Ausgabe 1/2009 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

CGI::IDS
Website Intrusion Detection mit PerlIDS

Videomixer in Perl
Eigene Filter für Videos

F*EX
Dateitransfer für Große Jungs

Nr 09

Sichern Sie Ihren nächsten Schritt in die Zukunft

Astaro macht Netzwerksicherheit einfach. Als Marktführer in Deutschland für UTM (Unified Threat Management) bietet Astaro die funktionsreichste All-In-One-Appliance für den Schutz von Netzwerken. Auf Sie warten ein dynamisches Team und eine internationale Arbeitsumgebung mit einem Partnernetzwerk und Niederlassungen weltweit. Arbeiten Sie in den Zukunftsmärkten IT-Sicherheit und OpenSource. Packen Sie mit an!

Astaro wächst weiter, zur Verstärkung unserer Produktentwicklung in Karlsruhe suchen wir:

Perl Backend Developer (m/w)

Ihre Aufgaben sind:

- Aktive Mitarbeit an der Produktentwicklung
- Entwicklung und Pflege technisch anspruchsvoller Software-Systeme im Backend-Bereich
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Diese Stärken sollten Sie mitbringen:

- Fundierte Kenntnisse der Programmiersprache Perl
- Kenntnisse in anderen Programmier- oder Scriptsprachen wie C oder Bash wären von Vorteil
- Sehr gute Englischkenntnisse
- Selbstständiges Planen, Arbeiten und Reporten

Perl/C Software Developer (m/w)

Ihre Aufgaben sind:

- Aktive Mitarbeit an der Produktentwicklung
- Systemnahe Entwicklung technisch anspruchsvoller Software unter Linux
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Diese Stärken sollten Sie mitbringen:

- Sehr gute Kenntnisse der Programmiersprachen Perl und C
- Kenntnisse des Linux Kernels und insbesondere des Network Stacks wären von Vorteil
- Sehr gute Englischkenntnisse
- Selbstständiges Planen, Arbeiten und Reporten

Bei uns erwarten Sie ein leistungsorientiertes Gehalt, ein engagiertes Team sowie ein freundliches, modernes Arbeitsumfeld mit flexiblen Arbeitszeiten und allem, was Sie bei Ihrer täglichen Arbeit unterstützt. Begeisterte Mitarbeiter liegen uns am Herzen: Neben einer spannenden Herausforderung am attraktiven IT-Standort Karlsruhe bieten wir Ihnen das besondere Flair im sonnigen Baden und subventionierte Sportangebote am Arbeitsplatz.

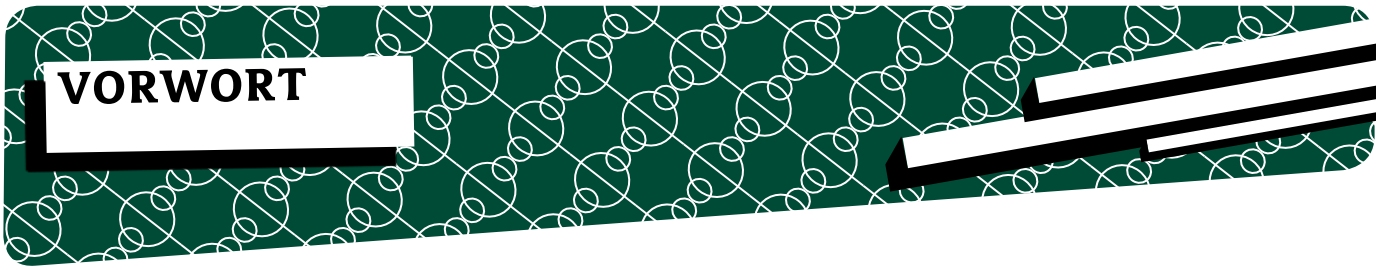
Interessiert? Dann schicken Sie bitte Ihre vollständigen Unterlagen mit Angabe Ihrer Gehaltsvorstellung an careers@astaro.com. Detaillierte Informationen zu den hier beschriebenen und weiteren Positionen finden Sie unter www.astaro.de. Wir freuen uns darauf, Sie kennen zu lernen!

Astaro AG • Monika Heidrich
Amalienbadstr. 36/Bau 33a
D-76227 Karlsruhe
Tel.: 0721 25516 0

www.astaro.de

e-mail | web | net | security

astaro internet security



Perl 5.8.x, Perl 5.10.x, Perl 5.12.x

Der Dezember scheint ein guter Monat für Perl zu sein. An einem 18. Dezember hat Perl das Licht der Welt erblickt - Alles Gute nachträglich!

Kurz vor dem 21. Geburtstag von hat Nick Clark Perl 5.8.9 veröffentlicht. Zwar etwas später als ursprünglich geplant, aber in den Mails der Perl 5 Porters lässt sich nachlesen, dass das ein hartes Stück Arbeit war.

Perl 5.8.9 wird aber das letzte große Release aus der 5.8.x-Reihe sein. Ab jetzt werden "nur" noch Bugfixes für Sicherheitsprobleme und für Plattformspezifische Bugs eingearbeitet. Jetzt wird natürlich Perl 5.10.x immer wichtiger.

Auf vielen Konferenzen hört man, dass Perl 5.10.x das beste Perl überhaupt sei. Die vielen neuen Features und die große Menge an Bugfixes werden also freudig angenommen. Viele hat jedoch die Bezeichnung "testing release" verwirrt. Gut, dass Dave Mitchell 5.10.1 vorbereitet und wahrscheinlich auch bald veröffentlich. Damit ist das "testing" durch ein "stable" ersetzt worden und nichts spricht mehr gegen Perl 5.10.x.

Perl 5.10.0 wurde übrigens auch in einem Dezember veröffentlicht - genau am 20. Geburtstag von Perl im Dezember 2007.

Die Perl 5 Porters ruhen sich aber nicht auf Perl 5.10.x aus, sondern arbeiten weiter an wesentlichen Verbesserungen von Perl. So wird jetzt schon diskutiert, welche neuen Features in Perl 5.12.x einfließen werden. Bis Perl 5.12.x veröffentlicht wird, wird es noch eine Zeitlang dauern, aber vielleicht wird auch das in einem Dezember sein.

Die Codebeispiele können mit dem Code

i23Z2

von der Webseite www.foo-magazin.de heruntergeladen werden!

\# Renée Bäcker

Ab dieser Ausgabe werden alle weiterführenden Links auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue9.

IMPRESSUM

Herausgeber:	Smart Websolutions Windolph und Bäcker GbR Maria-Montessori-Str. 13 D-64584 Biebesheim
Redaktion:	Renée Bäcker, Katrin Blechschmidt, André Windolph
Anzeigen:	Katrin Blechschmidt
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

ALLGEMEINES

6 Über die Autoren

MODULE

8 110% DBIx::Class

10 Website Intrusion Detection mit PerlIDS

PERL

15 autodie

18 Backendmodule

22 Typeglobs III

28 XS - Perl mit C erweitern

38 Perl 6 Tutorial - Teil 6

ANWENDUNG

43 Foswiki

45 F*EX

49 Videomixer in Perl

TIPPS & TRICKS

53 Perl's try-catch

NEWS

54 CPAN News

56 Neues von TPF

60 Microsoft unterstützt Strawberry Perl

61 Termine

62 LINKS

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Hinnerk Altenburg

Hinnerk hat bereits 2000 seine ersten Webformular-Skripte in Perl geschrieben, ist dann aber als Freelancer schnell auf PHP umgestiegen. Nach Informatikstudium und erstem festen Job ist er nun durch seine neue Arbeit als XING-Entwickler bei der epublica GmbH in Hamburg wieder 'dabei'. Sein erstes Perl-Modul ist seit November 2008 als CGI::IDS im CPAN verfügbar, welches er in diesem Heft vorstellt. epublica ist für die Kernentwicklung von XING verantwortlich und sucht laufend Perl-Entwickler in Festanstellung.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv - als Moderator bei Perl-Community.de, Organisator des kleinen Frankfurt Perl-Community Workshop und Mitglied im Orga-Team des deutschen Perl-Workshops.

Ferry Bolhár-Nordenkampf

Ferry kennt Perl seit 1994, als sich sein Dienstgeber, der Wiener Magistrat, näher mit Internet-Technologien auseinanderzusetzen begann und er in die Tiefen der CGI-Programmierung eintauchte. Seither verwendet er - neben clientseitigem Javascript - Perl für so ziemlich alles, was es zu programmieren gibt; auf C weicht er nur mehr aus, wenn es gar nicht anders geht - und dann häufig auch nur, um XS-Module für Perl schreiben. Wenn er nicht gerade in Perl-Sourcen herumstöbert, schwingt er gerne das Tanzbein oder den Tennisschläger.

Max Maischein

Max Maischein ist Baujahr 1973 und studierter Mathematiker. Seit 2001 ist er für die DZ BANK in Frankfurt tätig und betreut dort den Fachbereich Operations und Services im Prozess- und Informationsmanagement.

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Marcus Holland-Moritz

Geboren 1977 in Thüringen und seit Ende der 80er Jahre dem Programmieren verfallen (damals noch auf einem C16). Die Jahrtausendwende hat ihn in den Raum Stuttgart verschlagen, wo er seitdem Software für Patientenmonitore in C und C++ entwickelt. Perl hat er dabei eher zufällig (und zuerst widerwillig) entdeckt; mittlerweile schreibt er jedoch die meisten Tools, die er zum Arbeiten braucht, in Perl. Er ist Autor mehrerer CPAN-Module und auch bei den perl5-porters aktiv.

Ulli Horlacher

Ulli "Framstag" Horlacher (43) arbeitet als UNIX-Admin und -Programmierer am Rechenzentrum der Universität Stuttgart. Seine Lieblingssprache entdeckte er vor 20 Jahren mit Perl 3 unter VMS. Internet, Serverbetriebssysteme und Serverdienste sind seine Arbeitsschwerpunkte. Gelegentlich hält er auch (Perl-)Kurse an der Universität Stuttgart. Weitere Perl-UNIX-Software von ihm ist unter http://fex.rus.uni-stuttgart.de/fstools/ zu finden. Seine Freizeit verbringt er meistens mit Fahrrad- bzw. Tandemfahren und dem Bau von innovativer Illuminationshardware: http://tandem-fahren.de/Mitglieder/Framstag/LED/

Martin Seibert

Martin Seibert ist Geschäftsführer der //SEIBERT/MEDIA GmbH aus Wiesbaden. Die Multimedia-Agentur arbeitet seit 1996 mit heute knapp 60 Mitarbeitern und ist eine der professionellen und erfahrenen Web-Agenturen in Deutschland und hat vier Bereiche Consulting (Strategie, Konzepte, Usability), Design (Webdesign, Printdesign, Corporate Design), Technologies (Frontend, Backend inkl. Perl und Wiki-Anpassungen) und Systems (Webhosting, Web Security). //SEIBERT/MEDIA bietet umfangreiche Dienstleistungen rund um Foswiki (früher T Wiki) von individuellen Hosting-Angeboten über Full-Service-Implementierungen inklusive Strategie, Konzeption, Design, Implementierung, Betrieb und Schulung. Das Unternehmen setzt Foswiki (T Wiki) im eigenen Intranet und bei einigen Kunden ein.

MODULE

110% DBIx::Class

In Ausgabe 1 von $foo habe ich `DBIx::Class` schon vorgestellt. In dieser Ausgabe geht es um Möglichkeiten, die das Modul bietet, aber in den meisten Projekten nicht genutzt werden. Dabei gibt es viele Beispiele, die das Leben einfacher machen. Ein paar dieser Tricks möchte ich in dieser und den nächsten Ausgaben von $foo zeigen.

Eigene Resultset-Klassen

Wenn man Datenbanken mit `DBIx::Class` abfragt, bekommt man ein Objekt von `DBIx::Class::Resultset` zurück

```
use MySchema;

my $schema = MySchema->connect(
   "DBI:SQLite:Test" );
my $resultset = $schema->resultset(
   'Testtabelle' );

print "ja\n" if $resultset->isa(
                 'DBIx::Class::ResultSet' );
```

Diese Resultsets kann man dann noch weiter einschränken, in dem man die `search`-Methode darauf anwendet.

```
my $schema = MySchema->connect(
    "DBI:SQLite:Test"
);

my @test1 = $schema->resultset( 'User' )
                   ->search({
    username => 'Hugo',
});
my @admins = grep{
   my $t = $_;
   grep{ $_->id == 1 } $t->roles
} @test1;

print "Es gibt " . @admins .
      " Hugos, die Admin sind\n";
```

Listing 1

```
my @treffer = $resultset->search(
   key => 1 );
```

Dieses Beispiel ist sehr einfach, kann aber sehr schnell sehr komplex werden. Nehmen wir eine typische User-Verwaltung an, in der es eine User-Tabelle, eine Rollen-Tabelle und eine Tabelle, die die Verbindung herstellt, gibt. Möchte man alle Administratoren mit dem Namen 'Hugo' herausfiltern, kann man folgendes schreiben - siehe Listing 1.

Gerade wenn man diese Abfrage häufiger verwendet, ist das umständlich. Wäre es da nicht schöner, wenn man da nur eine Methode aufrufen muss? Genau das geht mit eigenen Resultsets. Dazu ändert man die User.pm etwas ab:

```
__PACKAGE__->resultset_class(
    'MySchema::ResultSet::HugoAdmins'
);
```

Damit macht man deutlich, dass es für 'User' eine eigene Klasse für die Datenbankabfrage gibt. Wie diese eigene Klasse aussieht ist in Listing 2 dargestellt.

Das ist ja quasi der gleiche Code wie oben. Und was bringt uns das jetzt? Auf den ersten Blick nicht viel, aber in Zukunft müssen wir nicht immer wieder denselben umständlichen Code schreiben, sondern können die Methode `hugo_admins` immer dann verwenden wenn wir die Administratoren mit dem Namen "Hugo" suchen.

Im weiteren Code können wir die Suche so schreiben:

```
my @hugos = $schema-> resultset( 'User' )
   ->hugo_admins;

print "Es gibt " . @hugos .
      " Hugos, die Admin sind\n";
```

Und das sieht doch wesentlich besser aus. So kann man die eigentliche Abfrage transparent gestalten und muss nicht an einigen Stellen den Code anpassen.

Diese Methoden lassen sich auch verketten. In der Userdatenbank stehen zusätzlich noch Informationen zu dem Lieblingshobby und der Lieblingsprogrammiersprache. Wenn wir alle User suchen wollen, die "tauchen" als Hobby angegeben haben, dann sieht die Abfrage so aus:

```
my $rs = $schema->resultset( 'User' )
                ->search({
   hobby => 'tauchen'
});
```

Möchte man die Perl-Programmierer finden, dann macht man das so:

```
my $rs = $schema->resultset( 'User' )
                ->search({
  language => 'Perl'
});
```

Und um die Perl-Programmierer zu finden, die gerne tauchen, dann kann man das schön kombinieren:

```
my $rs = $schema->resultset( 'User' )
                ->search({
    hobby    => 'tauchen',
    language => 'Perl',
});
```

Wenn man ein eigenes Resultset verwendet, dann kann man für die beiden ersten Abfragen eine eigene Methode schreiben (Listing 3) und für die letzte Abfrage dann die beiden einzelnen Abfragen verketten (Listing 3). So spart man sich viel Tipparbeit und bei guten Methodennamen ist gleich ersichtlich was gesucht wird.

Und wer jetzt Bedenken hat, dass da eventuell mehrere Abfragen gestartet werden, den kann ich beruhigen. In Listing 4 ist die Debug-Ausgabe für die Abfragen aus Listing 3 zu sehen.

```
package MySchema::ResultSet::HugoAdmins;

use strict;
use warnings;
use DBIx::Class::ResultSet;

our @ISA = qw(DBIx::Class::ResultSet);

sub hugo_admins {
    my ($res) = @_;

    my @res = $res->search({
       username => 'Hugo',
    });
    @res = grep{
        grep { $_->id == 1 } $_->roles
    }@res;
}

1;
```

Listing 2

`DBIx::Class` hat aus den zwei Methoden eine einzige Abfrage gemacht.

get_column

Der letzte Punkt, der in diesem Artikel betrachtet werden soll ist die Funktion `get_column`. In den meisten Fällen wird diese Funktion auf eine Ergebniszeile angewendet (siehe Listing x).

```
my $rs = $schema->resultset('User')
                ->taucher;

while( my $taucher = $rs->next ){
    print $taucher->get_column( 'id' ),"\n";
}
```

Hier wird der Wert der Spalte `id` für jeden einzelnen Taucher geholt. Man kann diese Methode aber auch schon in der Suche verwenden. So bekommt man die maximale ID eines Tauchers heraus:

```
my $max_id = $schema->resultset( 'User' )
                    ->get_column( 'id' )
                    ->max;
print $max_id, "\n";
```

Renée Bäcker

```
sub taucher{
    my ($self) = @_;

    $self->search({ hobby => 'tauchen' });
}

sub perler{
    shift->search({ language => 'Perl' });
}

print scalar $schema->
  resultset( 'User' )->taucher,      "\n",
     scalar $schema->
  resultset( 'User' )->perler,       "\n",
     scalar $schema->
  resultset( 'User' )->taucher->perler,"\n";
```

Listing 3

```
SELECT COUNT( * ) FROM User me \
   WHERE ( hobby = ? ): 'tauchen'
2
SELECT COUNT( * ) FROM User me \
   WHERE ( language = ? ): 'Perl'
1
SELECT COUNT( * ) FROM User me \
   WHERE ( ( ( language = ? ) \
   AND ( hobby = ? ) ) ): \
  'Perl', 'tauchen'
1
```

Listing 4

Website Intrusion Detection mit PerlIDS (CGI::IDS)

Aufgrund ihres hohen Aufkommens an Benutzereingaben sind besonders Web 2.0-Anwendungen anfällig für Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), SQL-Injections und ähnliche Attacken. Gerade bei der Entwicklung komplexer Websites ist es in der Praxis unmöglich, jede erdenkliche (und auch bisher noch nicht erdenkliche) Sicherheitslücke im Vorfeld applikationsseitig zu schließen. Der wohl populärste Angriff ist der MySpace-Wurm *Samy* (*http://namb.la/popular/*), der durch eingeschleusten JavaScript-Code dem Angreifer binnen weniger Stunden eine Million Friend-Requests auf MySpace zugeschanzt hat. Diesem Angriff ging ein längeres Testen der Ausführbarkeit von eingegebenen Codeteilen voraus (*http://namb.la/popular/tech.html*), bis schließlich alle Filtermechanismen von MySpace überwunden waren.

