mobiltelefons bringt mich dazu, die Entwicklung komplett auf meinem Mobiltelefon zu probieren.

Dank des technischen Fortschritts dauert das Kompilieren von Perl mit `gcc` und den Hilfsprogrammen der CCTools auf meinem Mobiltelefon nicht wesentlich länger als auf einem Desktop. Erstaunlicherweise wird die Batterieleistung des Mobiltelefons durch das Kompilieren auch nicht wesentlich beeinträchtigt.

Mit einer Snapdragon 810 CPU und 2GB RAM im Mobiltelefon dauert das Kompilieren etwa 40 Minuten. Auf einem Intel i7 dauert das Kompilieren zum Vergleich etwa 15 Minuten. Die Schritte für das Kompilieren von Perl unter Android sind in der Datei `INSTALL` dokumentiert und werden weitestgehend automatisch festgelegt. Einige Anpassungen sind momentan noch notwendig:

```
sh ./Configure -Dprefix=/storage/sdcard0/perl
  -des \
  -Dusedevel \
  -Dsysroot=/data/data/com.pdaxrom.cctools/root/cctools \
  „-Alibpth=/system/lib /vendor/lib“ \
  -Dman1dir=none \
  -Dman3dir=none \
  -Dsite1man=none \
  -Dsite3man=none \
  -Dvendorman1=none \
  -Dvendorman3=none \
  -DNO_LOCALE
```

`prefix` wird auf das Zielverzeichnis gesetzt, in welchem Perl letztendlich installiert werden soll.

`sysroot` ist das Verzeichnis, in welchem CCTools die Hilfsprogramme wie `gcc`, `ar` und `make` abgelegt hat.

`NO_LOCALE` muß gesetzt sein, da Android keinerlei Locale-Funktionalität anbietet.

Nach dem Konfigurieren und Kompilieren erfolgt der Selbsttest von Perl mit

```
# make test
...
Failed 16 tests out of 2190, 99.27% okay.
   ../cpan/Memoize/t/expmod_t.t
   ../cpan/Sys-Syslog/t/syslog.t
   ../cpan/Time-Piece/t/02core.t
   ../dist/Net-Ping/t/110_icmp_inst.t
   ../dist/Net-Ping/t/120_udp_inst.t
   ../dist/Net-Ping/t/130_tcp_inst.t
   ../dist/Net-Ping/t/140_stream_inst.t
   ../dist/Net-Ping/t/150_syn_inst.t
   ../dist/Net-Ping/t/450_service.t
   ../dist/Net-Ping/t/500_ping_icmp.t
   ../dist/Net-Ping/t/510_ping_udp.t
   ../dist/Net-Ping/t/520_icmp_ttl.t
   ../lib/warnings.t
   op/tie_fetch_count.t
   op/time.t
   porting/globvar.t
```

Im Testlauf schlagen ein paar Tests fehl. Diese Tests werte ich mit Hilfe von Google ein. Eine Erfolgsquote von etwas über 99% ist schon gut und die meisten Tests sind wahrscheinlich eher auf Inkompatibilitäten der Tests denn auf Probleme mit Perl zurückzuführen.

```
t/op/tie_fetch_count ..........................................
The crypt() function is unimplemented due to excessive paranoia.
at op/tie_fetch_count.t line 253.
# Looks like you planned 347 tests but ran 185.
FAILED--expected 347 tests, saw 185
...
t/op/tiehandle ................................................ ok
t/op/time ..................................................... #
Failed test 7 - changes to $ENV{TZ} respected at op/time.t line 62
FAILED at test 7
t/op/time_loop ................................................ ok
...
cpan/Time-Piece/t/02core ......................................
#   Failed test at t/02core.t line 62.
#          got: ,3600‘
#     expected: ,-18000‘
#   Failed test at t/02core.t line 63.
#          got: ,CET‘
#     expected: ,EST‘
# Looks like you failed 2 tests of 95.
FAILED at test 36
```

Diese Tests haben falsche Erwartungen an Android. `crypt()` ist nicht implementiert und muss demzufolge aus dem Test ausgeschlossen werden. Zeitzonen werden von der C Bibliothek nicht wie gewohnt unterstützt, die Tests müssen also übersprungen werden.

```
cpan/Sys-Syslog/t/syslog ..........................
#   Failed test ,[stream] syslog() called with level ,info‘ (string)‘
#   at t/syslog.t line 179.
#          got: ,Your vendor has not defined POSIX macro LC_TIME,
           used at ../../lib/Sys/Syslog.pm line 436
# ,
#     expected: ,‘
```

Hier verwendet ein Test oder das Modul die Funktion `LC_TIME()`, welche von Android nicht zur Verfügung gestellt wird. Hier muss also mittelfristig das Modul angepasst werden und idealerweise auf die Verwendung von `LC_TIME` verzichten oder eine optionale Alternative zu `LC_TIME` verwenden.

Installation von CPAN Modulen

Einer der großen Vorteile von Perl ist die große Zahl frei verfügbarer Module auf CPAN. Angenehmerweise funktioniert der CPAN Klient, so daß der Verwendung erstmal nur Eigenheiten von Android entgegenstehen.

Zugriff auf Anwendungsdaten - DBD::SQLite

Viele Android Apps speichern mit der Datenbankbibliothek SQLite ihre Daten. Insbesondere die Notizanwendung von Google, Keep, speichert ihre Daten lokal auf dem Mobiltelefon in einer SQLite Datenbank. Daher liegt es nahe, als eines der ersten Module `DBD::SQLite` zu installieren. Die dabei ausgegebene Fehlermeldung führe ich darauf zurück, daß die Tests nicht damit rechnen, daß Perl keine Locale-Unterstützung haben könnte:

Sobald der Datenbanktreiber für SQLite installiert ist, habe ich unter dem `root`-User auf dem Mobiltelefon direkten programmatischen Zugriff auf die Daten der Notizverwaltung und kann die Daten mit eigenen Programmen manupulieren [1].

Webserver - Dancer oder Mojolicious

Da bisher keine Möglichkeit existiert, von Perl aus auf die Android-Bibliotheken für grafische Benutzersteuerung zuzugreifen, versuche ich, mit CPAN die beiden Webframeworks Dancer und Mojolicious zu installieren. Dancer stützt sich bei seinen Bibliotheken auf CPAN ab, während Mojolicious eher die Strategie verfolgt, möglichst wenige Bibliotheken zu verwenden, die nicht schon mit Perl mitgeliefert werden.

Dancer stolpert leider in einer Kette verwendeter Module über `IO::File`, welches in `IO::File->new_tempfile()` davon ausgeht, dass es unter Android ein Verzeichnis für temporäre Dateien gibt. Ein solches Verzeichnis gibt es zwar, aber nicht unter `/tmp` und es gibt auch keine Umgebungsvariable, die auf den Namen des Verzeichnisses hinweist.

Der Pfad zum Verzeichnis lässt sich ausschliessliche über einen Aufruf an die Java Bibliotheken von Android ermitteln. Dies ist Perl derzeit noch unmöglich, daher kann Dancer auch vorerst nicht sinnvoll verwendet werden ohne hier einen Ausweg zu programmieren.

Integration von Perl in Android

Android reicht als Plattform aus, um Perl Programme laufen zu lassen. Um allerdings auch echte Android-Anwendungen mit Perl zu schreiben, sind noch weitere Schritte nötig, die ich hoffentlich in naher Zukunft angehen kann.

Zugriff auf Android APIs

Native Programme laufen gänzlich separat und separiert von der Java Virtuellen Maschine. Android stellt die meisten seiner APIs allerdings als Java Klassen zur Verfügung, so daß ein direkter Zugriff von Perl auf diese APIs ausgeschlossen ist. Eine mögliche Lösung ist hier eine Java-Anwendung, die die Aufrufe macht und die Ergebnisse über eine lokale Interprozesskommunikation an ein Perl-Programm zurückliefert.

Android-Apps mit Perl

Echte Apps lassen sich mit Perl nicht leicht schreiben, da zusammen mit der App auch die jeweilige Installation von Perl und den Modulen mit ausgeliefert werden müsste, selbst wenn die Kommunikation von Perl mit der Android-API gelöst ist. Zwei mögliche Ansätze hier sind Marc Lehmanns `App::staticperl` und `App::fatten`, die jeweils versuchen, ein Programm und die benötigten Module zusammenzupacken. `App::staticperl` kompiliert dazu sogar noch ein minimales Perl, mit dem eine Auslieferung über den Play Store zumindestens denkbar wird.

FUSE

Android verwendet für interne Laufwerke und Speicherkarten einen FUSE-Dämon. Idealerweise kann auch Perl über das *Fuse*-Modul über Dämon eigene Dateisysteme zur Verfügung stellen. Das Ergebnis ist dann ein USB-Gerät, dessen Inhalte direkt von Perl erzeugt werden. Der Host-Rechner benötigt keine zusätzlichen Treiber sondern nur ein USB-Kabel um die Ausgabe von Perl lesen zu können.