Hier greift der Ansatz der *Intrusion Detection*, um verdächtige Aktivitäten mitzuloggen und bei Bedarf schnell reagieren zu können. Das *Intrusion Detection System* (IDS) untersucht alle Requests an die Applikation auf verdächtige Strings und bewertet sie nach der Schwere - dem sogenannten *Impact* - des potenziellen Angriffs. Liegen diese Impacts über einem definierten Schwellwert, kann z.B. eine E-Mail an das Security-Team ausgesendet, oder in schweren Fällen der entsprechende User automatisch gesperrt werden.

Für PHP-Anwendungen steht hierfür das seit 2007 von Mario Heiderich, Christian Matthies und Lars H. Strojny entwickelte Intrusion Detection System PHPIDS (*http://php-ids.org*) zur Verfügung. PerlIDS ist eine Portierung der wesentlichen Teile von PHPIDS nach Perl.

Funktionsweise

Die Instrusion Detection basiert auf momentan 68 regulären Ausdrücken (*Filter*), die bekannte Angriffsmuster erkennen. Vor dem Anwenden der Filter auf die zu untersuchenden Strings, werden diese durch verschiedene Converterfunktionen umgewandelt, um auch verschleierte Angriffe erkennen zu können. Beispiele hierfür sind das Entfernen von Kommentaren, die Dekodierung von Hexadezimalwerten, JavaScript CharCode, Base64 etc. und verschiedene Formen der String-Verknüpfungen. Um viele der auftretenden *False-Alarms* (harmlose Strings, die fälschlicherweise als Attacken gewertet werden) zu vermeiden, werden z. B. Emoticons und andere oft benutzte, harmlose Zeichenkombinationen entfernt. Da die regulären Ausdrücke auf bekannten Angriffs-Vektoren basieren, können jedoch nicht alle neuen Angriffe erkannt werden. Um bisher unbekannte Vektoren abzudecken, wurde die sogenannte *Centrifuge* entwickelt, die darauf basiert, dass Angriffscode für gewöhnlich ein gewisses Aufkommen bestimmter Sonderzeichen hat.

PerlIDS führt nur ein Monitoring der Requests durch und greift nicht 'säubernd' ein. Daher verändern die Converterfunktionen lediglich eine interne Kopie der Request-Parameter, die original Parameter gehen unverändert an die Anwendung. Dieses Vorgehen vermeidet, dass hierdurch weitere Sicherheitslücken enstehen oder ein Angreifer durch direktes Feedback auf seine Angriffe einen tieferen Einblick in das Verhalten der Applikation bekommt.

Wie eingangs beschrieben, besitzt jede Filterregel einen *Impact*, der über die Schwere des Angriffsmusters informiert. Jeder getestete String erhält seinen Impact aus der Summe der einzelnen matchenden Filter-Impacts. Da Angreifer üblicherweise mit Vektoren kleinerer Impacts beginnen, die

Website auf Sicherheitslücken zu testen, können die Impacts applikationsseitig pro IP, Session oder User aufsummiert werden. So können verdächtige Aktivitäten über einen einzelnen Request hinaus erkannt werden.

Um das XML-Filterset möglichst auf dem aktuellsten Stand zu halten, ist es kompatibel zum PHPIDS-Filterset. Dieses wird laufend von internationalen Security-Experten verbessert und steht im Subversion Repository von PHPIDS (*https://svn.php-ids.org/svn/trunk/lib/IDS/default_filter.xml*) in der jeweils aktuellen Version zur Verfügung.

Die Möglichkeit, erlaubtes User-HTML zu parsen, wurde bisher nicht integriert, da PHPIDS hier auf das sehr ausgereifte und umfangreiche PHP-basierte Tool *HTML Purifier* zurückgreift. Dessen Portierung nach Perl wäre ein eigenes, erheblich aufwändigeres Projekt, wozu ich an dieser Stelle gern motivieren möchte.

Performance

Schon während der frühen Phase der Entwicklung haben wir den Einsatz im Livebetrieb auf der Perl-getriebenen Community-Website XING.com getestet. Die Analyse der gewonnenen Daten von dieser Website mit überdurchschnittlich vielen Usereingaben, ermöglichte es, die Performance von PerlIDS zu verbessern und die False-Alarm-Rate zu senken. Viele dieser Verbesserungen sind auch in den Code von PHPIDS eingeflossen.

Da die Ausführung der 68 regulären Ausdrücke sehr viel Rechenzeit beansprucht, werden grundsätzlich nur Strings getestet, welche durch den regulären Ausdruck `/[^\w\s\/@!?,]+/` fallen, d.h. in denen mindestens ein anderes Zeichen außer `a-z`, `A-Z`, `0-9`, `_`, `/`, `@`, `!`, `?`, Kommata oder Whitespaces vorkommt. Darüber hinaus wurde die Einführung regelbasierter Whitelists notwendig, da viele der auf XING.com verwendeten Request-Parameter zur Steuerung der Applikation diesen Test nicht bestehen und somit unnötigerweise getestet würden. Um keine Sicherheitslücken dadurch entstehen zu lassen, dass bestimmte Requestvariablen komplett gewhitelistet werden, können die einzelnen Parameter auf reguläre Ausdrücke getestet und in Bezug zu anderen Parametern des selben Requests gesetzt werden. Stimmt jede dieser Bedingungen, kann sich das System das teure Filterset für diese Parameterwerte ersparen. (Siehe Abschnitt *Whitelisting*.)

Als großer Performanceschub hat sich auch das Entfernen sämtlicher `/i`-Modifikatoren in den regulären Ausdrücken erwiesen. Das gleiche, wesentlich schnellere Ergebnis wird nun durch vorherige Konvertierung der Strings in lower-case erzielt.

How-To

Im CGI::IDS-Paket befindet sich das lauffähige Demo-Skript `examples/demo.pl`, mit dem man beliebige Angriffs-Vektoren testen kann und die komplette Auswertung des IDS angezeigt bekommt. Folgend ein kurzer Einstieg in den benötigten Code zum Einbau von PerlIDS in eine Webapplikation.

Instanziieren des IDS-Objekts:

```
use CGI;
use CGI::IDS;

my $cgi = new CGI;

my $ids = new CGI::IDS(
  whitelist_file  =>
        '/foo/bar/ids/param_whitelist.xml',
  scan_keys       => 0,
  # PHP-related filters
  disable_filters => [58,59,60],
);
```

Nun kann der Request auf Angriffe getestet werden. CGI::IDS ist unabhängig von CGI und testet jeden beliebigen Hashref. Im Beispiel greifen wir direkt auf die Request-Variablen des CGI-Objekts zu:

```
my %params = $cgi->Vars;
my $impact = $ids->detect_attacks(
                  request => \%params );
```

An dieser Stelle kann auf mögliche Attacken reagiert werden:

```
if ($impact > 0) {
  # Logging in Datenbank und/oder File
}
if ($impact > 35) {
  # E-Mail an Administrator
}
if ($impact > 100) {
  # SMS an Administrator
  # User ausloggen
}
```

Für das Logging in Datenbank oder Datei stehen nun folgende Variablen zur Verfügung. Jeder Aufruf von `$ids->detect_attacks()` setzt das Attack-Array neu, sodass das IDS-Objekt während seiner Lebenszeit mehrfach verwendet werden kann, ohne Filterset und Whitelist erneut aus dem Filesystem laden und parsen zu müssen (sehr nützlich bei der Verwendung von FastCGI).

```
my $attacks = $ids->get_attacks();

foreach my $attack (@$attacks) {
  # Dauer des Checks in ms
  $attack->{time_ms}
  # Parameter Key aus Request
  $attack->{key}
  # Parameter Key nach Converter,
  # falls scan_keys=1
  $attack->{key_converted}
  # Parameter Wert aus Request
  $attack->{value}
  # Parameter Wert nach Converter
  $attack->{value_converted}
  # Impact der Attacke
  $attack->{impact}

  # IDs der matchenden Filterregeln
  join(",", @{$attack->{matched_filters}})

  # Dokumentations-Strings der
  # matchenden Filterregeln
  join("\n\t",
       map {"$_: " .
            $ids->get_rule_description(
                            rule_id => $_)}
          @{$attack->{matched_filters}}
  )

  # Tags der matchenden Filterregeln
  # (z.B. SQLI, CSRF, XSS etc.)
  join(",", @{$attack->{matched_tags}})
}
```

Die einfache Auswertung nach Impacts wie hier im Beispiel ist für den Einsatz in der Praxis großer Websites noch nicht genau genug. Tatsächlich muss man weitere Mechanismen hinzufügen, welche vor einer Flut von SMS und E-Mails mit False-Alarms schützen. Hier ist ein Finetuning mit dem Website-spezifischen Aufkommen von gemeldeten Attacken erforderlich, da jede Website ihre eigene Art von Usereingaben hat. (Siehe Abschnitt *Einsatzbericht*.)

Whitelisting

Der Aufbau einer Whitelist XML-Datei ist wie folgt:

```
<whitelist>
    <param>
        <key>sender_id</key>
        <rule>
            <![CDATA[(?:[0-9]+\.[0-9a-f]+)]]>
        </rule>
        <conditions>
            <condition>
                <key>action</key>
                <rule>
                    <![CDATA[(?:^message$)]]>
                </rule>
           </condition>
        </conditions>
    </param>
</whitelist>
```

Im obenstehenden Beispiel wird der Parameter *sender_id* vom Filtering ausgenommen, falls sein Wert auf den regulären Ausdruck `/[0-9]+\.[0-9a-f]+/` matched (z. B. `3562.f3a54cd`) und der Parameter *action* des selben Requests den Wert `message` hat. Es können beliebig viele Conditions pro Key angegeben werden.

Generelles Whitelisting ohne Rules und Conditions:

```
<param>
    <key>uid</key>
</param>
```

Whitelisting nur auf einer Rule basierend:

```
<param>
    <key>scr_id</key>
    <rule>
        <![CDATA[(?:^[0-9]+\.[0-9a-f]+$)]]>
    </rule>
</param>
```

Der Parameterwert ist JSON-kodiert und soll vor dem Filtern dekodiert werden, um False-Alarms aufgrund der Kodierung zu vermeiden:

```
<param>
    <key>json_value</key>
    <encoding>json</encoding>
</param>
```

Für die Erstellung einer Whitelist führt man am besten zunächst eine Bestandsaufnahme durch, in dem man alle Parameter mitloggt. Anhand dieses Logfiles kann man dann iterativ die Whitelist aufbauen. Dazu stehen am IDS-Objekt die Arrays `$ids->{filtered_keys}` und `$ids->{non_filtered_keys}` zur Verfügung. `$key->{reason}` gibt den Grund an, weshalb der Parameter gefiltert bzw. nicht gefiltert wurde.

```
print FILTERLOG "---------\nFILTERED:\n";
foreach my $key (@{$ids->{filtered_keys}}) {
  print FILTERLOG "\t" . $key->{reason} .
                  "\t" . $key->{key} .
                  " -> " . $key->{value} .
                  "\n";
}

print FILTERLOG "NOT FILTERED:\n";
my @keys = @{$ids->{non_filtered_keys}};
foreach my $key ( @keys ) {
  print FILTERLOG "\t" . $key->{reason} .
                  "\t" . $key->{key} .
                  " -> " . $key->{value} .
                  "\n";
}
```

Einsatzbericht

Pro Woche meldet das IDS auf XING.com aus ca. 130 Millionen Requests insgesamt etwa 7.000 einzelne verdächtige Parameterwerte von nicht eingeloggten Clients. Durch zusammenfassen gleicher Werte pro Parameter lässt sich diese Zahl auf ca. 350 verringern. Hierbei handelt es sich vor allem um Message-Spambots und um Botnetze, die massenhaft versuchen, auf verbreitete PHP-Systeme zugeschnittene PHP- und SQL-Injections durchzuführen. Daher auch die hohe Rate mehrfacher identischer Angriffe.

Eingeloggte User verursachen pro Woche noch einmal etwa 8.000 weitere einzelne verdächtige Parameterwerte, nach zusammenfassen gleicher Werte pro Parameter noch rund die Hälfte. Bei echten Attacken beschränkten sich die User bisher zumeist auf einfache JavaScript-Tests wie z.B.

```
<script>alert('hallo')</script>
```

Eine Verbesserung der Filter und Converter ist nur noch in sehr geringem Maße möglich, da es sich bei den auffälligen Usereingaben oft um ASCII-Bilder, phantasievolle optische Trenner, ungewöhnliche Quotes-Kombinationen etc. handelt. Deshalb mussten anhand der gesammelten Daten Kennzahlen entwickelt werden, um Auffälligkeiten eines realen Angriffs schnell automatisiert herauszufiltern und je nach Schwere in sofortigen, täglichen und wöchentlichen E-Mail-Reports zu versenden. Zur genaueren Analyse der Angriffe steht eine Backendanwendung bereit, welche die gesammelten Daten nach verschiedenen Kriterien sortierbar und durchsuchbar macht und zu jeder Attacke die Umgebungsvariablen des Requests, die UserID, IP-Adresse etc. anzeigt.

Verfügbarkeit

Das Modul ist seit November 2008 im CPAN als CGI::IDS unter der GNU Lesser General Public License v3 verfügbar und wird möglichst auf dem aktuellen Stand der Änderungen des PHPIDS Subversion Repositorys gehalten.

Für genauere Informationen über Konfiguration und Anwendung steht die Dokumentation des Moduls zur Verfügung, bei weitergehenden Fragen bin ich unter `hinnerk@cpan.org` per E-Mail erreichbar und freue mich über Feedback!

Hinnerk Altenburg

http://yapceurope2009.org/

YAPC::EU::2009

Corporate Perl

YAPC::EU::2009, the tenth edition of the largest **European Perl Conference**.

Join us for three days of talks, workshops, and more, among hundreds of Perl programmers from all over Europe, in sunny **Lisbon, Portugal**.

Whether you're a Perl Beginner or a Perl Expert, a developer or a sysadmin, whether your interests are in Web Development, Databases, Object Oriented Perl, MVCs, Perl 6, or something else that is Perl related this conference has something for you.

Come and see what everybody else is doing.

Come and learn what everybody else is using.

Share the experiences and make your life better.

Learn. Evolve.

3 to 5 August - Lisbon, Portugal

* The YAPC::EU::2009 Logo celebrates YAPC's second year in Portugal. It includes a Perly version of the Portuguese coat of arms, replacing the five "quinas" with the five sigils of Perl ($, @, %, &, *), and the seven castles with camels. Design: Alberto Simões

autodie - Fehlerbehandlung leicht gemacht

Fehlerbehandlung, Fehlerbehandlung, Fehlerbehandlung. Immer wieder stößt man auf ein Problem, bei dem man die Fehlerbehandlung vergessen hat. Und dann kommt nur Unsinn raus. Paul Fenwick hat in der Beschreibung seines Moduls `autodie` die richtige Beschreibung:

```
     bIlujDI' yIchegh()Qo'; yIHegh()!
It is better to die() than to return()
          in failure.
              -- Klingon programming proverb.
```

Genau um dieses Modul geht es auch in diesem Artikel.

Üblicherweise macht man bei Funktionen wie `open` eine Fehlerbehandlung mit `or die ...`

```
open my $fh, '<', $filename or die $!;
```

Auf die Dauer ist es aber ganz schön mühsam, immer den `or die`-Teil zu schreiben.

`autodie` ermöglicht es, built-in Funktionen zu verwenden, ohne dass man immer ein `or die "Fehlermeldung"` tippen muss. Man muss `autodie` nur sagen, welche Funktionen automatisch eine Fehlerbehandlung bekommen sollen. Ohne import-Liste werden alle ":default" Funktionen überwacht. Paul Fenwick hat alle Funktionen in Kategorien eingeteilt, die hierarchisch aufgebaut ist. Als oberstes Element gibt es `:all`, darunter die Kategorien `:default` und `:system`. Diese teilen sich weiter auf.

Fatal.pm

Zum gleichen Zweck wurde ursprünglich das Modul `Fatal` geschrieben, das auch mit dem Perl-Core mitgeliefert wird. So kann man mit `Fatal` alle `open`-Aufrufe überwachen

```
use Fatal qw(open);

open my $fh, '<', 'keine.datei';
```

Auch wenn jetzt keine `or die`-Anweisung hinter dem `open` zu finden ist, stirbt das Skript mit einer Fehlermeldung.

```
C:\>fatal.pl
Can't open(GLOB(0x225f90), <, keine.datei):
   No such file or directory at (eval 1)
     line 3
        main::__ANON__('GLOB(0x225f90)', \
             '<', 'keine.datei')
          called at C:\fatal.pl line 5
```

Es besteht auch die Möglichkeit, eigene Subroutinen mit dem `do or die`-Ansatz zu versehen.

```
use Fatal;

sub mysub {
    die 'test' if $_[0] == 3; $_[0]
}

import Fatal 'mysub';

mysub( @ARGV );
```

`Fatal` hat aber auch einige Schwächen:

Das Modul lässt sich nicht "ausschalten". Es geht also nicht, dass ein Fehlschlagen von `open` nur in einem bestimmten Bereich zu einem Skriptabbruch führt.

Eine weitere Schwäche ist, dass es keine Fehlerobjekte gibt. Fängt man die Fehler mit `eval` ab, so landet in `$@` wirklich nur der Fehlerstring und kein Objekt (siehe Artikel Try-Catch). So wird die Auswertung relativ umständlich, da man dann mit Regulären Ausdrücken und Ähnlichem arbeiten muss.

Zusätzlich muss man auf den Rückgabewert von Funktionen achten. In dem Beispiel mit der eigenen Subroutine funkti-

oniert das Ganze nicht, wenn man nichts oder eine 0 an das Skript übergibt. Da kein "wahrer" Wert ("wahr" im Verständnis von Perl) zurückgegeben wird, geht Fatal davon aus, dass ein Fehler vorliegt.

```
C:\>fatal.pl 0
Can't mysub(0), $! is "" at (eval 1) line 3
        main::__ANON__(0) called at
           C:\fatal.pl line 7
```

Möchte man viele Funktionen überwachen, wird die Einbindung von `Fatal` mühsam, da jede einzelne Funktion in der Importliste angegeben werden muss:

```
use Fatal qw/chdir open close print .../;
```

autodie

Paul Fenwick hat mit `autodie` ein Pragma geschrieben, das dem gleichen Zweck wie `Fatal` dient und als direkter Ersatz verwendet werden kann. Fenwick hat auch gleich darauf geachtet, die Schwächen von `Fatal` zu beseitigen.