Statusuebersicht Perl VMs

German Perl Workshop, Dresden 7.5.2015
40 min
Reini Urban, cPanel

Nachdem es aus der Sicht des Vortragenden mit perl5 nicht zufriedenstellen aussieht, wird es in diesem Vortrag einen kleinen Überblick über die Probleme, die Pläne diese zu lösen, und einen Stand der Dinge bei p2, rperl, parrot, gperl und dem B::C* Compiler geben.

perl 5.22

• internal performance is up by 1.8. new multideref and maybe a new signatures op and esp. cached class pointers. compile-time known class and method names are now pre-initialized into the class cache. SUPER::new is now parsed at compile-time.
• Unicode 7.0
• UNIVERSAL->import (e.g. use UNIVERSAL ‚...') is now fatal
• -fstack-protector-strong as new default made perl slower, and the added security is questionable.
• empty sub bodies containing only undef are now „inlined"
• non-utf8, non-magic length is now 20% faster
• in @array = split the assignment can now be almost always optimized away.
• lots of more such lvalue assignment optimizations with other ops. (but still buggy in corner cases)
• lots of fixes for NaN, Inf and locale dependent. NaN handling is now pretty good, when supported by the system.
• NV (double numbers) can now be stored in the head only, no need for a seperate body.
• warnings ‚all" is not all anymore, added more warnings via extra, all is now everything. used for contro versial warnings: grep in void context. Interestingly I wrote the same patch for some preloaded registered modules, but backed out some months ago. Now it's in. (My warnings are compiled into a extendable read-only hash in a shared library)
• new optional OP_PARENT, op_sibling and op_lastsib is gone. rewritten pad API (again). actually the pad API and OP_PARENT are all very good things.
• use re ‚strict'

syntax

• <<>> treats the @ARGV name as literal, same as the 3 argument open. that means <> was not fixed to ignore shellchars in filenames (e.g |foo), same as 2 arg open was also never fixed. perl5 is still a security minefield.
• new refaliasing. the lhs of an alias may now be a reference, and is automatically dereferenced:
```
\$c = \$d,
foreach \%hash (@array_of_hash_refs) { ... keys %hash }
```
• new bitwise string operators (but still buggy)
• sub signatures parsed before attributes.
• defined(@array) and defined(%hash) are now fatal, but defined(@array = list) not.
• literal { needs to escaped in a regex
• prefer subs over barewords with the * prototype
• constant inlining of sub () { $var } may now throw deprecation warnings when the $var refers to a closed over variable, and a side-effect to change $var was ignored or not detected. it produces however still wrong code.

problems

(only a very tiny outline)

signatures are still not acceptable.
- no types support _(a 4 lines patch)_
- no call-by-ref _(a much longer patch)_
- still copying the valus to @_ even when not needed, not using MARK yet as in the ops. *(a trivial patch on top of OP_SIGNATURE)*
- too slow arity check at run-time. nothing done at compile-time. *(changes perldiag)*
- only primitive optimizations possible. *(not enough attributes)*
- not merged with old protoypes. need now to announce as experimental, even if not necessary, which disables old prototypes syntax. this was not the plan.

Nested subs and lexical values in subs are still not compiled properly in corner cases.

regex compiler still a mine-field (e.g. ops via longjmp), still not using non-backtracking optimizations (DFA re2), still not using dynamic compilation, i.e. two pass compilation: 1st for size and 2nd for the ops. PCRE has a jit in the meantime. Perl devs have no idea still.

silly new double readonly system. *(proper patch blocked)* silly old hash table and new, slower hash function implementations. *(fruitless discussion)*

horrible old parser and new syntax extension system.

With p5p no progress is possible, only bugfixes. They are basically just maintaining the status quo, and even that not properly. Most bugs are not getting fixed.
Not a single successful perl5 feature implemented or managed by p5p was ever successful since Larry left. Wait, except one. Which? *(defined-or)*

parrot

Basically the same management problems as p5p after the core developers left after excessive infighting. then, from 1.0 to 4.6 only destruction being done, much worse than the infighting before.

parrot performance is still terrible, but this is fixable. It was fast before some people decided to destroy performance and compatibility, for aesthetic reasons and lack of management.

After all the people left I had the chance to fix it. It has a superior threads architecture after all. Only owner may write, the compiler automatically knows the owning thread and assigns proxy objects to accessing threads. So it‘s basically lockless for all object types. Writes are done via prioritized deferred writer tasks in the owner thread.

I got it much faster, with the GSOC boost in summer and many more internal improvements. Basically some of the damage done in the 2.7 - 3.6 period was undone, but there are still some big damages to undo. The calling convention/runloop slowdown (20%), fixing string compacting GC (20%) and the jit (100%). About 6 month work, but I stepped back again after the parrot/perl6 fiasco(*), working on our perl5 project now.

moar

The MoarVM project has my full sympathy as some parrot devs just had no idea what they were doing, and it only got worse.

Got stable and incredibly fast, even with a good optimizing jit now. Working on NFG and the great list refactor now.

perl 6 will go 1.0

It already is out of beta and IMHO production ready for quite some time, but they decided to rework list comprehension and enable unicode normalization (NFG) in the remaining months.

parrot support was discontinued for not understandable reasons. the cited reasons were bogus and never reported. there was no ticket, so we never knew, prioritization could not be done and when we heard about it, it was already too late for them. apparently nobody but froggs wants to work on the parrot backend. you will not care, but you should if you care about multithreaded performance. moar and jvm do not help there.

Technically cited was lack of Unicode NFG support, but no other VM has NFG support either, and with parrot it‘s much more trivial to use the existing icu Normalization2 API, which moar has to do now manually.

Alternative vm‘s

Remember: Compare this to the healthy state of other dynamic languages VM‘s: php, ruby, python, javascript

p2

p2 was my project to use a good small dynamic vm, potion written by why the lucky stiff, to use as perl5 and perl6 backend. the features were already better than moarvm and parrot and of course the old perl5 vm. function calls are 200x faster, nice jit, full dynamic MOP. e.g. I implemented BEGIN block support in one line of code.

In the last year I started adding a debugger, compiler framework, and type support, but got frustrated with the not yet GC-safe parser, complete lack of interest (or competence), and strange performance characteristics in some benchmarks.

gperl

gperl is a llvm backed parser and jitted runtime by gocchi from Japan. even faster than p2. He still has a mind-blocking bug in the parser with eval, and stepped back to work on submodules.

rperl

Is the 2nd perl11 project cited here. it is a restricted perl, which favors typed, no-magic data structures, which can be compiled to pure C++ stdlib structures, and some operations on those are therefore as fast as possible. It has it‘s own parser for the extended syntax, and can fallback from C++ ops to normal dynamic perl ops. Standalone it is already usable, which is a major accomplishment.

The further plan is to compile modules into shared libs and use such compiled libs, compiled with

• the standard and stable B::C , but during run-time not faster, fully dynamic perl5 vm,
• type optimized B::CC libraries, unrolled runloop, most ops optimized.
• type optimized rperl compiled into C++.

via perlcc and a new buildcc tool to manage the dependencies.

e.g. with cPanel a typical binary is 50MB, has 200 modules dependencies, has almost zero startup-time(*), perl5 bound run-time, and again almost zero destruction-time. most modules are already compiled in (i.e. just mapped into the process space, even if not used), shared libs are loaded at startup. only some bigger modules not constanty used are run-time loaded from source, not via `.pmc` yet.

our internal perl5 project

In cPanel we work on a better perl5 along with the better perl compiler (zero startup and destruction time, and optimized run-time). I outlined a lot in perl5 TODO and compiler blog posts some years ago and did some YAPC talks about it. Now after the parrot and p5p fiascos I actually started working on it. 5.14 looked like a good enough version to keep, but now with Syber‘s massive OO improvements 5.22++ is an attractive enough target to switch to.

Better type support (coretypes, like compile-time checked sized and typed arrays), but also classes, methods, roles (i.e. compile-time composable classes), signatures, type checker and inferencer, static GVs, and lot of internal improvements we consider important, without considering the p5p deadlock at all.

Static GVs is the only problem cited above in almost zero startup-time and destruction time. It is unfortunately non-zero yet, we want to make it zero. So far it is just 10x faster.

Types will get you a run-time performance boost of about ~2-3x faster. And will make typed modules incompatible to perl5 proper. I‘ve tried, sorry. It will help our huge codebase for added security and documentation. Experimental benchmarks were 6-8x faster, almost getting to v8 speeds, bypassing the similar python and ruby compilers.

p5p has already stated that it has no interest. They are actively blocking type support which only exists for lexicals to be added to signatures, so it will not be backwards compatible. They were also blocking dynamic optimizations, so v8-like profile-guided run-time optimizations are not possible with perl5 proper. (basically improving hot loops and method calls).

The good thing: p5p voiced interest into merging „p5-mop“, which is basically the same, but nobody but me is working on that for core integration. Stevan Little gave up 2x so far out of frustration and his current goal is not interesting at all anymore. His current feature set in p5-mop v3 is dumbed down and not performant anymore.

I got now probably Yuval Kogman to help working on our version. Most of the perl6 object and type system can be easily implemented within perl5. The question is how much. Not too much, but not too less either. Roles and method combinations yes, the full type tree not, only basic coretypes. The full MOP probably not, or maybe as extensions. The new phases not.

Modellierung mathematischer Objekte in Perl

Abstract

Dank Kryptographie haben mathematische Begriffe wie endliche Körper und elliptische Kurven Eingang in alltägliche Applikationen gefunden. Der Autor arbeitet derzeit daran, einige dieser seltsamen, sonst tief in C-Bibliotheken vergrabenen Konstrukte in Perl-Modulen zur Verfügung zu stellen.

Nach Polynomen (Math::Polynomial 1.000, 2009) und Restklassen (Math::ModInt, 2010) folgt nun ein Modul für endliche Körper (Math::GaloisField, 2015), auf das später elliptische Kurven über endlichen Körpern (Math::EllipticCurve::GaloisField) aufbauen können.