Wie bei Pragmas üblich hat `autodie` einen lexikalischen Gültigkeitsbereich. Damit kann man das Pragma in einem Block einschalten und im Nächsten ist es nicht mehr gültig. Dieser Unterschied wird deutlich, wenn man die folgenden zwei Skripte ausführt:

```
#!/usr/bin/perl

{
  use Fatal qw(open);
  open my $fh, '<', 'existierende.datei';
}

open my $fh, '<', 'keine.datei';
```

und

```
#!/usr/bin/perl

{
  use autodie qw(open);
  open my $fh, '<', 'existierende.datei';
}

open my $fh, '<', 'keine.datei';
```

Mit `Fatal` stirbt das Skript, während das Skript mit `autodie` durchläuft. Durch die lexikalische Gültigkeit hat man vielmehr die Kontrolle, in welchen Code-Teilen das Skript abbrechen soll wenn ein Fehler auftritt.

Damit die Importliste bei `autodie` nicht so lang wird, hat Paul Fenwick die Funktionen in Kategorien eingeteilt. Ein kleiner Ausschnitt der Einteilung sieht so aus:

```
:all
      :default
          :io
              read
              seek
              sysread
              sysseek
              syswrite
```

Damit wird schon deutlich, was in der Importliste für `autodie` stehen kann. Die komplette Einteilung ist in der Dokumentation des Moduls zu finden. Wenn man `use autodie qw(:all)` angibt, wird für alle relevanten built-in-Funktionen die Überwachung gestartet. Allerdings geht es noch kürzer mit `use autodie`. Sollen nur die IO-Funktionen überwacht werden, muss man `use autodie qw(:io)` schreiben. Natürlich kann man auch einfach nur eine einzelne Funktion überwachen: `use autodie qw(open)`.

Tritt ein Fehler auf, liefert `autodie` nicht einfach nur einen String zurück, sondern echte Objekte. Das hat den Vorteil, dass man nicht mit regulären Ausdrücken überprüfen muss, welche Art von Fehler geworfen wurde. Mit den Objekten kann man einfach Methoden verwenden.

```
#!/usr/bin/perl

use strict;
use warnings;
use autodie;

eval {
    open my $fh, '<', 'keine.datei';
    1;
} or do {

    if( $@->matches( 'open' ) ){
        print "open schlägt fehl: $@\n";
    }
    else{
        print "unbekannter fehler: $@\n";
    }

};
```

Wie man sieht, steht in der Spezialvariablen `$@` (siehe auch `perldoc perlvar`) nicht nur ein einfacher String, sondern man hat ein echtes Objekt, das überladen ist. Im Stringkontext wird die Fehlermeldung ausgegeben, sonst kann man damit umgehen wie mit anderen Objekten auch.

Die `matches`-Methode verlangt einen String, der dem Funktions- oder Kategorienname entspricht. So kann man dann überprüfen, ob allgemein ein IO-Fehler passiert ist: `$@->matches( ':io' )`.

Schwäche von autodie

Die Integration in Perl 5.8.x ist noch nicht ganz vollständig, da bis Perl 5.10 keine geeignete API für eigene Pragmas existierte. Fenwick arbeitet aber daran, dass auch Programmierer, die noch nicht auf Perl 5.10 umsteigen konnten, das Modul verwenden können.

Probleme gibt es mit ein paar built-in-Funktionen mit speziellen Prototypen. Da `autodie` diese Funktionen umschreiben muss, ist es schwierig, das gleiche Verhalten der Funktion zu erlangen.

So gibt es Probleme mit `system`/`exec` in einer speziellen Form: `system { $cmd } @args`. Das wird als Syntaxfehler erkannt. Wer diese Form trotzdem verwenden will, sollte `CORE::system` bzw. `CORE::exec` verwenden oder vor dem Aufruf `autodie` mit `no autodie qw(system exec)` deaktivieren.

Renée Bäcker

Backendmodule - ein Blick hinter die Kulissen

Was passiert denn da? Sind die zwei Ausdrücke äquivalent? Ein Blick hinter die Kulissen bei der Ausführung eines Perl-Programms ist nicht immer notwendig Aber häufig interessant und bei "komischen Fehlern", bei denen ein Bug in Perl vermutet wird, stellt sich mit einem Blick auf die Hintergründe meist heraus, dass es doch ein eigener Fehler ist.

Perl kommt mit einigen Modulen, die einen Blick auf die Machenschaften des Perl-Compilers erlauben. Diese Module liegen im `B::*`-Namensraum. Eines der bekanntesten dürfte `B::Deparse` sein, mit dem man sich anschauen kann, welcher Code tatsächlich ausgeführt wird. Mit `perl -MO=Deparse programm.pl` bekommt man Perl-Code zu sehen, der auf dem basiert, was perl nach dem Parsen eines Programms erzeugt. Dann bekommt man auch Optimierungen zu sehen, die der Compiler schon vornimmt.

B::Deparse

Im folgenden ist ein ganz einfaches Programm zu sehen, das mit `B::Deparse` angeschaut werden soll:

```
if(0){}

if(1){}

my $a = 3 + 4;
print $a;

if( not $a ){}

if( ! $a   ){}
```

Mit diesem Beispiel sollen gleich mehrere Sachen gezeigt werden: Dass der Perl-Compiler schon gewisse Optimierungen vornimmt und wie man sehen kann, ob zwei Ausdrücke äquivalent sind. Der Block der ersten Bedingung kann nie erreicht werden, da `if(0)` immer falsch ist. Aus diesem Grund optimiert der Compiler das weg.

```
C:\>perl -MO=Deparse bmodule.pl

'???';
do {
    ()
};
my $a = 7;
print $a;
if (not $a) {
    ();
}
if (not $a) {
    ();
}
bmodule.pl syntax OK
```

Die ersten if-Bedingung wird also in `'???';` geändert und die zweite if-Bedingung ist immer wahr. Es wird also keine Abfrage benötigt, sondern der Block wird immer ausgeführt, so kann daraus ein `do`-Block gemacht werden.

Bei den letzten beiden if-Bedingungen sieht man, dass perl diese - im Quellcode unterschiedlich geschriebenen - Ausdrücke gleich behandelt.

Als letzte Meldung bekommt man noch gesagt, ob in dem Programm Syntaxfehler vorliegen.

Es gibt noch einige Optionen, mit denen der ausgegebene Perl-Code angepasst werden kann: Sehr interessant ist die `-l`-Option. Da der von `B::Deparse` generierte Perl-Code stark vom eigenen Code abweichen kann, können mit der `-l`-Option Kommentare mit der Angabe der zugehörigen Zeilennummer im Originalcode erzwungen werden.

```
C:\>perl -MO=Deparse,-l bmodule.pl
#line 3 "bmodule.pl"
'???';
#line 4 "bmodule.pl"
print 'hallo';
bmodule.pl syntax OK
```

Mit `B::Deparse` lässt sich auch gut nachvollziehen, was in Programmen passiert, die vielleicht mit `Acme::Bleach` oder anderen Source-Filtern bearbeitet wurden. Das Programm lässt sich zwar nicht ganz genau rekonstruieren, aber man kann eine Idee von dem Ursprungsprogramm bekommen.

Der Code, der von `B::Deparse` generiert wird, ist schon optimiert. Wie oben gesehen werden fixe Berechnungen schon ausgeführt und ein ähnliches Phänomen gibt es mit Konstanten. In der aktuellen (Stand: April 2008) perltodo wird folgendes Beispiel gebracht:

```
use constant PI => 4;
warn PI;
```

wird von `B::Deparse` in

```
use constant ('PI',4);
warn 4;
```

übersetzt.

Die folgenden zwei Module geben Informationen über verwendete Operatoren und den zeitlichen Ablauf aus. Perl kennt über 300 unterschiedliche Operationen, die die Built-in-Funktionen und Operatoren und weitere - intern notwendige - Operationen wie "betreten" und "verlassen" einer Schleife implementieren.

B::Concise

Das Programm

```
#!/usr/bin/perl

my @info = split / /, 'Dies ist ein Test';
print $_,"\n" for @info;
```

mit `B::Concise` und dessen Option `-exec` aufgerufen ergibt die Ausgabe in Listing 1.

Mit der `-exec`-Option werden die einzelnen Schritte in zeitlicher Abfolge dargestellt. Schritt 1 ist immer ein "enter" - da beginnt das Programm. Im Grunde ist die Ausgabe 4-spaltig aufgebaut. In der ersten Spalte ist die "Schrittnummer" zu finden. Durch die Option `-exec` ist diese aufsteigend sortiert, aber bei der Standardeinstellung werden die Schritte nicht (unbedingt) in der zeitlichen Abfolge dargestellt.

In der zweiten Spalte ist dann etwas über die Art des Operators zu erfahren. Sind dort Zahlen (0,1,2) aufgeführt, gibt das die Anzahl der "Children" an. Unäre Operatoren z.B. das unäre + haben - wie man es wahrscheinlich schon erwartet hat - ein "Child". Einige Zeichen bei der Operatorklasse erklären sich schon fast von selbst:

Listenoperatoren sind mit einem `@` gekennzeichnet. Ein Beispiel in dem obigen Beispiel ist das `split`. Die ganze Liste der Kennzeichnung von Operatorklassen ist in der Dokumentation von `B::Concise` zu finden.

Als nächstes ist der Operator zu sehen. Einige Sachen kommen einem schon aus dem Programm bekannt vor: `split`, `print` usw. Interessant sind aber die Operatoren, die so nicht im Code zu finden sind. In der Schrittnummer 9 ist ein `padav` zu sehen. Das ist die Deklaration eines lexikalischen Arrays. Diese `pad`-Operatoren gibt es für alle Datentypen.

Diese sind intern abgekürzt: Arrays sind `AV`s, Skalare `SV`s und Hashes sind `HV`s. Die Benennung der anderen Datentypen ist ähnlich intuitiv.

Aus der Ausgabe von `B::Concise` lässt sich aber noch mehr lesen: In $foo Nr. 7 ("Sommer 2008") hat Ferry Bolhár im Typeglob-Artikel erklärt, dass es für Strings und Zahlen ver-

```
1  <0> enter
2  <;> nextstate(main 1 bmodule.pl:3) v
3  <0> pushmark s
4  </> pushre(/" "/) s/64
5  <$> const[PV "Dies ist ein Test"] s
6  <$> const[IV 0] s
7  <@> split[t2] lK
8  <0> pushmark s
9  <0> padav[@info:1,2] lRM*/LVINTRO
a  <2> aassign[t3] vKS
b  <;> nextstate(main 2 bmodule.pl:4) v
c  <;> nextstate(main 2 bmodule.pl:4) v
d  <0> pushmark sM
e  <0> padav[@info:1,2] sRM
f  <#> gv[*_] s
g  <{> enteriter(
           next->l last->o redo->h) lKS/8
m  <0> iter s
n  <|> and(other->h) vK/1
h      <0> pushmark s
i      <#> gvsv[*_] s
j      <$> const[PV "\n"] s
k      <@> print vK
l      <0> unstack v
           goto m
o  <2> leaveloop vK/2
p  <@> leave[1 ref] vKP/REFC
```

Listing 1

schiedene "Slots" gibt (neben weiteren Slots). Mit `B::Concise` lässt sich das einsehen:

```
C:\>perl -MO=Concise -e "my $var = 8.4;
                         $var = 'test';
                         $var = 3;"
[...]
3         <$> const[NV 8.4] s ->4
[...]
7         <$> const[PV "test"] s ->8
[...]
b         <$> const[IV 3] s ->c
[...]
```

Dabei steht `PV` für `PointerValue`, `IV` für `IntegerValue` und `NV` für `NumericValue`. Und dass die verschiedenen Slots auch tatsächlich unterschiedlich gefüllt sind, lässt sich mit `Devel::Peek` sehen.

```
C:\>perl -MDevel::Peek -e
    "my $var = 8.4;
        $var = 'test';
        $var = 3;
        Dump $var;
        print qq~\n\nErgebnis: ~, $var + 1"
SV = PVNV(0x23a274) at 0x236038
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,IOK,pIOK)
  IV = 3
  NV = 8.4
  PV = 0x18258ec "test"\0
  CUR = 4
  LEN = 8


Ergebnis: 4
```

Für die Addition wird der Wert aus dem Integer-Slot verwendet. Im Zusammenspiel von mehreren Modulen kann man so auf die Erklärung der verwunderlichsten Phänomene kommen.

B::Debug

Mit `B::Concise` wurde angezeigt, welche Art von Operatoren vorlagen und ein paar wenige Informationen mehr. Mit `B::Debug` (siehe Listing 2) werden viel mehr Informationen über die Operatoren angezeigt. Hier sind auch nicht mehr die Zeichen für die Operatorenklassen zu finden, sondern hier kann man schon am Namen erkennen, um welche Klasse von Operator es sich handelt.

Nach der Operatorklasse (z.B: `LISTOP`) wird noch die Speicheradresse aufgeführt. Darunter gibt es die ausführlichen Informationen zu dem Operator.

B

Das Modul `B` steckt hinter diesen Modulen und bietet eine API für alle möglichen Dinge in Perl, so dass man sich ein eigenes Backendmodul schreiben kann oder in einem "normalen" Modul an wichtige Informationen kommt. Mit Hilfe des `B`-Moduls kann man auch schnell ein Programm schreiben, dass mir alle Namen von OPs in einem "Baum" ausgibt.

```
#!/usr/bin/perl

use strict;
use warnings;

use B ();

test();

B::walkoptree( B::main_root(), 'debug' );

sub test {
    my $a = 3;
}

sub UNIVERSAL::debug {
    print $_[0]->name,"\n";
}
```

```
C:\>perl -MO=Debug bmodule.pl
LISTOP (0x1838458)
        op_next         0x0
        op_sibling      0x0
        op_ppaddr       PL_ppaddr[OP_LEAVE]
        op_targ         1
        op_type         178
        op_seq          8190
        op_flags        13
        op_private      64
        op_first        0x18380f0
        op_last         0x18381a8
        op_children     6
OP (0x18380f0)
        op_next         0x183847c
        op_sibling      0x183847c
        op_ppaddr       PL_ppaddr[OP_ENTER]
[...]
Nullsv
OP (0x1838234)
        op_next         0x1838250
        op_sibling      0x0
        op_ppaddr       PL_ppaddr[OP_UNSTACK]
        op_targ         0
        op_type         176
        op_seq          8186
        op_flags        1
        op_private      0
bmodule.pl syntax OK
```

Listing 2

Nutzen

Und wofür braucht man das jetzt? Wie schon anfangs erwähnt, kann man dieses Wissen nutzen, um ein "komisches" Verhalten von Perl zu untersuchen. Liege ich mit meiner Vermutung richtig oder habe ich da was falsch verstanden?

Mit dem Modul `B::Concise` kann man sich ganz gut anschauen, was der Compiler bei `!` beziehungsweise `not` macht. Zur Veranschaulichung wird folgender Code genommen:

```
my $link = 'htt://foo-magazin.de';
if (! $link =~ m!^https?://!i ){}
```

Aufgerufen wird es mit `perl -MO=Concise,-exec link.pl`. Ein Ausschnitt sieht so aus:

```
C:\>perl -MO=Concise,-exec link.pl
...
7  <0> padsv[$link:1,4] s
8  <1> not sK/1
9  </> match(/"^https?://"/) sKS/RTIME
...
link.pl syntax OK
```

Im Gegensatz dazu der Code, bei dem das `!` durch `not` ersetzt wurde.:

```
C:\>perl -MO=Concise,-exec link.pl
...
7  <0> padsv[$link:1,4] s
8  </> match(/"^https?://"/) sKS/RTIME
9  <1> not sK/1
...
link.pl syntax OK
```

Im ersten Beispiel erkennt man, dass der Skalar deklariert wird (padsv) als nächstes negiert wird (not) und dann erst der Match gemacht wird (match). Mit dem `not` statt dem `!` ist die Reihenfolge von `match` und `not` getauscht, was durch die Operatorrangfolge erklärbar ist (die ist in `perldoc perlop` genauer erläutert).

Die gezeigten Module sind in Perl 5.10 standardmäßig dabei und auf CPAN gibt es noch jede Menge weiterer `B::*`-Module.

Renée Bäcker

Typeglobs III

In der vorigen Ausgabe wurde gezeigt, was man mit Typeglobs alles anstellen kann. In diesem abschließenden Teil des Typeglob-Tutorials geht es um das Erstellen von Wrapperfunktionen und den Umgang mit Datei- und Filehandles.

Einrichten von Wrappern

Ein *Wrapper* ist eine Funktion, die um eine andere Funktion "gelegt" wird, um deren Arbeitsweise zu ändern oder an spezielle Gegebenheiten anzupassen. Das Besondere daran ist, dass im Idealfall weder der Code, der die ursprüngliche Funktion (im folgenden als *Original* bezeichnet) aufruft, noch das Original selber merkt, dass es von einem Wrapper umgeben worden ist. Im Fall von Perl ist diese Bedingung erfüllt, wenn sein Builtin `caller` innerhalb des Originals vor und nach dem Wrappen dasselbe Resultat zurückliefert.

Durch den Einsatz von Typeglobs läßt sich das recht einfach erreichen. Zum Testen dient die folgende Funktion:

```
sub hello {
 print 'Args:   ', join(',', @_), "\n",
       'Caller: ', join(',', caller), "\n\n";
}
```

Der Code, der den Wrapper generiert, sieht wie folgt aus (siehe Listing 1):

Mittlerweile wird dieser vormals etwas kryptisch anmutende Code hoffentlich schon einigermaßen vertraut wirken - das Wesentliche sind dabei die Zeilen 5 und 12.