Operationen auf und Beziehungen zwischen diesen Objekten lassen sich gut mit einer objektorientierten API darstellen. Entscheidend für die Nützlichkeit des neueröffneten mathematischen Zoos sind jedoch auch manche eher knifflige Design-Entscheidungen. Zu den Prinzipien, die sich dabei herausgebildet haben, gehören die Folgenden:

• Klassenhierarchien einfach halten
• zahlen-ähnliche Dinge als unveränderliche Objekte darstellen
• für aufzählbare Dinge Iteratoren bereitstellen
• zyklische Referenzen vermeiden oder nach Gebrauch bereinigen
• aufwendige Berechnungen isolieren
• umfangreiche vorab berechnete Daten auslagern
• vorhandene Limitierungen deutlich erkennbar machen
• weitestgehende Einheitlichkeit anstreben

Der Vortrag richtet sich an leicht fortgeschrittene Perl-Programmierer mit Interesse an der Entwicklung eigener Bibliotheken. Die erwähnten Module sind objektorientiert, ohne die von *Moose* und *Perl 6* geprägte moderne Syntax zu verwenden, was im Verlauf auch begründet wird.

Anhang: Beispiele endlicher Körper

Für Elemente endlicher Körper sind wie für rationale oder reelle Zahlen die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division definiert. Das Schöne an diesen endlichen Räumen ist, dass sie beim Rechnen nicht verlassen werden, d.h., wenn zwei Operanden in den Speicher passen, gilt dies auch für das Resultat. Zur Veranschaulichung folgen drei sehr kleine Beispiele.

GF(7)

Die 7 ist eine Primzahl, daher ist der Restklassenring modulo 7 ein Körper.

```
+ | 0 1 2 3 4 5 6      * | 0 1 2 3 4 5 6
--+-------------      --+-------------
0 | 0 1 2 3 4 5 6      0 | 0 0 0 0 0 0 0
1 | 1 2 3 4 5 6 0      1 | 0 1 2 3 4 5 6
2 | 2 3 4 5 6 0 1      2 | 0 2 4 6 1 3 5
3 | 3 4 5 6 0 1 2      3 | 0 3 6 2 5 1 4
4 | 4 5 6 0 1 2 3      4 | 0 4 1 5 2 6 3
5 | 5 6 0 1 2 3 4      5 | 0 5 3 1 6 4 2
6 | 6 0 1 2 3 4 5      6 | 0 6 5 4 3 2 1
```

GF(8)

Die 8 ist eine Zweierpotenz. Ein Körper mit 8 Elementen hat also die Charakteristik 2. Hier ist jedes Element sein eigenes additives Inverses. Wir wählen eine von mehreren Möglichkeiten, die Elemente dieses Körpers zu nummerieren, und erhalten:

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

*	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

GF(9)

Als letztes Beispiel zeigen wir noch einen Körper mit 9 Elementen, also Charakteristik 3.

+	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	2	0	4	5	3	7	8	6
2	2	0	1	5	3	4	8	6	7
3	3	4	5	6	7	8	0	1	2
4	4	5	3	7	8	6	1	2	0
5	5	3	4	8	6	7	2	0	1
6	6	7	8	0	1	2	3	4	5
7	7	8	6	1	2	0	4	5	3
8	8	6	7	2	0	1	5	3	4

*	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	1	6	8	7	3	5	4
3	0	3	6	4	7	1	8	2	5
4	0	4	8	7	2	3	5	6	1
5	0	5	7	1	3	8	2	4	6
6	0	6	3	8	5	2	4	1	7
7	0	7	5	2	6	4	1	8	3
8	0	8	4	5	1	6	7	3	2

Autor

Dipl.math. Martin Becker studierte Mathematik und Informatik in Ulm. Er leitet die Qualitätssicherung beim Cloud-Service-Provider ScanPlus.

Copyright

Web-APIs entwickeln & testen: Vom curl-Einzeiler zur MooseX::App

Bio Daniel Böhmer

Daniel Böhmer ist Informatiker und arbeitet seit 2013 als freiberuflicher Software-Entwickler mit Schwerpunkt Perl. Er unterstützt vorrangig Entwickler-Teams beim Einsatz von Perl, Linux, Git etc.; und entwickelt für Unternehmer Software, die ihre Betriebsabläufe organisiert. Server-Dienste und Web-Anwendungen natürlich am liebsten in Perl!

Zu seinem Leben gehören außerdem seine Ehefrau und der Alltag in ihrer evangelisch-freikirchlichen Gemeinde in Leipzig.

Abstract

Beim Entwickeln einer Web-API (z.B. eines RESTful Web-Service) muss man mitunter Aufrufe per Hand testen. Unter *nix ist das Tool der Wahl `curl`: schnell einen Einzeiler auf der Shell geschrieben, im JSON vergessene Anführungszeichen nachgetragen, Tippfehler in der URL korrigiert und eine Hand voll curl-Optionen angegeben - schon klappt es!

Dem Autor war das schnell zu umständlich und er zeigt, wie man mit MooseX::App und bspw. REST::Client schnell einen kleinen, komfortablen Client baut.

Links

http://www.daniel-boehmer.de/2015/gpw/
Unterlagen zum Vortrag, Links, Beispiel-Code

https://metacpan.org/release/MooseX-App
Die vorgestellte CPAN-Distribution MooseX::App

https://metacpan.org/release/REST-Client
Das Modul, auf dem der Beispiel-Client basiert

https://metacpan.org/release/Dancer2
Das Dancer2 Web-Framework, auf der Beispiel-Web-Service basiert

https://www.xing.com/profile/Daniel_Boehmer11
XING-Profil des Autors

perlall

do something with all my perls

Reini Urban / rurban houston.pm, 2012

Design Goals

parallel perl installations as supported by Configure *(or not).*
no chdir or switching to other perls *(better perlbrew)*

/usr/local/bin/perl5.15.8
/usr/local/bin/perl5.14.2
/usr/local/bin/perl5.12.4
/usr/local/bin/perl5.10.1
/usr/local/bin/perl5.8.9
/usr/local/bin/perl5.8.8
/usr/local/bin/perl5.8.5
/usr/local/bin/perl5.8.4
/usr/local/bin/perl5.6.2

Design Goals

Build, maintain and test with a lot of perls, on a lot of machines.
Your private cpantesters framework.

Standardize feature names: DEBUGGING and threads.

Single file - scp to all remote vm's. Only 4 external modules - self-installer independent on CPAN.

perlall vs perlbrew

Globally shared non-arch modules	Private non-arch - good for usedevel testing
manual installation process (root, sudo or non-sudo)	automatic bash installation
Not only build. Use it: init cpan cpanm maketest makeinstall, ...	No parallel usage
MSWin32, msys	No MSWin32, what is msys?

Common tasks: setup + test + install

```
for p in 6.2 8.4 8.5 8.8 8.9 10.1 12.4 14.2 15.8
do
  perlall build 5.${p};  perlall build 5.${p}-nt
  perlall build 5.${p}d; perlall build 5.${p}d-nt
  perlall build 5.${p}-m
done                    # and wait ~2h

perlall list            # what failed?
perlall=5.15.* perlall cpanm -f --sudo YAML
perlall init            # installs e.g. Bundle::CPANReporter2
                                      # wait and prompt for the next 2hrs

cd ~/Perl/B-Generate
perlall maketest
perlall makeinstall     # generate and upload cpanreports
```

Creating test reports

```
cd ~/Perl/B-Generate

perlall maketest              local

perlall testvm --all                and remote (vm or phys)


=> log.test-osx10.6.8-5.10.1d-nt, log.test-linuxmint1-5.15.8d-
nt, log.test-freebsd7.4-5.10.1, log.test-cygwin1.7.10s_winxp-5.14.2d-nt,
... (> 50 logfiles)


$ ../B-C/store_rpt            save reports away
```

Working with test reports

$ ../B-C/status_upd -fqd

cygwin1.7.10s_winxp-5.14.2d-nt:
t/cc.t Failed tests: 10, 38, 46, 101
t/e_perlcc.t Failed tests: 22, 52

linuxmint1-5.14.2-m:
t/c_o1.t Failed test: 15
t/c_o2.t Failed test: 15
t/c_o3.t Failed test: 15
t/c_o4.t Failed test: 15
t/cc.t Failed test: 15
t/e_perlcc.t Failed tests: 53..54

linuxmint1-5.15.8d:

linuxmint1-5.15.8d-nt:
t/c_o1.t Failed test: 15
t/c_o2.t Failed test: 15
t/c_o3.t Failed test: 15
t/e_perlcc.t Failed tests: 53..54

Working with test reports

Download external cpanreports:

$ ../B-C/t/download-reports 1.43

And check all reports, yours and theirs:

$ ../B-C/status_upd -fqd t/reports/1.43

-fqd *fail only, quiet, no dump display (broken)*

Reports are created by:

```
make test TEST_VERBOSE=1 2>&1 | tee log.test
git diff >> log.test
perl -V  >> log.test
```

Cfg and Shortcuts in ~/.perlall

.perlall is in bash format. source it from your .profile

env and alias

```
PERLALL_BUILDROOT=~/perl5/src
alias perl-git='cd /usr/src/perl/blead/perl-git'
# currently used perl (set by perlall)
alias p=perl5.15.8d-nt
```