`create_wrapper` wird der Name der zur wrappenden Funktion übergeben. In Zeile 5 wird durch Auslesen des Codeslots des zugehörigen Typeglobs eine Codereferenz, d.h. ein Zeiger auf diese Funktion in `$oldsub` abgespeichert (man hätte natürlich auch

```
$oldsub = \&{$name};
```

schreiben können [1]). Existiert die angegebene Funktion nicht, so ist der Slot unbelegt und man erhält `undef` zurück, worauf mit Ausgabe einer entsprechenden Meldung und Abbruch reagiert wird. Ansonsten wird nun eine neue Funktion - der Wrapper - erzeugt, der hier ausgibt, von wem er aufgerufen wurde und anschließend mit

```
goto &$oldsub;
```

zur ursprünglichen Funktion springt [2]. Die gezeigte Verwendung von `goto` bewirkt, dass der Callstack des Wrappers durch den des Aufrufers ersetzt wird; für die angesprungene Funktion sieht alles so aus, als wenn sie direkt aufgerufen worden wäre. Dieses spezielle `goto` wird auch in Zusammenhang mit `AUTOLOAD`-Funktionen häufig verwendet.

```
 1  sub create_wrapper {
 2    no strict 'refs';
 3    no warnings 'redefine';
 4    my $name = caller . '::' . shift;
 5    my $oldsub = *{$name}{CODE} or die "Can't find subroutine '$name'!\n";
 6    my $newsub = sub {
 7      my ($pkg, $file, $line) = caller;
 8      print STDERR "WRAPPER: Hi! $name(@_) was called:\n",
 9                   "WRAPPER: from '$pkg', file '$file', line $line\n\n";
10      goto &$oldsub;
11    };
12    *{$name} = $newsub;
13    return $oldsub;
14  }
```

Listing 1

Nun wird in Zeile 12 durch die Zuweisung an den Typeglob dessen Codeslot, der bislang immer noch auf das Original zeigt, mit einer Referenz auf die Wrapper-Funktion überschrieben. Das Ergebnis ist, dass nun eine neue Funktion - das "gewrappte" Original - unter demselben Namen aufgerufen werden kann [3].

Zwei Dinge sollen noch kurz erwähnt werden:

• Da auch hier wieder mit symbolischen Referenzen jongliert wird (Zeile 5 und 12), ist ein `no strict 'refs'` unerläßlich. Weiters ist Perl sehr misstrauisch, was Zuweisungen an Typeglobs zur Laufzeit betrifft, wenn dabei Codeslots überschrieben werden, und weist mit

```
Subroutine ... redefined
```

auf solche Praktiken hin, wenn `use warnings` (oder der `-w` Schalter) angegeben wurde [4]. Innerhalb von `create_wrapper` ist dieses Misstrauen allerdings ungerechtfertigt, daher wird die Ausgabe dieser Meldung mit dem entsprechenden `no warnings 'redefine'` Pragma unterbunden. In Perl-Versionen vor 5.6 hätte man (weniger spezifisch, aber genauso effektvoll)

```
local $^W = 0;
```

geschrieben.

• Die Möglichkeit des Wrappens (oder vollständigen Umdefinierens) von Funktionen zur Laufzeit ist nicht auf selbstbenannte Funktionen beschränkt. Reservierte Funktionen wie `AUTOLOAD` oder `DESTROY`, aber auch Methoden (die ja im Prinzip auch nur Funktionen mit einer anderen Aufrufkonvention sind), können genauso damit "behandelt" werden. So kann man das Verhalten von Autoload-Code, aber auch von `DESTROY` beeinflussen, und, wenn Packages als *Klassen* verwendet werden, durch Erzeugen neuer benannter Funktionen diesen Klassen *Methoden* zur Laufzeit hinzufügen oder bestehende abändern.

Nicht funktionieren wird das allerdings mit den reservierten Namen von *Blöcken* wie z.B. `BEGIN`, weil dies keine Funktionsnamen sind und vom Parser intern anders abgehandelt werden (ein Typeglob repräsentiert ja nur genau *einen* Namen, es kann aber *mehrere* `BEGIN`-Blöcke geben).

Das folgende Beispiel zeigt einen Aufruf von `create_wrapper` in der Datei `cre_wrp.pl`:

```
hello(qw[a b c]);
my $oldfunc = create_wrapper('hello');
hello(qw[d e f]);
{
  no warnings 'redefine';
  *hello = $oldfunc;
}
hello(qw[g h i]);
```

und deren Ausgabe:

```
Args:   a,b,c
Caller: main,cre_wrp.pl,1

WRAPPER: Hi! main::hello(d e f) was called:
WRAPPER: from 'main',
              file 'cre_wrp.pl', line 3

Args:   d,e,f
Caller: main,cre_wrp.pl,3

Args:   g,h,i
Caller: main,cre_wrp.pl,8
```

In der ersten Zeile wird das Original aufgerufen, das ausgibt, mit welchen Argumenten und von wem der Aufruf erfolgte. Danach wird der Wrapper erzeugt und man erhält nun beim Aufruf von `hello()` vor dessen Ausgabe noch den Output des Wrappercodes. Beachte, dass die von `caller` gelieferten Werte - von der Zeilennummer abgesehen - in beiden Fällen dieselben sind, es handelt sich also um einen "echten" Wrapper.

Der Code ab der vierten Zeile zeigt, wie man die Wirkung von `create_wrapper` wieder *rückgängig* machen kann. In der zweiten Zeile wurde der von dieser Funktion zurückgelieferte Wert - eine Codereferenz auf das Original - in einem Skalar zwischengespeichert. Innerhalb des folgenden Blocks wird nun durch Zuweisung dieses Skalars an den Typeglob wieder der alte Zustand hergestellt; beim nächsten Aufruf von `hello()` wird daher wieder das Original, ohne Wrapper ausgeführt.

Auch hier muss vorher wieder die Ausgabe der `redefined` Meldung mittels `no warnings` unterdrückt werden, und die sauberste Art, dies zu tun, ist das Verwenden eines eigenen Blocks, der die Wirkung des Pragmas auf die folgende Zuweisung beschränkt.

Eine denkbare Erweiterung des obigen Codes wäre das Zwischenspeichern auch der zweiten Codereferenz des Wrappers und dann (z.B. mittels zweier Funktionen `wrapper_on` und `wrapper_off`) durch Zuweisung der jeweiligen Referenz an den Typeglob zwischen der Version mit Wrapper und

der ohne Wrapper "hin- und herzuschalten". Die sich aus dieser Technik ergebenden Möglichkeiten sind außerordentlich vielfältig.

Umgang mit Handles

Neben den bisher erwähnten Wertetypen (*Skalar*, *Array*, *Hash*, *Code*) gibt es noch zwei weitere: **Datei- und Formathandles**. Im Gegensatz zu den anderen Typen verfügen sie über keine besondere Kennzeichnung im Namen (kein *Sigil*), sondern werden direkt (als *Bareword*) angegeben. Das bedeutet, dass sie nur an solchen Stellen vorkommen dürfen, an denen sie der Parser per Definition erwartet. Das macht einen flexiblen Einsatz in Programmen schwierig.

Während dies bei Formathandles weniger problematisch ist, da diese ein eher bescheidenes Dasein fristen [5], wird es bei Dateihandles oft als lästig empfunden, weil dadurch z.B. keine Übergabe von und an Funktionen und keine Lokalisierung möglich ist. Weiß man aber, dass man an Stellen, an denen ein Dateihandle erwartet wird, auch einen Typeglob verwenden kann (über dessen `IO` Slot dann der Handle angesprochen wird), dann sind die durch Barewords vorgegebenen Einschränkungen kein Problem mehr [6].

Möchte man den Handle einer geöffneten Datei an eine Funktion zur Verarbeitung übergeben, so kann man

```
open FH, 'input.dat';
process(*FH);

sub process {
  my $fh = shift;
  while (<$fh>) {
    ...
  }
}
```

schreiben. An `process` wird einfach der gleichnamige Typeglob übergeben. Dieser wird innerhalb der Funktion in einen Skalar kopiert (Fake-Kopie!) und danach über die Kopie der darin enthaltene IO Slot angesprochen. Eine weniger schöne Alternative wäre:

```
sub process {
  *SUBFH = shift;
  while (<SUBFH>) {...
```

Hier wird der Typeglob in einen zweiten kopiert und in Befehlen, die die Angabe eines Handles durch ein Bareword vorsehen, dieses verwendet. Diese Alternative ist deswegen weniger schön, weil der innerhalb der Funktion angelegte Typeglob ebenfalls global ist (wodurch man sich die Übergabe als Funktionsargument sparen und gleich `*FH` hätte verwenden können) und damit die Gefahr besteht, inner- und außerhalb der Funktion unabsichtlich denselben Handle bzw. Typeglob zu verwenden. Außerdem ist damit ein rekursives Verwenden der Funktion nicht mehr möglich (sie ist *non-reentrant*).

Abzuraten ist auch von folgendem Code:

```
$name = 'FH';
# Dasselbe wie: open FH, '>output.dat';
open $name, '>output.dat';
# Dasselbe wie: print FH 'Hallo, Welt!;
print $name 'Hallo, Welt!';
```

Hinter diesem Code steckt nichts weiter als eine symbolische Referenz [7]; der Skalar, der als Handle-Argument in `open` verwendet wird, enthält hier einen String und daher wird dieser als Handlename (d.h., Name des Typeglobs) verwendet. Die Verwendung von symbolischen Referenzen in Verbindung mit IO Handles ist auch deswegen gefährlich, weil sie leicht zu falschen Annahmen führen kann:

```
sub process {
  my $fh = 'FH';
  open $fh, '>temp.tmp.';
  print $fh 'Hallo, Welt!';
}
```

Hier wird innerhalb der Funktion eine lexikalische Variable, die als Filehandle verwendet wird, deklariert. Es liegt die Vermutung nahe, dass nach Beendigung der Funktion durch das Verschwinden der lexikalischen Variable infolge des Blockendes auch die Datei geschlossen wird. Diese Vermutung ist jedoch falsch. Da es sich nur um eine symbolische Referenz handelt, besteht zwischen dem lexikalischen Skalar und dem über ihn angesprochenen Handle kein Zusammenhang; der globale Typeglob (hier also `*FH`) und der darin enthaltene IO Handle bleiben nach dem Ende der Funktion unverändert bestehen und somit die betreffende Datei auch weiterhin geöffnet. Die intuitiv erwartete "Bereinigung" am Blockende findet hier also nicht im erwarteten Ausmaß statt und daher sollte man solchen Code - auch im Interesse derer, die ihn später warten müssen - tunlichst vermeiden.

Eine Möglichkeit, das automatische Schließen einer Datei bei Beenden einer Funktion einzurichten, besteht im *Lokalisieren des Typeglobs*, der als Handle verwendet wird:

```
sub process {
  local *FH;
  open FH, '>temp.tmp';
  ...
}  # temp.tmp wird jetzt geschlossen
```

Durch das Lokalisieren des Typeglobs werden *alle* Slots temporär abgespeichert und durch neue, unbenutzte Slots ersetzt. War also bereits eine Datei geöffnet, so bleibt sie geöffnet, kann aber innerhalb der Funktion nicht mehr angesprochen werden. Stattdessen wird der neu angelegte IO Slot zum Öffnen der temporären Datei verwendet. Beim Erreichen des Funktionsendes muss Perl wieder alle abgespeicherten Slots zurückholen. Dabei erkennt es, dass der neu angelegte IO Slot belegt ist und dass die betreffende Datei daher erst geschlossen werden muss, bevor ein Überschreiben mit dem alten Slotwert möglich ist..

Mit Perl 5.6 wurde *Auto-Vivification* mit IO Handles eingeführt. Diese Fähigkeit, die man am besten mit "zum Leben erwecken" übersetzen kann, gibt es auch in anderen Fällen, z.B. wird mit

```
$alle->[4]->{gamma}->[76] = 'Hallo';
```

das 77. Element eines anonymen Arrays belegt, auf das das Element `gamma` eines anonymen Hashes zeigt, welches wiederum über das fünfte Element eines weiteren anonymen Arrays angesprochen wird, auf das schließlich die Referenz in `$alle` zeigt. Mit Auto-Vivification ist nun hier gemeint, dass alle diese Elemente und Referenzen beim Ausführen dieses einen Befehls automatisch angelegt werden, wenn sie noch nicht existieren.

Das Gleiche ist ab 5.6 bei

```
open $fh, '>temp.tmp';
```

der Fall, wenn der Skalar `$fh` zum Zeitpunkt der Ausführung noch auf `undef` gesetzt ist. Da der Skalar noch nichts (d.h., auch keinen String, der als symbolische Referenz verwendet werden könnte) enthält, wird automatisch ein Typeglob und in dessen IO Slot der Handle "zum Leben erweckt" und in den Skalar eine Referenz darauf geschrieben. Handle und Typeglob sind hier aber, wie auch die Arrays und Hashes im Beispiel weiter oben, *anonym*, d.h., man kann den IO Handle nur über den Skalar, der nun eine *echte* Referenz enthält [8] ansprechen. Und damit wird auch klar, dass man mit

```
{
  open my $fh, '>temp.tmp';
  print $fh 'Hallo, Welt!';
  ...
}
```

nun auch erreichen kann, dass eine Datei bei Verlassen eines Blocks (oder einer Funktion) automatisch geschlossen wird: da der lexikalische Skalar der einzige Bezugspunkt zum anonymen Typeglob ist, kann dieser, nachdem der Skalar verschwunden ist, ebenfalls entfernt werden und damit wird vorher implizit die Datei, die über seinen IO Slot geöffnet war, geschlossen.

Auto-Vivification wird natürlich nicht nur von `open`, sondern auch von alle anderen Builtins, die Handles anlegen (z.B. `opendir`, `pipe`, `sysopen`, `socket` und `accept`) unterstützt. Tatsächlich zeigt das obige Beispiel den seit Perl 5.6 empfohlenen Weg, Dateihandles zu verwenden.

Aber auch Benutzer älterer Perl-Versionen brauchen auf diese Funktionalität nicht zu verzichten: es gibt zwar noch keine Auto-Vivification, aber die Verwendung lexikalischer Handles ist genauso möglich. Man muss Perl nur etwas unter die Arme greifen:

```
# Erzeuge Globreferenz
my $fh = \*FH;
# Lösche Stashelement (Typeglob wird anonym)
delete $main::{FH};
# Weiter wie gehabt
open $fh, '>x.x';
```

Hier wird zunächst eine Referenz auf einen benannten Typeglob angelegt. Anschließend wird der Eintrag des Globs aus der Symboltabelle (Stash) wieder gelöscht [9], der Typeglob wird dadurch anonym. Nun kann der Skalar genauso wie beim Auto-Vivifying als Handle verwendet werden; auch hier wird die Datei automatisch geschlossen, wenn der Bereich verlassen wird, innerhalb dessen `$fh` sichtbar ist.

In diesem Zusammenhang soll auch noch die Funktion

```
Symbol::geniosym();
```

erwähnt werden, deren Code vereinfacht als

```
sub geniosym {
  my $name = 'GEN' . $genseq++;
  my $sym = \*{'Symbol::' . $name};
  delete $Symbol::{$name};
  select(select $sym);
  return *{$sym}{IO};
}
```

dargestellt werden kann. Die ersten drei Zeilen erzeugen, wie weiter oben beschrieben, einen anonymen Typeglob. Die vorletzte Zeile erzwingt durch die Verwendung von `select` die Initialisierung des IO Slots des erzeugten Typeglobs (dadurch wird vermieden, dass die Funktion eine Referenz auf `undef` zurückliefert), und die letzte Zeile gibt den Inhalt des IO Slots (eine Referenz auf das Dateihandle) zurück. Dadurch kann das Handle direkt über die Referenz und somit schneller als über einen Typeglob angesprochen werden, allerdings ist die Erzeugung, wie man sieht, umständlicher. Der von `geniosym` zurückgelieferte Wert ist somit keine Glob-, sondern eine **IO (Handle) Referenz** [10].

Tatsächlich kann man an `open` (und die anderen Funktionen zum Anlegen von Filehandles) folgendes direkt oder in einem Skalar übergeben:

```
# Auto-Vivifying
$fh =  undef;     open $fh, 'x.x';
# Globreferenz
$fh = \*GLOB;     open $fh, 'x.x';
# Typeglob
$fh =  *GLOB;     open $fh, 'x.x';
# IO (Handle) Referenz
$fh =  *GLOB{IO}; open $fh, 'x.x';
```

wobei Methoden 1 und 4 erst ab Perl 5.6 unterstützt sind [11]. Das mit Methode 1 durchgeführte Auto-Vivifying entspricht, wie weiter oben gezeigt, der Globreferenz von Methode 2.

Das Arbeiten mit lexikalischen Globreferenzen ist die einfachste und sicherste Möglichkeit, mit Handles umzugehen. Bis Perl 5.6 wurden stattdessen Typeglobs direkt verwendet und waren zumindest in diesem Bereich absolut unentbehrlich. Da man immer wieder auf Code, der glob-basierende Handles verwendet, stoßen kann, macht es Sinn, auch heute noch diesen Einsatzbereich von Typeglobs zu kennen.

Ferry Bolhár-Nordenkampf

[1] Obwohl die beiden Schreibweisen

```
*glob =  *func{CODE};
*glob = \&func;
```

letztlich dasselbe bewirken, werden unterschiedliche Tätigkeiten durchgeführt (und daher ist auch der kompilierte Code unterschiedlich): Im ersten Fall wird der Slot eines Typeglobs ausgelesen und die darin enthaltene Referenz wird zugewiesen. Im zweiten Fall hingegen wird der Funktionswert als solcher angesprochen und erst eine Referenz darauf erzeugt, bevor die Zuweisung durchgeführt wird. Anders gesagt liefert das zweite Konstrukt zur Laufzeit genau denselben Wert, der vom Parser beim Kompilieren einer `sub`-Deklaration im Slot des entsprechenden Typeglobs abgelegt würde.

Meistens wird die zweite Schreibweise verwendet, obwohl eigentlich die erste transparenter macht, was hier tatsächlich zugewiesen wird. Außerdem ist erste, da lediglich ein bereits vorhandener Wert kopiert wird, effizienter.