Shortcuts in ~/.perlall

```
# some aliases suggestions
alias pb="p -Iblib/arch -Iblib/lib" # no -Mblib!
alias pmf="if [ -f Makefile.PL ]; then p Makefile.PL; else rm -rf _build; p Build.PL; fi"
alias pm='pmf && m'
alias ppan='p -S cpan'

alias m=make
alias mt='make -j4 test'
alias mi='mt && smi'
alias mtee='mt 2>&1 | tee log.test'
alias smi='sudo make install'

#set p alias from current Makefile
function ppm { p=$(perl -ane'print $F[2] if /^FULLPERL =/' Makefile); echo alias
p=$p; test -n "$p" && alias p=$p; }
alias pgrp='pgrep -fl perl'
```

Typical test session

alias pb="p -Iblib/arch -Iblib/lib"
alias pmf="if [-f Makefile.PL]; then p Makefile.PL; else rm -rf _build; p Build.PL; p Build; fi"
alias pm='pmf && m'

```
$ pm                        # make with your current perl
$ pb t/02failing_test.t     # single test
$ pb -d t/02failing_test.t  # debug it

$ mt                        # test with this perl
$ perlall -m --nogit maketest  # all major perls
```

perlall do

```
# who has no YAML?
perlall do -MYAML -e0

# install on older versions
perlall=5.8.* perlall cpanm -f -S YAML

# check memory leaks, with fresh make before
# use current p as \$p      (pb not, sorry)
perlall make '-e1 && valgrind \$p -Mblib test.pl'
```

perlall make does always a do, i.e $p $@

testvm

Need to setup every vm, with perlall of course. See INSTALL.
ssh-copy-id your .ssh key, adjust .profile or .bashrc.
Create the same layout as on the master for your work modules.

```
ssh win mkdir -p Perl/MyModule
perlall testvm win centos5 freebsd10 --fork
```

If *win* is on a vm, then the vm is started/resumed. (kvm only so far). Switched from vmware,xen,virtualbox to kvm.
With -j4 ensure that max. 4 vm's run concurrently. Memory and IO pressure destabilizes the system, esp. with cgroups and 3.x kernels.
TODO: Automated mingw/activeperl testing. ssh with cmd.exe?

New perl release

```
perl-git                        # cd to git srcdir
git co blead; git fetch         # get new tags
perl-root                       # cd to buildroot

# for testing
perlall build 5.15.8d
perlall build 5.15.8d-nt
# for benchmarking
perlall build 5.15.8-nt
# start CPAN update
perlall=5.15.8* perlall init
# how does it look like?
cd ~/Perl/MyModule
alias p=perl5.15.8d-nt
pm && mtee
```

Test blead

perl-git # cd to git srcdir
git co blead; git fetch # get latest
perlall -v build **blead**d-nt **--link** -Dcc=gcc-mp-4.7
=> /usr/local/bin/perl5.15.8d-nt@8129baca installed

blead - magic version, --link -Dmksymlinks to perl-git

perlall -v build bleadd-nt --link -Dcc=clang \
-Dld=clang **--as** perl5.15.8d-nt**-clang**
...test error...
perlall -v build bleadd-nt \
--as perl5.15.8d-nt-clang **--install** # continue

perlall build bleadd-nt smoke-me/khw-tk # branch
=> perl5.15.8d-nt@khw-tk

Planned

perlall smoke -j4 bleadd-nt smoke-me/*

=> TODO: send smoke reports

perlall=5*[0-9]-nt perlall -m bench t/benchscript.t

=> TODO: stable Benchmark.pm (check load, wait until stable)

perlall cpan Devel::*Prof*

=> query CPAN for matching modules. *metacpan not yet good enough. CPAN is better*

cpan App::perlall

```
cd ~/Perl
git clone https://github.com/rurban/App-perlall.git
cd App-perlall
perl Makefile.PL && make test install
cd ~/bin
ln -s ~/Perl/App-perlall/scripts/perlall

ln perlall perlall-maketest
ln perlall perlall-do
ln perlall perlall-cpan
ln perlall perlall-init

# See INSTALL
```

Tuning Algorithm::Diff

Autor: Helmut Wollmersdorfer

Deutscher Perl-Workshop 2015

Problem

- Vergleich zweier Sequenzen
- Ausrichtung mit maximaler Ähnlichkeit
- Ergebnis als Positionen (Flexibilität)
- Sequenzen als Arrays of Strings
- Elemente: Chars, Wörter, Zeilen etc.

Ausrichtung

```
Chrerrplzon
Choerephon
```

0	**1**	2	**3**	**4**	5	**6**	7	8	**9**	**10**
C	**h**	r	**e**	**r**	r	**p**	l	z	**o**	**n**
|	|		|	|		|			|	|
C	**h**	o	**e**	**r**	e	**p**	h		**o**	**n**
0	**1**	2	**3**	**4**	5	**6**	7		**8**	**9**

```
0 1 3 4 6 9 10
0 1 3 4 6 8  9
```

Vorteil durch Beschränkung

- Alt: `LCSidx(\@s1,\@s2,\&hash,\&cmp)`
- Neu: `LCS(\@s1,\@s2)`

Ersparnis:

- ~35 LoCs
- Prüfen und Calls der Code-Refs (langsam)

Twiddling

```
sub _withPositionsOfInInterval{
    my $aCollection = shift;    # array ref
    my $start       = shift;
    my $end         = shift;
    my $keyGen      = shift;
    my %d;
    my $index;
    for ($index = $start ; $index <= $end ; $index++) {
        my $element = $aCollection->[$index];
        my $key = &$keyGen( $element, @_ );
        if ( exists( $d{$key} ) ) {
            unshift ( @{ $d{$key} }, $index );
        }
        else {
            $d{$key} = [$index];
        }
    }
    return wantarray ? %d : \%d;
}
```

```
my $bMatches;
unshift @{ $bMatches->{$b->[$_]} },$_ for $bmin..$bmax;
```

Inlining

- Alt: 4 subs
- Neu: 1 sub

Vorteil:

- Weniger Zeilen
- Weniger Parameterübergaben
- Weniger Kontext erzeugen und aufräumen

Komplexität

- Alt: 37 Punkte McCabe, 162 LoC
- Neu: 32 Punkte McCabe, 55 LoC

Benchmark

```
             Rate LCSidx LCSnew  LCSXS S::Sim
LCSidx    25025/s     --   -39%   -55%   -98%
LCSnew    41152/s    64%     --   -25%   -96%
LCSXS     55188/s   121%    34%     --   -95%
S::Sim  1086957/s  4243%  2541%  1870%     —


LCSidx  Algorithm::Diff::LCSidx()
LCSnew  LCS::Tiny::LCS()
LCSXS   Algorithm::Diff::XS::LCSidx()
S::Sim  String::Similarity::similarity()
```

Minimal Coding in Perl

Autor: Helmut Wollmersdorfer
Deutscher Perl-Workshop 2015

Optimaler Code?

Optimal ist minimal oder maximal:

- minimale Wartung
- DRY (Don't repeat yourself) → min. Code
- minimale Komplexität – ein Aspekt
- minimale Durchführung – 1 x (Laufzeit)
- minimale Abhängigkeiten
- minimale Features (nur das Notwendige)

Ja, klar ...

Die Technik entwickelt sich immer mehr vom **Primitiven** über das **Komplizierte** zum **Einfachen**.

(Antoine de Saint-Exupéry, französischer Flieger und Schriftsteller [1900 - 1944])

DRY

```
sub from_tokens {
  my ($self, $tokens1, $tokens2) = @_;

  my %unique1;
  @unique1{@$tokens1} = ();
  my %unique2;
  @unique2{@$tokens2} = ();
  return $self->from_sets(\%unique1,\%unique2);
}

sub from_tokens {
  my ($self, $tokens1, $tokens2) = @_;

  return $self->from_sets(
    [$self->uniq($tokens1)],
    [$self->uniq($tokens2)],
  );
}
```

Ein Aspekt

```
sub from_sets {
  my ($self, $set1, $set2) = @_;

  # ( A intersect B ) / min(A,B)
  return (
    $self->intersection($set1,$set2)
      / $self->min($set1,$set2)
  );
}
```

Minimale Durchführung

- Verarbeitungsschritt wird nicht wiederholt
- widerspricht manchmal DRY
- widerspricht manchmal „Ein Aspekt“
- im Zweifelsfall gegen Performance und für Wartbarkeit

Minimale Abhängigkeit

```
sub min {($_[0] < $_[1]) ? $_[0] : $_[1]}
```

```
use List::Util qw(min);
```

Nur Notwendiges

```
package Set::Similarity::Cosine;

use parent 'Set::Similarity';

sub from_sets {
  my ($self, $set1, $set2) = @_;

  # it is so simple because the vectors
  # contain only 0 and 1
  return (
    $self->intersection($set1,$set2)
       / (sqrt(scalar @$set1)
         * sqrt(scalar @$set2))
  );
}
```

Wann Minimal Coding?

- stabiler, wiederverwendbarer Code
- CPAN Module
- langlebig
- hohe Änderungsrate

Coole Jobs bei genua

Für gehobene Sicherheit in ITK-Netzen und -Systemen sind wir die Top-Adresse. Wir suchen Sie als (w/m):

Software-Entwickler
Netzwerk-Spezialisten
IT-Security Consultant
IT-Systemspezialisten

Wir bieten:

- abwechslungsreiche Aufgaben in einem wachstumsstarken Unternehmen
- regelmäßige Fortbildung & Entwicklungsmöglichkeiten
- flexible Arbeitszeitmodelle & vielseitige Möglichkeiten zur Work-Life-Balance
- positive Unternehmenskultur durch offene Kommunikation und Wertschätzung
- kurze Entscheidungswege aufgrund flacher Hierarchien

Weitere Informationen unter
www.genua.de/career

genua