[2] Tatsächlich ist diese Schreibweise unvollständig - genau genommen, bedeutet

```
goto &$oldsub;
```

"Rufe die Funktion, deren Name (oder Referenz) in `$oldsub` steht, auf und springe anschließend zu jener Stelle, auf die die von ihr zurückgelieferte Codereferenz zeigt". Da das aber in den seltensten Fällen wirklich gewünscht ist, zeigt der Parser einmal mehr Initiative, indem er daraus ungefragt ein

```
goto \&$oldsub;
```

macht, was tatsächlich dem entspricht, was erwartet wird: "Springe zur Codereferenz, deren Name in der Variable `$oldsub` steht". Und da `$oldsub` in diesem Beispiel selbst eine Codereferenz enthält (Zeile 5 im Beispielcode), kann man kürzer auch

```
goto $oldsub;
```

schreiben. Das funktioniert aber nur, wenn die Variable eine Codereferenz enthält; andernfalls wird der Inhalt als Name einer Sprungmarke (Label) interpretiert.

[3] Tatsächlich erzeugt `create_wrapper` eine *Closure*, d.h., eine Funktion, die eine lexikalische Variable (`$oldsub`), die außerhalb ihres Geltungsbereiches liegt, anspricht. Dadurch wird der Wert der Variablen, den diese zum Zeitpunkt der Definition der Closure hatte, darin "eingeschlossen" (Name!). Es können beliebig viele derartige Closures gleichzeitig erzeugt werden, jede mit ihrem eigenen Wert für `$oldsub`.

Während Closures meist nur als anonyme Subroutinen angelegt werden, erhalten sie hier - da sie ja bestehende, benannte Funktionen ersetzen sollen - durch Zuweisung ihrer Codereferenzen an Typeglobs ebenfalls Namen.

[4] Und der wird doch immer angegeben, oder etwa nicht? ;-)

[5] Zu unrecht, wie ich meine. Das in Perl eingebaute, automatische Formatierungssystem, das mit den Variablen `$.`, `$%`, `$=`, `$-`, `$~`, `$^`, `$:` und `$^L` und den Befehlen `format` (Deklaration) und `write` (Ausführung) gesteuert wird, ist sehr leistungsfähig und Perl verdankt das "r" in seinem Namen (*Practical Extraction & **Report** Language*) nicht zuletzt diesem System. Ein Blick in `perlform` lohnt sich allemal. Formate müssen allerdings - wie Packages, Subroutinen und lexikalische Variable - vor der ersten Verwendung *deklariert* werden, was aber auch in einem String-`eval` und damit erst zur Laufzeit geschehen kann.

[6] Auf die sich durch das *Auto-Vivifying* ab Perl 5.6 ergebenden Möglichkeiten wird weiter unten eingegangen; es werden zunächst die Wege, die bereits in älteren Perl-Versionen vorhanden waren, erläutert.

[7] Was man bei der Verwendung von `strict 'refs'` auch schnell merkt.

[8] Man kann sich davon mit

```
print ref $fh;
# Gibt 'GLOB' (d.h., Globreferenz) aus
```

überzeugen.

[9] Hier unter der Annahme, dass der Code im Package `main` ausgeführt wird.

[10] Allerdings wird man über die Ausgabe von

```
use Symbol 'geniosym';
my $sym = geniosym();
print ref $sym;     # Gibt 'IO::Handle' aus
```

etwas erstaunt sein - ein IO Handle (d.h., die IO Datenstruktur, auf die die Referenz in `$sym` zeigt), ist aus implementierungstechnischen Gründen stets ein *Objekt* des Packages `IO::Handle` (d.h., eine Referenz auf einen Wert, der mit dieser speziellen Klasse "gesegnet" wurde). Dieser Mechanismus ermöglicht das Verwenden von IO Handles als Objektinstanzen. So sind folgende Befehle (weitgehend) identisch:

```
print $sym 'Hallo';
$sym->print('Hallo');
```

oder, da hier auch wieder Typeglobs als Stellvertreter verwendet werden können:

```
print FH 'Hallo';
# oder: print {*FH} 'Hallo';

FH->print('Hallo');
# oder: *FH->print('Hallo');
```

Auch wenn es nicht so aussieht, ist `FH` hier kein Klassenname, sondern eine Objektinstanz - genauer gesagt wird damit das durch den IO Slot des Typeglobs `*FH` repräsentierte Objekt und darüber die Methode `print` angesprochen. Ganz schön gefinkelt, nicht wahr?

Der objekt-orientierte Ansatz hat den Vorteil der größeren Flexibilität: durch Vererbung können Methoden hinzugefügt werden und damit kann der Funktionsumfang eines IO Handles weit über die von Perl's Builtin-Funktionen bereitgestellten Möglichkeiten hinausreichen. Außerdem wird dadurch die Benutzung von Funktionen oder Eigenschaften, die sich auf das *gerade aktuelle Handle* beziehen, erleichtert - "das gerade aktuelle" Handle ist dann die Objektinstanz, die beim Aufruf der jeweiligen Methode angegeben wird. Das macht Code, der mit solchen Methoden arbeitet, transparenter.

[11] Man kann in Perl-Versionen vor 5.6 auch eine noch nicht verwendete Handlereferenz an `open` und Konsorten übergeben, man muss sie nur vorher über den Typeglob *initialisieren*:

```
select (select *GLOB);   # Initialisierung
my $fh = *GLOB{IO};      # IO-Referenz holen
open $fh,'x.x';       # An open() uebergeben
```

Der `select` Befehl legt eine Handlestruktur an und läßt den IO Slot des angegebenen Typeglobs darauf zeigen. Danach kann dieser in den Skalar übernommen (oder auch direkt mit `open()` angegeben) werden.

Nicht, dass man das umbedingt brauchen würde, es sollte nur der Vollständigkeit halber gezeigt werden. :-)

Dieser Umstand (ein IO Slot musste bis Perl 5.6 vor der ersten Benutzung initialisiert werden), ist auch der Grund dafür, dass dies, wie weiter oben gezeigt, in `Symbol::geniosym` (das natürlich auch noch ältere Perl-Versionen unterstützt) immer noch geschieht. In perldata (Abschnitt *Typeglobs und Filehandles*) wird sogar extra darauf hingewiesen:

> *That's because *HANDLE{IO} only works if HANDLE has already been used as a handle.*

Ab Perl 5.6 stimmt das nicht mehr.

XS - Perl mit C erweitern

Teil 2: Fortgeschrittene XS-Programmierung 1

In den folgenden Abschnitten wollen wir ein wenig nicht-trivialen XS-Code unter die Lupe nehmen. Dazu brauchen wir wieder ein leeres XS-Modul, in das wir die Beispiele einfügen können.

```
$ h2xs -An AdvancedXS
```

Minimum und Maximum einer Liste

Als erstes wollen wir zwei Funktionen schreiben, um den kleinsten bzw. größten Wert einer Liste zu bestimmen. Selbstverständlich gibt es so etwas schon (`min` und `max` im Modul `List::Util`), aber wir werden das Rad trotzdem neu erfinden.

Eine schöne Eigenschaft der `min`- und `max`-Funktionen ist, dass der Algorithmus in beiden Fällen fast identisch ist. Eine ebenso schöne Eigenschaft von XS ist, dass man in einem solchen Fall Dank des `ALIAS`-Schlüsselwortes nur eine Funktion implementieren muss (siehe Listing 1).

Aber beginnen wir vorne im Code. Als erste Neuerung ist da das `PROTOTYPE`-Schlüsselwort, mit dessen Hilfe man, wie in Perl auch, einen Prototyp für eine Funktion definieren kann. Als zweite Neuerung gibt es das `...` anstelle der Funktionsparameter. Wie aus dem Prototyp schon ersichtlich ist, sollen `min` und `max` mit einer Liste von Werten aufgerufen werden. Und genau das ist auch die Bedeutung von `....` Es ist die C-Syntax für eine variable Argumentliste. Wir werden später noch ein weiteres Beispiel dafür sehen.

Als nächstes haben wir das gerade schon erwähnte `ALIAS`-Schlüsselwort. Damit lassen sich für dieselbe Implementierung einer XSUB mehrere Namen vergeben. Jedem Namen (in unserem Fall `min` und `max`) kann eine beliebige Konstante zugewiesen werden. Je nachdem unter welchem Namen die XSUB aufgerufen wird, steht in der Variablen `ix` dann genau diese Konstante. In unserem Beispiel ist `ix` bei einem Aufruf von `max` auf `1` gesetzt. Die explizite Angabe des Alias `min = 0` ist übrigens nicht notwendig, schadet aus Dokumentationssicht aber auch nicht.

Der Abschnitt, der durch `PREINIT` eingeleitet wird, dient zur Deklaration der später verwendeten Variablen. Natürlich können die Variablen bei der Deklaration auch initialisiert werden.

Abschließend steht nach `PPCODE` der eigentliche Code, der unseren Algorithmus implementiert. `PPCODE` verwendet man statt `CODE`, wenn man sich in der XSUB gerne selbst um

```
void
min(...)
  PROTOTYPE: @

  ALIAS:
    min = 0
    max = 1

  PREINIT:
    int i;
    SV *rv;

  PPCODE:
    if (items == 0)
      XSRETURN_UNDEF;

    rv = ST(0);

    /* not the fastest way to do it      */
    /* see Scalar::Util for a better way */
    for (i = 1; i < items; i++)
      if (SvNV(ST(i)) < SvNV(rv) ^ ix)
        rv = ST(i);

    ST(0) = rv;
    XSRETURN(1);
```

Listing 1

die Verwaltung des Stacks kümmern möchte. Vom `xsubpp` wird dann z.B. kein Code erzeugt, um die `RETVAL`-Variable zurück auf den Stack zu legen.

In der Variablen `items` steht die Anzahl der Argumente, mit denen eine XSUB aufgerufen wurde. Dies ist natürlich speziell im Fall einer variablen Argumentliste von Interesse. Als erstes wird in unserer Implementierung überprüft, ob eventuell keine Argumente übergeben wurden. In diesem Fall kann man natürlich keinen kleinsten oder größten Wert ermitteln. Daher brechen wir die XSUB vorzeitig mit einem `XSRETURN_UNDEF` ab. Wie der Name schon sagt, gibt diese Funktion `undef` an den Aufrufer zurück.

Mit Hilfe des Makros `ST` kann man innerhalb des `PPCODE`-Abschnitts direkt auf ein Element auf dem Stack zugreifen. `ST(0)` greift auf das erste Element auf dem Stack zu, also das erste Argument der Funktion. Dieses ist unsere erste Näherung für das Minimum bzw. Maximum der Liste, wir speichern es daher in `rv`. Die `for`-Schleife iteriert nun über die restlichen Argumente. Wird ein kleinerer bzw. größerer Wert als der in `rv` gefunden, wird dieser in `rv` gespeichert.

Die `if`-Bedingung hat es in sich und verdient daher noch eine kurze Erläuterung: Da ist zum einen die `SvNV`-Funktion, die den numerischen Wert (`NV`) eines `SV` liefert. Sollte der `SV` noch keinen gültigen `NV`-Slot besitzen, so wird Perl versuchen, einen so gut wie möglich passenden Wert zu bestimmen. Es wird dazu gegebenenfalls ein Integerwert oder ein String in einen `NV` konvertiert. Damit sollte zumindest der Vergleich klar sein.

Bleibt zum anderen das `^ ix` am Ende der Bedingung. Wie in Perl auch ist `^` in C der binäre xor-Operator. Die linke Seite des xor-Ausdrucks ist das Ergebnis des `<`-Operators, der entweder `0` oder `1` liefert. Auf der rechten Seite steht in `ix` ebenfalls `0` oder `1`, je nachdem, ob `min` oder `max` aufgerufen wurde. Im Falle von `min` bleibt das Ergebnis des Vergleichs unverändert, denn `a ^ 0` liefert immer `a`. Im Falle von `max` wird das Ergebnis des Vergleichs negiert (`a ^ 1` ist `!a`, wenn `a` `0` oder `1` ist). Bei Aufruf der XSUB als `max` wird durch die xor-Verknüpfung aus dem `<` also quasi ein `>=`.

Zu guter Letzt legen wir unser Ergebnis in `rv` zurück auf den Stack, natürlich als erstes Element. Mit `XSRETURN(1)` sagen wir Perl, dass wir genau ein Element zurückgeben möchten.

Wenn wir jetzt `min` und `max` noch in die `@EXPORT`-Liste in `lib/AdvancedXS.pm` eintragen, können wir die Routinen einfach ausprobieren:

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for AdvancedXS
$ make
$ perl -Mblib -MAdvancedXS -le \
  '@a=(3,1,4,2); print min @a; print max @a'
1
4
```

Nachdem wir uns im vorigen Teil so intensiv mit Reference Counts beschäftigt haben, stellt sich die Frage, ob wir diesbezüglich in dem Code alles beachtet haben. Die kurze Antwort lautet: ja. Wir haben weder Referenzen auf Variablen behalten, die wir übergeben bekommen haben, noch haben wir neue Variablen angelegt, die wir sterblich hätten machen müssen. Wir haben lediglich die Variablen auf dem Stack ein wenig umsortiert. Sollten darunter Variablen sein, die bereits beim Aufruf unserer XSUB sterblich sind, so sind sie es auch danach noch. Es gibt also keinen Grund, in unserer XSUB an den Reference Counts zu manipulieren.

Histogramm eines Strings

Das nächste Beispiel soll den Umgang mit komplexen Datenstrukturen, im konkreten Fall mit Hashes, und Strings demonstrieren. Es geht darum, aus einem String einen Hash zu generieren, der als Schlüssel alle Zeichen des Strings und als Wert jeweils die absolute Häufigkeit dieses Zeichens im String enthält. Natürlich ist es fragwürdig, für dieses Problem eine XSUB zu schreiben, da es sich auch mit einer Zeile Perl lösen lässt:

```
$hash{$_}++ for $string =~ /(.)/g
```

Genau genommen ist der Perl-Code sogar deutlich besser als der im folgenden vorgestellte XS-Code, da er auch Unicode-Zeichen korrekt behandelt. Aber auch hier geht es ja in erster Linie nicht um den Sinn, sondern um XS (siehe Listing 2).

Einiges sollte uns aus dem ersten Beispiel schon bekannt vorkommen, z.B. der Prototyp oder der `PREINIT`-Teil. Neu ist, dass wir einen `SV *` als Argument erwarten und auch einen `SV *` zurückgeben. Ebenfalls neu sind der `CODE`- und der `OUTPUT`-Teil. Im `OUTPUT`-Teil teilen wir dem `xsubpp` lediglich mit, dass wir die Variable `RETVAL` an den Aufrufer zurückgeben möchten.

```
SV *
 charhist(string)
   SV *string

   PROTOTYPE: $

   PREINIT:
     UV hist[256];
     const char *str;
     STRLEN i, len;
     HV *hash;

   CODE:
     /* initialize the histogram */
     for (i = 0; i < 256; i++)
       hist[i] = 0;

     /* sv_dump(string); */

     str = SvPV_const(string, len);

     /* gather character counts */
     for (i = 0; i < len; i++)
       hist[(unsigned char) str[i]]++;

     hash = newHV();

     /* put non-zero buckets into hash */
     for (i = 0; i < 256; i++)
       if (hist[i] > 0)
       {
         char key = (char) i;
         (void) hv_store(hash, &key, 1,
                         newSVuv(hist[i]), 0);
       }

     RETVAL = newRV_noinc((SV *) hash);

     /* sv_dump(RETVAL); */

   OUTPUT:
     RETVAL
```

Listing 2

Nun aber zum `CODE`-Teil. Als erstes initialisieren wir jedes Element des C-Arrays `hist` mit 0. Das Array hat 256 Elemente vom Typ `UV` (die vorzeichenlose Variante eines `IV`), also genau ein Element für jedes Byte. Wir werden dieses Array benutzen, um die einzelnen Zeichen im String zu zählen.

Anschließend holen wir uns mittels `SvPV_const` einen Zeiger `str` auf den Anfang des übergebenen Strings sowie dessen Länge `len`. Da `SvPV_const` als C-Makro implementiert ist, kann dem Argument `len` tatsächlich ein neuer Wert zugewiesen werden. Die `_const`-Variante der SvPV-Makros verwenden wir, da wir den String nicht modifizieren, sondern nur lesen wollen.

In der nächsten Schleife wird dann die absolute Häufigkeit jedes Zeichens ermittelt. Der numerische Wert des `i`-ten Zeichens `str[i]` dient als Index in das Array `hist`, dessen Element dann inkrementiert wird.

Nun wird es interessant: Mittels `newHV` wird ein neuer Hash erzeugt. In der Schleife über das Array `hist` wird dann für jedes im String vorkommende Zeichen (absolute Häufigkeit ist größer als Null) mit `hv_store` ein neues Schlüssel-Wert-Paar in diesen Hash eingetragen. Als Schlüssel wird das Zeichen `key`, als Wert ein mit `newSVuv` neu erzeugter `SV` verwendet. Den Rückgabewert von `hv_store` ignorieren wir an dieser Stelle. Er gibt an, ob das Eintragen in den Hash erfolgreich war.

Abschließend wird mit `newRV_noinc` noch eine Referenz auf den Hash erzeugt, die dann in `RETVAL` gespeichert wird.

Tragen wir also `charhist` wieder in die `@EXPORT`-Liste ein und probieren den Code kurz aus:

```
$ make
$ perl -Mblib -MAdvancedXS -MData::Dumper \
  -le \
  'print Dumper(charhist("Hello World!"))'
$VAR1 = {
          'e' => 1,
          'r' => 1,
          'W' => 1,
          ' ' => 1,
          'd' => 1,
          'H' => 1,
          'l' => 3,
          '!' => 1,
          'o' => 2
        };
```

Überprüfen wir auch hier, ob wir in Sachen Reference Counts alles richtig gemacht haben. Betrachten wir als erstes die Werte, die wir in den Hash eintragen: ein mit `newSVuv` (oder allen anderen `newSV`-Funktionen) erzeugter Skalar hat initial einen Reference Count von 1. Wie wir der Dokumentation von `hv_store` in `perlapi` entnehmen können, wird der Reference Count des übergebenen `SV`s nicht verändert. Da der `SV` nun vom Hash, aber nicht mehr von unserem Code, referenziert wird, ist der Reference Count also korrekt.

Sehen wir uns jetzt den Hash selbst an: auch sein Reference Count ist initial 1. Der Aufruf von `newRV_noinc` verändert den Reference Count nicht. Wir übertragen mit diesem Aufruf unsere Referenz an den neu erzeugten `RV`. Auch der Reference Count des Hashes ist also in Ordnung. Die Funktion `newRV` sollte übrigens nie verwendet werden. Sie ist ein

Synonym für `newRV_inc` (was durchaus verwendet werden darf) und inkrementiert den Reference Count des ihr übergebenen `SV`s.

Bleibt noch der `RV`, den wir in RETVAL speichern. Sein Reference Count ist ebenfalls 1, es ist ein von uns erzeugter SV, und wir geben die Referenz ab. Das ist ein klares Indiz dafür, dass wir diesen SV *mortal* machen müssen. Praktischerweise ist dies auch genau das Verhalten, welches der `xsubpp` implementiert, wenn wir von einer Funktion mit `CODE`-Teil explizit einen `SV *` zurückgeben. Schauen wir uns das kurz im generierten `Advanced.c` an:

```
    RETVAL = newRV_noinc((SV *) hash);

    /* sv_dump(RETVAL); */

#line 112 "AdvancedXS.c"
        ST(0) = RETVAL;
        sv_2mortal(ST(0));
    }
    XSRETURN(1);
}
```

Wir sehen, dass der vom `xsubpp` erzeugte Code zuerst `RETVAL` als erstes Element auf den Stack legt. In der nächsten Zeile macht er dann durch den Aufruf von `sv_2mortal` genau dieses erste Element auf dem Stack sterblich.

Dieses Verhalten des `xsubpp` ist zwar normalerweise praktisch, kann aber in Einzelfällen auch zu Problemen führen, nämlich gerade dann, wenn man die Referenz auf den `SV` *nicht* abgibt. Man sollte sich dessen also immer bewusst sein, wenn man als Rückgabetyp bei einer XSUB `SV *` angibt.

So wie es aussieht, ist aber auch bei dieser Funktion mit den Reference Counts alles im grünen Bereich. Eine andere Möglichkeit, das zu überprüfen, bietet `Devel::Peek` (siehe Listing 3).

Wir sehen hier die Perl-interne Repräsentation der zurückgegebenen Hashreferenz. Anhand der `REFCNT`-Einträge können wir verifizieren, dass die Reference Counts sowohl für den `RV` (ein `IV`, den wir am `ROK`-Flag erkennen können), als auch für den Hash (`SVt_PVHV`) und seine Werte (`SVt_IV`) jeweils 1 sind. Da es genau ein referenzierendes Objekt, nämlich `$a`, gibt, sind die Reference Counts alle korrekt.

Die Fibonacci-Folge

Die Fibonacci-Folge dürfte jedem bekannt sein, der eine Informatik-Vorlesung besucht hat, da sie immer als Paradebeispiel für Rekursionen missbraucht wird. Die *n*-te Zahl der Folge ist jeweils die Summe der beiden vorhergehenden Zahlen. Für die beiden ersten Zahlen werden 0 und 1 angenommen. Die klassische rekursive Lösung sieht in Perl z.B. so aus:

```
#!/usr/bin/perl -w

sub fibo
{
  my $f = shift;
  $f < 2 ? $f : fibo($f - 1) + fibo($f - 2);
}

print fibo($_), " " for 0 .. shift || 10;
printf "\n%.2f s\n", (times)[0];
```

Die Funktion `fibo` gibt die *n*-te Zahl der Folge zurück. Das Programm gibt somit die Fibonacci-Folge bis zur *n*-ten Zahl, wobei *n* optional als Argument übergeben werden kann. Weiterhin gibt das Programm die Zeit aus, die für die Berechnung der Folge benötigt wurde.

```
$ perl fibonacci.pl
0 1 1 2 3 5 8 13 21 34 55
0.01 s
```

Wie wir sehen, geht die Berechnung der ersten 11 Zahlen der Folge recht schnell. Das Manko des rekursiven Ansatzes sehen wir erst, wenn wir versuchen, eine längere Folge auszugeben:

```
$ perl -Mblib -MAdvancedXS -MDevel::Peek \
  -le'$a=charhist("FOO"); Dump $a'
SV = IV(0x81cc29c) at 0x81cc2a0
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x815c1e0
  SV = PVHV(0x81619d0) at 0x815c1e0
    REFCNT = 1
    FLAGS = (SHAREKEYS)
    ARRAY = 0x81683f8  (0:6, 1:2)
    hash quality = 125.0%
    KEYS = 2
    FILL = 2
    MAX = 7
    RITER = -1
    EITER = 0x0
    Elt "F" HASH = 0xfe39fc60
    SV = IV(0x815c3ac) at 0x815c3b0
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 1
    Elt "O" HASH = 0x80037ff1
    SV = IV(0x815c0bc) at 0x815c0c0
      REFCNT = 1
      FLAGS = (IOK,pIOK)
      IV = 2
```

Listing 3

```
$ perl fibonacci.pl 30
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
987 1597 2584 4181 6765 10946 17711 28657
46368 75025 121393 196418 317811 514229
832040
2.99 s
```

Sehr effizient ist diese Implementierung also nicht. Dass man die Folge auch iterativ berechnen kann - und zwar wesentlich effizienter -, ist nicht schwer zu sehen. Auch der folgende XS-Code verwendet eine iterative Lösung.

```
void
fibonacci(n = 10)
  UV n

  PROTOTYPE: ;$

  PREINIT:
    UV i, x[2];

  PPCODE:
    EXTEND(SP, n);

    for (i = 0; i < n; i++)
    {
      x[i%2] = i < 2 ? i : x[0] + x[1];
      ST(i) = sv_2mortal(newSVuv(x[i%2]));
    }

    XSRETURN(n);
```

Die XSUB `fibonacci` gibt die ersten `n` Zahlen der Fibonacci-Folge zurück. Zum ersten Mal haben wir jetzt bei einer XS-Funktion mehr als einen Rückgabewert. Im Grunde ist das gar nicht schwer: Man verwendet einfach den Argumenten-Stack für die einzelnen Rückgabewerte und setzt abschließend mit `XSRETURN` die Anzahl der Rückgabewerte.

Es gibt allerdings zwei kritische Punkte. Zum einen, wie schon zuvor, die Reference Counts der Rückgabewerte; zum anderen muss sichergestellt werden, dass auf dem Stack genug Platz für alle Rückgabewerte ist. Letzteres können wir z.B. mit dem Makro `EXTEND` sicherstellen. `EXTEND(SP, n)` sorgt dafür, dass auf dem Stack (`SP` steht für Stack Pointer), Platz für mindestens `n` Elemente bereitgestellt wird.

Die Problematik der Reference Counts ist uns ja mittlerweile gut bekannt, daher ist das Ablegen der Folge auf dem Stack kein großes Problem mehr. Wir erzeugen für jede Zahl der Folge einen neuen `UV` und machen diesen vor dem Ablegen auf dem Stack *mortal*.

Es gibt aber noch etwas neues an dieser XSUB: Wenn wir uns den Kopf anschauen, sehen wir, dass dem Argument `n` die Zahl 10 zugewiesen wird. Auf diese Art lassen sich in XS Default-Werte für einzelne Argumente angeben. In unserem Fall kann mal `fibonacci` demnach ohne Argumente aufrufen und bekommt die ersten zehn Zahlen der Folge zurückgeliefert.

```
$ make
$ perl -Mblib -MAdvancedXS -le \
  'print join ", ", fibonacci'
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Wenn man, wie in unserem Beispiel, die Rückgabewerte sequenziell auf den Stack legen möchte, gibt es noch eine andere Möglichkeit, dies zu tun. Statt jedem Element auf dem Stack explizit mit `ST` einen Wert zuzuweisen, kann man die verschiedenen `PUSH`-Makros verwenden. Diese Makros legen jeweils genau einen Wert auf den Stack. Dabei gibt das Suffix des Makros an, welchen Typ der Wert hat:

```
--------------
 Suffix   Typ
--------------
   s      SV
   p      PV
   n      NV
   i      IV
   u      UV
--------------
```

Wenn man dem Makro zusätzlich das Präfix `m` voranstellt, werden die auf den Stack gelegten Werte als sterblich markiert. Somit kann man also z.B. mit `mPUSHu` einen sterblichen UV auf den Stack legen. Praktischerweise ist das genau das Makro, das wir für unsere Fibonacci-Funktion brauchen:

```
void
fibonacci2(n = 10)
  UV n

  PROTOTYPE: ;$

  PREINIT:
    UV i, x[2];

  PPCODE:
    EXTEND(SP, n);

    for (i = 0; i < n; i++)
    {
      x[i%2] = i < 2 ? i : x[0] + x[1];
      mPUSHu(x[i%2]);
    }

    XSRETURN(n);
```

Diese neue Funktion ist vollkommen äquivalent zu unserer ersten Implementierung, lediglich die Zeile, in der wir den Rückgabewert auf den Stack legen, ist nun etwas übersichtlicher. Natürlich funktioniert auch diese Funktion:

```
$ make
$ perl -Mblib -MAdvancedXS -le \
  'print join ", ", fibonacci2(10)'
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Es geht aber noch einfacher: Will man sich nicht selbst mit `EXTEND` um das Erweitern des Stacks kümmern oder weiß man vorher nicht, wie viele Elemente man insgesamt auf den Stack legen will, kann man mit den `XPUSH`-Makros arbeiten. Das `X` steht für *eXtend*, diese Makros stellen also sicher, dass immer genug Platz für den Wert auf dem Stack vorhanden ist. Zu jedem `PUSH`-Makro gibt es ein korrespondierendes `XPUSH`-Makro. Damit sieht die dritte Variante der Fibonacci-Funktion so aus:

```
void
fibonacci3(n = 10)
  UV n

  PROTOTYPE: ;$

  PREINIT:
    UV i, x[2];

  PPCODE:
    for (i = 0; i < n; i++)
    {
      x[i%2] = i < 2 ? i : x[0] + x[1];
      mXPUSHu(x[i%2]);
    }

    XSRETURN(n);
```

Auch diese Variante ist funktional identisch:

```
$ make
$ perl -Mblib -MAdvancedXS -le \
  'print join ", ", fibonacci3(10)'
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Es gibt im Übrigen eine einfache Möglichkeit, zu überprüfen, ob man beim Reference Counting alles richtig gemacht hat. Mit `DumpArray` aus dem Modul `Devel::Peek` kann man sich leicht die interne Repräsentation aller Rückgabewerte einer Funktion ansehen:

```
$ perl -Mblib -MAdvancedXS -MDevel::Peek \
   -le 'DumpArray 0, fibonacci3(3)'
 Elt No. 0  0x815c1e0
 SV = IV(0x815c1dc) at 0x815c1e0
   REFCNT = 1
   FLAGS = (TEMP,IOK,pIOK)
   IV = 0
 Elt No. 1  0x815c3b0
 SV = IV(0x815c3ac) at 0x815c3b0
   REFCNT = 1
   FLAGS = (TEMP,IOK,pIOK)
   IV = 1
 Elt No. 2  0x815c0c0
 SV = IV(0x815c0bc) at 0x815c0c0
   REFCNT = 1
   FLAGS = (TEMP,IOK,pIOK)
   IV = 1
```

Bei einem neu erzeugten Rückgabewert, auf den es keine weiteren Referenzen gibt, sollte der `REFCNT` genau auf `1` stehen und weiterhin das `TEMP`-Flag gesetzt sein. Letzteres ist ein Hinweis darauf, dass der `SV` *mortal* ist, in Kürze also der Reference Count dekrementiert wird und der `SV` damit stirbt.

Typemaps

Ab und zu kann es vorkommen, dass einem die Typen, mit denen der `xsubpp` standardmäßig umgehen kann, nicht genügen. In solchen Fällen kann man dem `xsubpp` mit Hilfe einer `typemap`-Datei beibringen, wie er mit bestimmten neuen Typen umgehen soll.

Eine `typemap`-Datei besteht im einfachsten Fall lediglich aus einem `TYPEMAP`-Abschnitt, in dem neue Typen auf bekannte Typen abgebildet werden. Welche bekannten Typen es gibt, kann man in der `typemap`-Datei des `ExtUtils`-Moduls nachlesen.

Nehmen wir einmal an, wir haben die folgende XSUB geschrieben, welche alle ohne Rest durch `divisor` teilbaren Argumente zurückliefert:

```
void
dividable(divisor, ...)
  divisor_type divisor

  PROTOTYPE: $@

  PREINIT:
    UV i, c;

  PPCODE:
    for (c = 0, i = 1; i < items; i++)
    {
      SV *sv = ST(i);

      if (SvUV(sv) % divisor == 0)
      {
        ST(c) = sv;
        c++;
      }
    }

    if (GIMME_V == G_ARRAY)
      XSRETURN(c);
    else
      XSRETURN_UV(c);
```

Der Typ von `divisor` ist `divisor_type`. Diesen Typ gibt es noch nicht, aber wir können ihn beispielsweise folgendermaßen im C-Teil unserer XS-Datei definieren:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

typedef unsigned int divisor_type;

MODULE = AdvancedXS     PACKAGE = AdvancedXS
```

Es handelt sich bei `divisor_type` also um einen vorzeichenlosen Integerwert, einen `UV`. `ExtUtils::typemap` weiß bereits, wie mit einem `UV` umzugehen ist, und bietet dafür `T_UV` an. Wir müssen also lediglich eine `typemap`-Datei mit folgendem Inhalt in unserem Modulverzeichnis anlegen:

```
TYPEMAP
divisor_type     T_UV
```

Dadurch *erbt* `divisor_type` die in den `INPUT`- und `OUTPUT`-Abschnitten von `ExtUtils::typemap` definierten Eigenschaften, und der `xsubpp` ist in der Lage, den richtigen Code für die Umwandlung einer Perl-Variablen in einen `divisor_type` und umgekehrt zu erzeugen.

```
$ make
$ perl -Mblib -MAdvancedXS -le \
  'print join ", ", dividable(3, 1 .. 20)'
3, 6, 9, 12, 15, 18
```

Schauen wir uns die XSUB aber nochmal etwas genauer an. Wir finden an deren Ende einiges, das wir noch nicht kennen. Mit Hilfe von `GIMME_V` kann man - analog zu `wantarray` in Perl - feststellen, in welchem Kontext eine XSUB aufgerufen wurde. `GIMME_V` kann `G_VOID`, `G_SCALAR` oder eben `G_ARRAY` zurückgeben. Den Fall, dass die XSUB im Listenkontext aufgerufen wird, haben wir ja oben ausprobiert.

Im Skalar- oder Voidkontext wird nun `XSRETURN_UV` aufgerufen. Dies ist lediglich eine nützliche Abkürzung, um sofort einen einzelnen `UV` zurückzugeben. Statt also die Elemente selbst zurückzugeben, gibt `dividable` im Skalarkontext die Anzahl der Elemente zurück:

```
$ perl -Mblib -MAdvancedXS -le \
  'print scalar dividable(3, 1 .. 20)'
6
```

Debugging

Bis jetzt hat ja alles wunderbar geklappt! Was aber, wenn die neu geschriebene XSUB nicht so richtig funktionieren will? Dann muss man früher oder später einmal mit dem Debugger ran. Wie das mit C-Code funktioniert, haben wir bereits im ersten Teil gesehen. Mit XS-Code ist es im Grunde nicht viel anders, trotzdem gibt es ein paar Dinge zu beachten.

Als erstes muss der Code wieder mit Debug-Symbolen übersetzt werden. Am einfachsten ist es, sich dafür einen eigenen Perl-Interpreter zu bauen, der mit Debug-Symbolen übersetzt ist, da alle Module die Compileroptionen des Interpreters erben. Dazu muss man auf *nix-Systemen dem `Configure`-Skript lediglich die Option `-Doptimize=-g` übergeben, unter Windows muss man in der Datei `win32/Makefile` das Kommentarzeichen in der Zeile

```
#CFG            = Debug
```

entfernen.

Anschließend kann man das XS-Modul mit diesem Perl-Interpreter neu bauen:

```
$ make realclean
$ debugperl Makefile.PL
$ make
```

Sehen wir uns in Listing 4 nun einmal die `fibonacci`-Funktion im Debugger an.

Gleich das erste Kommando ist sehr wichtig: Wir setzen einen Breakpoint auf die Funktion `perl_run`. Der Grund dafür ist, dass die Funktion, auf die wir eigentlich einen Breakpoint setzen wollen, noch gar nicht verfügbar ist. Sie wird erst später vom Perl-Interpreter geladen. Wenn `perl_run` ausgeführt wird, ist das Perl-Programm kompiliert, alle mit `use` (oder `-M`) eingebundenen Module sind geladen, das Programm wird aber noch nicht ausgeführt.

Jetzt können wir einen Breakpoint auf die `fibonacci`-Funktion setzen. Leider heißt diese nicht einfach `fibonacci`. Der `xsubpp` hat ihr einen deutlich komplizierteren Namen gegeben: `XS_AdvancedXS_fibonacci`. Zum Glück kann der `gdb` aber Symbolnamen automatisch vervollständigen. Da alle XSUBs mit `XS_` beginnen, ist man schnell am Ziel.

Mit `n 8` begeben wir uns mitten in die `for`-Schleife. Statt `n 8` kann man auch achtmal nacheinander `n` eingeben. Jetzt lassen wir uns mit dem extrem nützlichen Befehl `display` die Ausdrücke `i`, `x` und `x[i%2]` anzeigen. `display` funktioniert im Grunde wie `print`, nur dass die Ausdrücke bei jedem weiteren Schritt erneut angezeigt werden.

```
$ gdb -q debugperl
Using host libthread_db library
"/lib/libthread_db.so.1".
(gdb) b perl_run
Breakpoint 1 at 0x810b2bf: file perl.c, line
2302.
(gdb) r -Mblib -MAdvancedXS -e fibonacci
Starting program:
/home/mhx/perl/blead-debug-nothread/bin/perl
-Mblib -MAdvancedXS -e fibonacci

Breakpoint 1, perl_run (my_perl=0x83a5008)
at perl.c:2302
2302        int ret = 0;
(gdb) b XS_AdvancedXS_fibonacci
Breakpoint 2 at 0x400226ce: file
AdvancedXS.c, line 124.
(gdb) c
Continuing.

Breakpoint 2, XS_AdvancedXS_fibonacci
(cv=0x8428128) at AdvancedXS.c:124
124         dVAR; dXSARGS;
(gdb) n 8
96              ST(i) =
sv_2mortal(newSVuv(x[i%2]));
(gdb) display i
1: i = 0
(gdb) display x
2: x = {0, 20}
(gdb) display x[i%2]
3: x[i % 2] = 0
(gdb) n 3
96              ST(i) =
sv_2mortal(newSVuv(x[i%2]));
3: x[i % 2] = 1
2: x = {0, 1}
1: i = 1
(gdb) n 3
96              ST(i) =
sv_2mortal(newSVuv(x[i%2]));
3: x[i % 2] = 1
2: x = {1, 1}
1: i = 2
(gdb) n 3
96              ST(i) =
sv_2mortal(newSVuv(x[i%2]));
3: x[i % 2] = 2
2: x = {1, 2}
1: i = 3
(gdb) n 3
96              ST(i) =
sv_2mortal(newSVuv(x[i%2]));
3: x[i % 2] = 3
2: x = {3, 2}
1: i = 4
(gdb) q
The program is running.  Exit anyway? (y or
n) y
```

Listing 4

Gehen wir nun mit `n 3` immer jeweils einen Schleifendurchlauf weiter, können wir sehr schön die Entwicklung der Variablen `i` und `x` verfolgen und überprüfen, ob der Algorithmus wie erwartet funktioniert.

Etwas komplizierter wird es, wenn die Ausdrücke, für die man sich interessiert, Makros enthalten. Der Debugger ist nicht in der Lage, diese Makros aufzulösen, und wird daher eine Fehlermeldung ausgeben, zum Beispiel:

```
Breakpoint 2, XS_AdvancedXS_min
(cv=0x842f050) at AdvancedXS.c:28
28          dVAR; dXSARGS;
(gdb) n 4
27          rv = ST(0);
(gdb) p ST(0)
No symbol "ST" in current context.
```

Da es in Perl eine ganze Menge Makros gibt, steht man nicht selten vor diesem Problem. In der Perl-Distribution gibt es aber seit einiger Zeit ein kleines Skript, mit dessen Hilfe sich die Makros auflösen lassen. Dazu muss jedoch das Verzeichnis, in dem der Perl-Interpreter gebaut wurde, noch existieren. In diesem Verzeichnis kann man dann das Skript `Porting/expand-macro.pl` starten:

```
$ perl Porting/expand-macro.pl 'ST(0)'
`sh  cflags "optimize='-ggdb3'" try.c` -E
try.c > try.i
          CCCMD =  gcc -DPERL_CORE -c
-DPERL_PATCHNUM=35082 -DDEBUGGING
-fno-strict-aliasing -pipe -fstack-protector
-D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64
-std=c89 -ggdb3 -Wall -ansi -W -Wextra
-Wdeclaration-after-statement -Wendif-labels
-Wc++-compat
# 4 "ST expands to"
PL_stack_base[ax + (0)]
```

Als Argument kann man dem Skript den Namen eines Makros oder ein Makro mit Argumenten übergeben. In der letzten Zeile der Ausgabe steht dann der aufgelöste Ausdruck. Diesen kann man dann direkt im Debugger verwenden:

```
(gdb) p PL_stack_base[ax + (0)]
$1 = (SV *) 0x83a82a0
```

Ab und zu kommt es auch vor, dass man gerne einfach nur den Inhalt oder Aufbau eines `SV`s sehen möchte, ohne gleich den Debugger bemühen zu müssen. Auch dafür gibt es eine Möglichkeit. Man kann die Funktion `sv_dump` verwenden. In unserer XS-Datei befinden sich in der XSUB `charhist` zwei auskommentierte Zeilen, die `sv_dump` benutzen. Entfernt man die Kommentarzeichen und übersetzt das Modul erneut, ergibt sich folgendes Verhalten:

```
$ make
$ debugperl -Mblib -MAdvancedXS -e \
  'charhist("Hello World!")'
SV = PV(0x83f3018) at 0x83b9348
  REFCNT = 1
  FLAGS = (POK,READONLY,pPOK)
  PV = 0x8449cf0 "Hello World!"\0
  CUR = 12
  LEN = 16
SV = IV(0x83b93a4) at 0x83b93a8
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x83a81e0
    SV = PVHV(0x83ad930) at 0x83a81e0
      REFCNT = 1
      FLAGS = (SHAREKEYS)
      ARRAY = 0x8448e60  (0:11, 1:2, 2:2,
3:1)
      hash quality = 71.1%
      KEYS = 9
      FILL = 5
      MAX = 15
      RITER = -1
      EITER = 0x0
        Elt "e" HASH = 0x11162210
        SV = IV(0x83a829c) at 0x83a82a0
          REFCNT = 1
          FLAGS = (IOK,pIOK)
          IV = 1
        Elt "r" HASH = 0x26014be2
        SV = IV(0x83b93c4) at 0x83b93c8
          REFCNT = 1
          FLAGS = (IOK,pIOK)
          IV = 1
        Elt "W" HASH = 0x11f523d2
        SV = IV(0x83b9314) at 0x83b9318
          REFCNT = 1
          FLAGS = (IOK,pIOK)
          IV = 1
```

Diese Art der Ausgabe sollte uns von `Devel::Peek` bekannt vorkommen. Und in der Tat verwenden sowohl `Devel::Peek` als auch `sv_dump` die gleichen Funktionen zur Ausgabe.

Im Geschwindigkeitsrausch

Zu Beginn des Tutorials wurde bereits erwähnt, dass man XS sehr gut verwenden kann, um die Geschwindigkeit von Algorithmen zu steigern. Aber wie groß ist der Unterschied zwischen einer reinen Perl- und einer XS-Implementierung wirklich?

Von Fall zu Fall ist das natürlich verschieden. Das Modul `Digest::CRC` ist jedoch ein sehr schönes Beispiel, denn es zeigt, wie man mit relativ wenig Aufwand aus einem Perl-Modul ein hybrides Perl/XS-Modul machen kann. Der Vorteil des hybriden Ansatzes ist, dass das Modul weiterhin auch ohne einen C-Compiler installiert werden kann. Auf den Performancegewinn muss man dann allerdings verzichten.
Aber schauen wir uns einmal im Detail an, welche Änderungen an `Digest::CRC` vorgenommen wurden.

Der XS-Teil kam mit Version 0.09 zu `Digest::CRC` hinzu, man kann also einfach die Versionen 0.08 und 0.09 miteinander vergleichen. Die Hybridisierung besteht im Wesentlichen aus einer Compilererkennung in `Makefile.PL` und dem evaluieren der Perl-Implementierung in `lib/Digest/CRC.pm` für den Fall, dass die XS-Implementierung nicht geladen werden kann. Die XS-Implementierung in `CRC.xs` besteht lediglich aus der C-Funktion `reflect` sowie den XSUBs `_tabinit` und `_crc`. Die entsprechenden Perl-Implementierungen sind `_reflect`, `_tabinit` und `_crc` in `lib/Digest/CRC.pm`.

Die jeweiligen Perl- und XS-Routinen sind sehr ähnlich aufgebaut, so dass man hier noch einmal die Gemeinsamkeiten und Unterschiede zwischen Perl- und C-Code studieren kann. Sehen wir uns als Beispiel einmal `_crc` an:

```
sub _crc {
  my ($message,$width,$init,$xorout,$refin,
      $refout,$tab) = @_;
  my $crc = $init;
  $crc = _reflect($crc,$width) if $refin;
  my $pos = -length $message;
  my $mask = 2**$width-1;
  while ($pos) {
    if ($refin) {
      $crc = ($crc>>8)^$tab->[($crc^ord(
        substr($message, $pos++, 1)))&0xff]
    } else {
      $crc = (($crc<<8))^$tab->[(($crc>>
        ($width-8))^ord(substr $message,
                        $pos++,1))&0xff]
    }
  }

  if ($refout^$refin) {
    $crc = _reflect($crc,$width);
  }

  $crc = $crc ^ $xorout;
  $crc & $mask;
}
```

Die entsprechende XSUB sieht aus, wie in Listing 5 dargestellt.

Der algorithmische Teil ist in der Tat weitgehend identisch. Auch das Verarbeiten von Perl-Strings mit Hilfe von `SvPV` haben wir schon kennen gelernt. Das Einzige, das uns wirklich zu denken geben sollte, ist die Funktion `SvGETMAGIC`. Sie dient dazu, den tatsächlichen Wert einer Perl-Variablen, die *magische* Eigenschaften besitzt, zu evaluieren. Dies kann beispielsweise eine `tie`-Variable sein, oder auch ein Rückgabewert von `substr`. Der Aufruf ist allerdings an dieser Stelle völlig überflüssig, da `SvPV` implizit ebenfalls

SvGETMAGIC aufruft. Um genau zu sein, ist der explizite Aufruf von SvGETMAGIC sogar falsch, da auf diese Weise bei einer tie-Variablen zweimal FETCH aufgerufen wird. Im konkreten Fall stellt dies jedoch kein Problem dar, denn _crc wird nie direkt mit einer Variablen mit magischen Eigenschaften aufgerufen. Typischerweise wird man SvGETMAGIC und ihr Pendant SvSETMAGIC nur sehr selten wirklich brauchen.

Nachdem der XS-Code an sich also wenig Neues zu bieten hat, wollen wir einen Blick auf die Geschwindigkeit werfen. Digest::CRC bieten verschiedene Routinen an, um Standard-CRC-Prüfsummen zu berechnen. Mit diesen Routinen läßt sich schnell ein kleines Benchmark-Skript schreiben:

```
SV *
_crc(message, width, init, xorout, refin, \
     refout, table)
  SV *message
  IV  width
  UV  init
  UV  xorout
  IV  refin
  IV  refout
  SV *table

  PREINIT:
    UV crc, mask, *tab;
    STRLEN len;
    const char *msg, *end;

  CODE:
    SvGETMAGIC(message);

    crc  = refin ? reflect(init, width)
                 : init;
    msg  = SvPV(message, len);
    end  = msg + len;
    mask = ((UV)1)<<(width-1);
    mask = mask + (mask-1);
    tab  = (UV *) SvPVX(table);

    if (refin) {
      while (msg < end)
        crc = (crc >> 8) ^
              tab[(crc ^ *msg++) & 0xFF];
    }
    else {
      int wm8 = width - 8;
      while (msg < end)
        crc = (crc << 8) ^ tab[((crc >> wm8)
                           ^ *msg++) & 0xFF];
    }

    if (refout ^ refin)
      crc = reflect(crc, width);

    crc = (crc ^ xorout) & mask;

    RETVAL = newSVuv(crc);

  OUTPUT:
    RETVAL
```

Listing 5

```
#!/usr/bin/perl
use strict;
use warnings;
use Digest::CRC qw( crc8 crc16
                    crcccitt crc32 );
use Benchmark;

my $input = join '', 'aa' .. 'zz';
print "length: ", length $input, " bytes\n";

timethese(-3, {
  crc8     => sub { crc8($input) },
  crc16    => sub { crc16($input) },
  crcccitt => sub { crcccitt($input) },
  crc32    => sub { crc32($input) },
});
```

Für Version 0.08 von Digest::CRC erhalten wir folgendes Resultat:

```
length: 1352 bytes
     crc16: 455.76/s (n=1463)
     crc32: 433.44/s (n=1374)
      crc8: 303.70/s (n=984)
  crcccitt: 307.50/s (n=984)
```

Version 0.09 mit XS-Unterstützung liefert folgendes Ergebnis:

```
length: 1352 bytes
     crc16: 54803.24/s (n=169342)
     crc32: 53488.60/s (n=164210)
      crc8: 54797.49/s (n=174804)
  crcccitt: 54276.28/s (n=169342)
```

Die XS-Version ist also zwischen 120 und 180 mal schneller als die ursprüngliche Perl-Implementierung. Ein durchaus beachtliches Ergebnis, wenn man den relativ geringen Aufwand für den XS-Teil bedenkt.

Ausblick

Der letzte Teil behandelt Themen wie die Portabilität von XS-Code, Exception Handling und objektorientierte XS-Programmierung.

#Marcus Holland-Moritz

Perl 6 Tutorial - Teil 6 : Objekte und Roles

Willkommen und hereinspaziert mein Damen und Herren. Hier erleben sie Menschen - Tiere - Sensationen, so etwas hat die Welt noch nicht gesehen. Und fürwahr das Objektsystem von Perl 6 übertrifft in seiner Ausgereiftheit und Vielseitigkeit alles bisher aus der Praxis Bekannte. Hier gab es einige der stärksten Änderungen gegenüber Perl 5. Vieles davon ist jedoch heute bereits (leicht abgeändert) mit Moose umsetzbar.

Das alte System

Die alte OOP bestach vor allem durch ihre Freiheit und dass sie mit minimalstem syntaktischen Aufwand in das objektlose Perl 4 eingefügt wurde. Nur `bless` (der Befehl zur Erzeugung eines Perl 5-Objektes) kam extra deshalb dazu. Und dieser Befehl bindet "lediglich" eine Referenz - meist eine Hashreferenz - an ein `package` und macht sie so zum Objekt. Damian Conway bewundert diesen "Taschenspielertrick", und zeigt in seinem Buch *"Object Oriented Perl"* die immensen Möglichkeiten des "Tricks" auf. Larry hatte diesen "Trick" eigentlich von Python abgeschaut, es aber seitdem bereut. Nicht nur Guido van Rossum hat mit der kürzlich erschienenen Version 3 diese Art der OOP aus Python gestrichen, auch Perlkreise sehen dieses Objektsystem derzeit als größte Schwachstelle der Perlsyntax. Es ist nicht nur für viele Perlneulinge eine echte Hürde, sondern verstößt auch gegen Perls eigene Regeln.

Was Perl ausmacht, zeigt das alltägliche Grundbeispiel, das gleichzeitig den ersten Babyschritten jedes Programmierers entspricht: `print "Ich kann schon laufen!\n"`. Keine Systembibliothek wird importiert, keine `main`-Routine ist notwendig, keine Klasse muss drumherum gestrickt werden. Dem Programmierer wird weder Arbeit noch Wissen abverlangt, das nicht unbedingt notwendig ist. In Perl 6 und 5.10 reicht sogar `say "Ich kann schon laufen!"`, da die Bedeutung von Steuerzeichen sicher nicht das Erste ist, was Anfänger lernen sollten. Wenn er jedoch eine einfache Klasse schreiben will, braucht er einige Kenntnis von Namensräumen, Referenzen, Hashes, `bless` und Perls eigenwilliger Art der Parameterübergabe, wie es das folgende Beispiel zeigt:

Perl 5:

```
package Heart::Gold;

sub new {
    bless {speed => 0 }, shift;
}

sub speed {
  my $self = shift;
  my $speed = shift;
  if ($speed) { $self->{speed} = $speed }
  else        { $self->{speed} }
}

sub stop {
  my $self = shift;
  $self->{speed} = 0;
}
```

Benutzt wird diese Klasse wie folgt:

Perl 5:

```
my $ufo = Heart::Gold->new();
# "gib Gas, Scotty"
$ufo->speed('7c');
# "alle Maschinen halt"
$ufo->stop();
```

Für gute Perl-Programmierer ist das kein Problem, aber sie freuen sich über Wege, das Gleiche mit weniger Tippaufwand zu erreichen, wie die Existenz von *Moose*, *Class::Accessor::Fast* und mindestens einem Dutzend weiterer Module beweist. Tipparbeit zu reduzieren ist ebenfalls ein wichtiges Perlziel.

... und nun das Neue

Eine fast funktionsgleiche Klasse sähe in Perl 6 so aus:

```
class Heart::Gold {
    has $.speed;
    method stop   { $speed = 0 }
}
```

Das ist kurz, klar und weniger fehleranfällig. Bereits der erste Befehl kündigt unmissverständlich an, dass hier eine Klasse erzeugt wird (deklarativer Syntax). `class` ist (genau genommen) ein *Blockmodifikator*, der dem nachfolgendem (in geschweiften Klammern stehenden) lexikalischen Bereich einen Namen zuweist (wie `package`) und daraus ein Protoobjekt erzeugt aus dem nachfolgend Objekte geklont werden können. Perl 6 hat eine prototyp-basierte OOP, ähnlich Javascript. Möglich ist auch die Schreibweise `class Heart::Gold;` ohne geschweifte Klammern. Dann nimmt der Interpreter jedoch an, dass alles bis zum Dateiende zu jener Klasse gehört.

Der zweite Befehl (`has`) definiert ein Attribut der Klasse. Attribute sind die Daten die zu jedem Objekt gehören und die von außen unsichtbar sind. (Das wäre in Perl 5 nicht einfach möglich.) Auf manche dieser Daten ist aber der Zugriff von außen gewünscht. In vielen Sprachen werden deshalb Methoden geschrieben die Werte von und zu den Attributen weiterleiten. In Perl 6 kann der Programmierer mit der Twigil (dem Punkt nach "$") die öffentlichen Attribute markieren und die Methode gleichen Namens, die einen Zugriff von außen erlaubt, wird automatisch erzeugt. Möchte man neben dem "Getter" auch einen "Setter" (Methode, die dem Attribut einen Wert zuweist), muss man das Attribut mit dem Trait (Eigenschaft die während der Kompilierung unabänderlich festgelegt wird - zu deutsch: Charakterzug) `rw` versehen.

```
has $.speed is rw;
```

`rw` steht für "read/write" (lesen/schreiben) und wird zu vielen Gelegenheiten angewendet, z.B. wenn eine Datei für Lese- und Schreibzugriff geöffnet wird. Der erzeugte Setter ist eine Lvalue-Routine. Ihm kann also zugewiesen werden. Das erlaubt nicht nur einen einfachen, intuitiven Umgang, da Getter und Setter sich wie Variablen verhalten, es vermeidet auch hinterhältige Fallen, die eine selbstgeschriebene, kombinierte Getter/Setter-Methode aufstellt. Sie sind auch angenehmer im Umgang als die Getter/Setter, die in unserem Beispiel "GetSpeed" und "SetSpeed" oder "get_speed" / "set_speed" heißen würden. Sind alle oder die Mehrheit des Attributs "rw", ist es kürzer zu schreiben:

```
class Heart::Gold is rw { ...
```

Soll ein Attribut privat sein, wird es mit "!" (Punkt mit Wand darüber) markiert. Der automatisch erzeugte Getter ist dann eine private Methode, die nur innerhalb der Klasse benutzt werden kann. Ohne Twigil wird keine Zugriffsmethode erzeugt. Das ist auch nicht notwendig, da jedes Attribut innerhalb der Klasse einer lexikalisch lokalen Variable entspricht. Deshalb kann es auch kein gleichnamiges privates und öffentliches Attribut geben. (Innerhalb der Beispielklasse greift `$.speed`, `$!speed` und `$speed` auf das gleiche Attribut zu.) Eine lokale Klassenvariable (kein Attribut) ist jedoch möglich, wenn auch nicht empfehlenswert, es der Autor wünscht sich eine Variable die alle Instanzen der Klasse teilen. Dies hieße ebenfalls `$speed`, während das Attribut weiterhin mit `$!speed` in der Klasse erreichbar wäre, in der es definiert wurde. Die Befehle `my` und `our` haben bei *Accessoren* (Getter/Setter) keinen Effekt.

Der dritte neue Befehl ist `method`, der selbsterklärend (anstatt `sub`) eine Methode erzeugt (deklarativer Syntax). Methoden sind Routinen eines Objektes. Objekte wurde ja "erfunden" um Daten mit zugehörigen Routinen zusammenzufassen. Sie werden nun mit `$objekt.methode()` anstatt `$objekt->methode()` aufgerufen. Das spart ein Zeichen zu tippen, liest sich leichter und ist verbreiteter Standard. Mit diesem Wissen wird klar, warum `$.speed` der Befehl ist, eine gleichnamige Methode zu erzeugen.

Verwendung von Objekten

Ein weiterer großer Unterschied zum Perl 5-Beispiel ist das Fehlen der Methode `new`. Die wird automatisch erzeugt, da der Interpreter auf alle nötigen Informationen zugreifen kann. Bei Bedarf kann allerdings eine eigene `new`-Methode oder ebenso eigene Getter/Setter erstellt werden, welche die autogenerierten Methoden überschreiben. Die Verwendung einer Klasse sieht damit so aus:

```
my $ufo = Heart::Gold.new();
$ufo.speed ='5c';
$ufo.stop;
# 0, da numerischer Kontext erzwungen wird
say + $ufo.speed;
```

Die ersten beiden Schritte ließen sich sogar noch zusammenfassen:

```
my $ufo = Heart::Gold.new( speed =>'5c');
#oder
my Heart::Gold $ufo .= new( speed =>'5c');
```

Praktischerweise kann die autogenerierte `new`-Methode auch Attribute befüllen. Im zweiten Fall wird der Variable die Klasse (Protoobjekt) als Datentyp zugewiesen. Gemäß der Logik selbszuweisender Operatoren (z.B."+=") wird .new auf den Aufrufer angewendet und so entsteht ebenfalls ein Objekt. Diese Art der "selbstzuweisenden" Methoden wird *Mutator* genannt (z.B. auch `@array .= sort`). Auf diese Weise kann man ebenso Objekte durch Klonen erzeugen:

```
# mit alternative Paar-Konstruktor
my $arv = $ufo.clone( :speed<2c> );
```

Natürlich kann ein ganzer `%Hash` an Werten bei der Erzeugung übergeben werden. Dabei sieht die Syntax etwas wie Perl 5 aus, da es immer noch eine Methode namens `bless` gibt, die ähnlich dem altem `bless`-Befehl funktioniert.

```
my $klasse = class Heart::Gold { ... }
my %werte = :speed<20c>;
$efv = $klasse.bless( %werte );
```

Eine Kleinigkeit fehlt aber noch: Wie erklärt man eine Methode `protected`, also nicht über ein Objekt aufrufbar? (`$objekt.methode()`) Genau wie man seit Perl 5 Variablen auch in einem Namensraum von außen abschottet, mit `my`.

```
class Heart::Gold {
    my method secret_maneuver { ... }
}
my $ufo = Heart::Gold.new;
# Laufzeiterror:
$ufo.secret_maneuver;
```

Mit `my` lassen sich auch lexikalisch lokale Klassen deklarieren. Die benötigt man zum Beispiel, wenn Attribute selber Objekte sind, deren Klasse geheim bleiben soll. In diesem Beispiel:

```
class Heart::Gold {
    class Movement { ... }
    has int $.length where {$_ >= 0};
    has Movement $drive;
    my method secret_maneuver { ... }
}
```

hat ein Attribut einen vor Ort erstellten Subtyp (siehe letzte Folge), das andere ist ein Objekt (Instanz der Klasse `Movement`). Da diese Klasse nicht mit `my` als lexikalisch lokal deklariert wurde, ließen sich außerhalb von `Heart::Gold` ebenso `Movement`-Objekte erstellen. Auch prozeduraler Zugriff auf den Namensraum `Movement` wäre dann möglich.

Delegier die Arbeit

Eine Klasse `Movement` ist sinnvoll, da solch ein fortschrittliches Fluggerät, sicher sehr viele Eigenschaften (Attribute) und Funktionen (Methoden) besitzt. Sie alle in eine Klasse zu schreiben ginge zu Lasten der Nachvollziehbarkeit. Unser Attribut `$drive`, welches das Antriebsaggregat repräsentiert, ist also ein Objekt, dessen Methoden nun Informationen über Geschwindigkeit und ähnliches herausgeben. Es erscheint als müsste nun doch ein *Getter* `$.speed` geschrieben werden, welcher die Methode `$drive.speed` aufruft und ihr Ergebnis weiterleitet. Aber auch hierfür gibt es eine kurze Schreibweise die genau dies automatisch tut.

```
class Heart::Gold {
    has Movement $drive handles 'speed';
```

Dieses Konzept wird in der OOP-Welt *Delegation* genannt und wird natürlich wie fast alles bekannte auch von Perl 6 angeboten. Nicht umsonst heißt das neue Motto: "all your paradigms are belong to us".

Regelung der Erbschaft

Somit erübrigt sich auch die Frage: "Gibt es Mehrfachvererbung in Perl 6?", doch beginnen wir mit der Einfachen. Klassen vererben einander all ihre Attribute und Methoden. Das nennt man auch "ableiten". So kann eine einmal geschriebene Klasse wiederverwendet und Schritt für Schritt erweitert werden. Besitzt die Elternklasse dabei eine Methode, welche der Sprössling auch definiert, so überschreibt er damit die geerbte Methode. Ich könnte eine einfache Klasse schreiben, die den Flug von Pollen oder einer Pusteblume wiedergibt. Von dieser könnte eine nächste erben und einige Methoden zufügen, welche Manöver eines Heißluftballons beschreiben, usw. bis wir in sechster oder siebenter Generation einen Ufoantrieb haben der auch auf die Zeitverschiebungen bei Interdimensionsreisen eingehen kann.

```
class UfoDrive is AdvancedRocket {
    ...
}
```

Da die "Heart of Gold" bekanntermaßen einen Unwahrscheinlichkeitsantrieb besitzt, der die unwahrscheinlichen Beträge auf den Rechnungen des integrierten Restaurants nutzt, muss noch von einem solchen alles vererbt werden.

```
class UfoDrive is AdvancedRocket
                  is Restaurant { ... }
```

Das ist eine Kurzschreibweise für:

```
class UfoDrive {
    is AdvancedRocket;
    is Restaurant;
    ...
}
```

Es sei anzumerken, dass die Klasse *Restaurant* gleichnamige Methoden der Klasse *AdvancedRocket* "überschreibt". In Java können Klassen als "final" markiert werden. Das ist in Perl 6 nicht direkt vorgesehen, jedoch kann in einem Programm mit dem Pragma `use oo :final;` angewiesen werden, alle Klassen automatisch als "final" zu betrachten. Ausnahmen können mit

```
{
    use class :nonfinal;
    class UfoDrive { ... }
    class Heart::Gold { ... }
}
```

oder

```
class Insect is nonfinal { ... }
```

angegeben werden. Einzelne Methoden, die nicht vererbt werden sollen, können als `submethod` deklariert werden, die ebenfalls mit einem `my` davor nicht über ein Objekt oder den Namensraum aufrufbar sind.

Rollen die ich spiele

Eine Rolle wird definiert wie eine Klasse, nur mit dem Schlüsselwort `role` anstatt `class`.

```
role Blinken {
    has @.lichter;
    method blinken { ... }
}
```

Von einer `role` kann aber kein Objekt erzeugt (geklont) werden. Wozu dient dann eine `role`? Die Praxis hat gezeigt, dass mit Klassen allein der Code nicht immer einfach wieder verwendbar ist. Schnell kam es zu Bäumen von Vererbungshierarchien. Was aber tun, wollten zwei "Äste eines Baums" wenige Methoden gemeinsam verwenden? Selbst Sprachen, die Mehrfachvererbung kennen (nicht Java und C#) erzeugen damit neue Probleme, die mit Rollen vermieden werden können. Sie sind auch syntaktisch natürlicher und bieten sogar mehr Sicherheit ungewollte Methoden zu überlagern als Rubys *Mixins*. Rollen werden während der Kompilierung in Klassen "eingebunden", wenn es die Klasse fordert.

```
class Heart::Gold does Blinken {
```

oder

```
class Heart::Gold {
    does Blinken;
    ...
}
```

Wie zu erahnen, besitzt nun auch die Klasse *Heart::Gold* die Methode *blinken*. Roles können jedoch nicht Methoden einer Klasse überlagern. Versuchen zwei Roles gleichnamige Methoden zu importieren gibt es einen Fehler beim Kompilieren, es sei denn die Klasse hatte eine gleichnamige Methode und der Konflikt muss nicht gelöst werden. Perl 6 kennt Wege wie Roles und Klassen ihre Konflikte selbst lösen können, dies würde aber den Rahmen überdehnen. Was jedoch jeder über Roles wissen sollte, der ihre Verwendung in Betracht zieht, betrifft Attribute. Roles können natürlich eigene Attribute mitbringen. Dies tat die Role im Beispiel mit:

```
has @.lichter; # oder:
has @!lichter; # oder:
my @!lichter;  # für Klasse unsichtbar
```

Sie könnte aber auch Attribute der Klasse mitnutzen. Das meldet sie an mit:

```
has @lichter;
```

und der Kompiler prüft ob die Klasse dieses Attribut besitzt. Ein großer Nutzen der Roles kommt auch von ihrer Fähigkeit sich auch zur Laufzeit einzubinden.

```
    $ufo does Blinken;
```

Mehrere Roles müssen einzeln gebunden oder geklammert werden:

```
    (($ufo does Blinken) does Tarnen);
```

Bei diesen Fähigkeiten, gibt es auch Bedarf zur Laufzeit zu prüfen, ob eine Role in ein Objekt "eingemischt" wurde.

```
if $ufo.does(Blinken) { ... }
```

Was in unserem Falle wahr (Bool::True) ist. Ebenso kann geprüft werden ob ein Objekt von einer Klasse abgeleitet wurde:

```
if $ufo.isa(Heart::Gold) { ... }
```

Dieser Syntax ähnelt sehr Perl 5, Folgendes ist jedoch neu, auch wenn Perl 5.10 bereits smartmatch kennt:

```
if $ufo ~~ Blinken     { ... }
if $ufo ~~ Heart::Gold { ... }
```

Beide Fälle sind genauso wahr, da `~~` eine große Tabelle hat und je nach Typ der Parameter versucht das "Richtige" zu tun. Solches Verhalten erreicht man durch `multi`-Methoden (in voriger Folge vorgestellt). Wenn *Roles Multimethoden* importieren, dürfen diese gleichnamig sein, solange die Signaturen nicht gleich sind.

Zeig mir dein Inneres

Objekte lassen sich in Perl 6 weitaus tiefer durchleuchten als es das Smartmatch tut. `$ufo.HOW` oder `^Heart::Gold` erlaubt z.B. den Zugriff auf die Metaklasse und `Heart::Gold.^methods()` liefert beispielsweise alle Methoden der Klasse. Doch an der Stelle möchte ich mich für dieses mal verabschieden, da die Methoden zur Introspektion (hoffentlich bald vollständig) im Tutorial der Perl-Community.de Wiki aufgelistet sind und ein Aufzählen einer API langweilig ist. Diese API ist, im Gegensatz zu der in dieser Folge vorgestellten Syntax, keinesfalls stabil und endgültig, sodass ein Blick in den tabellarischen Index B des Tutorials sinnvoll ist. Die nächste Folge wird behandeln, was Perl einst berühmt und berüchtigt machte: die regulären Ausdrücke die jetzt wirklich regulär sind, sich nicht mehr Ausdruck nennen und auch wie Objekte behandelt werden können.

Herbert Breunung

Foswiki: Die frühere TWiki-Community geht neue Wege

Das in Perl programmierte Wissensmanagement-System TWiki im Unternehmenseinsatz war bereits Gegenstand eines Artikels im Perl-Magazin. Unbestritten ist das perl-basierte TWiki nach wie vor eines der ausgereiftesten Corporate-Wiki-Systeme.

Im Oktober 2008 ist die Wiki-Landschaft allerdings in Bewegung geraten: Die Community hat sich geschlossen aus dem TWiki-Projekt zurückgezogen und entwickelt seitdem im Alleingang eine eigene Wiki-Software auf Basis von TWiki. Abgesehen vom Gründer sind wirklich alle Community-Teilnehmer gewechselt (siehe http://cli.gs/JTeUGt).

Zunächst unter dem Arbeitstitel NextWiki geführt, hat das System inzwischen einen offiziellen Namen erhalten: Foswiki (Free and Open Source Wiki). Die Applikation unterliegt der GNU-Lizenz für freie Software.

Hintergründe zum TWiki-Fork

Das TWiki-Projekt wurde von Peter Thoeny aus der Taufe gehoben und in seiner Zeit als Mitarbeiter von Wind River vorangetrieben. 2007 gründete er dann mit Venture-Kapital die Firma TWIKI.NET und sorgte für die erste Verstimmung in der Community. Für die Entwicklung des Systems zeichnete sich jedoch nahezu gänzlich eine freie und engagierte Community aus Programmierern, Designern und Projektmanagern verantwortlich.

Die Fronten zwischen der Gemeinschaft und dem Unternehmen, das auch die Markenrechte an TWiki innehat, waren allerdings seit längerer Zeit verhärtet: Streitpunkte waren die zunehmende Kommerzialisierung des Projekts, die durch TWIKI.NET vorangetrieben wurde, und das belastete Vertrauensverhältnis zwischen beiden Parteien. Darüber hinaus wurde der Community nie eine verbindliche Genehmigung zur Markennutzung erteilt.

Als TWIKI.NET die Projektleitung im Oktober 2008 schließlich gänzlich an sich zog und die Community mittels zeitweilig gesperrten Zugriffs auf den Code und das Projektwiki zur Anerkennung eines neuen Führungsmodells zwingen wollte, entschloss sich die Gemeinschaft inklusive des Kernentwickler-Teams dazu, TWiki den Rücken zu kehren: Das Vorgehen des Unternehmens sei als feindliche Übernahme zu werten und bedeute das Ende des Open-Source-Charakters von TWiki, so die einhellige Meinung.

Erwartungen an das neue Projekt

Anstatt das Handtuch zu werfen hat die Community in Eigenregie ein neues ambitioniertes Projekt aus der Taufe gehoben, das auch eine Rückbesinnung auf den reinen Open-Source-Charakter und damit auf die Wurzeln von TWiki darstellt. Der TWiki-Fork hat breite Zustimmung und Unterstützung innerhalb der wiki-affinen Web-Gemeinde und auch bei Unternehmen gefunden, die TWiki nutzen – Anreiz genug für die Community, große Anstrengungen zu unternehmen, um kurzfristig geeignete infrastrukturelle Voraussetzungen für das Foswiki-Projekt zu schaffen.

Die Community knüpft dort an, wo sie aufgehört hat, und entwickelt Foswiki auf Basis von TWiki. Mit Foswiki ist also ein systemorientiertes, strukturiertes Wiki-System zu erwarten, dessen Parsing Engine in Perl programmiert ist. Wie TWiki wird Foswiki ohne eigene Datenbank auskommen und sich durch einen entsprechend schlanken Kern auszeichnen, zugleich aber die Vorteile einer Wiki-Anwendung mit denen

einer Datenbank vereinen und das unkomplizierte Abbilden von Content in strukturierter Form ermöglichen. Hinsichtlich der Organisation von Inhalten behält Foswiki die bewährte Strukturierung in Webs, Topics und Attachments bei.

Es wird leicht sein, von TWiki auf Foswiki zu wechseln. Darüber hinaus sind Funktionen vorhanden, die die einfache Migration von Inhalten auch aus anderen Wiki-Systemen gestatten.

Der Ausblick auf Foswiki

Das Hauptaugenmerk der Community hat zunächst auf dem Rebranding des Systems inklusive der Plugins und der Dokumentation gelegen, um markenrechtlichen Problemen mit TWIKI.NET vorzubeugen. Diese Maßnahmen sind inzwischen weitgehend abgeschlossen.

Viele Entwickler haben im Rahmen der Querelen der letzten Monate mit Peter Thoeny eigenen Code und fertige Module zurückgehalten und nicht für die TWiki-Community bereitgestellt. Viele haben nun angekündigt, diese Erweiterungen der Community zur Verfügung zu stellen. Wenn das tatsächlich passiert, können sich alle bisherigen TWiki- und künftigen Foswiki-Nutzer auf ein richtiges Feuerwerk an neuen Funktionen freuen.

Das sogenannte TWikiCompatibilityPlugin sorgt dafür, dass auch TWiki-Nutzer von Foswiki-Plugins profitieren können und umgekehrt. Unternehmen, die TWiki einsetzen oder den Einsatz planen, sollten sich jedoch mit dem Namen Foswiki vertraut machen und langfristig darauf einstellen, auf das neue System zu migrieren. Wenn die Community sich nicht selbst im Weg steht, sondern wie in den letzten Wochen und Monaten von Foswiki voranschreitet, dann ist es sehr wahrscheinlich, dass Foswiki TWiki sowohl hinsichtlich der Nutzerzahlen als auch in Sachen Bekanntheit ein- und überholt.

Die größten Schwächen des Systems liegen derzeit noch in der Usability der Oberfläche und in der Skalierbarkeit für sehr große Instanzen mit mehr als 10.000 Benutzern und mehr als 100.000 Dokumenten. Für letzteres hat Foswiki bereits eine Lösung mit dem DistributedServersPlugin (siehe http://cli.gs/nQGmbo) und einem Caching-Modul geschaffen. Für die Usability fühlen sich bereits viele Community-Mitglieder inklusive des Autors verantwortlich.

Unternehmen dürfen gespannt auf Foswiki sein und sich auf ein System freuen, das die vorhandene Leistungsfähigkeit von TWiki abbildet und gleichzeitig deutlich schneller und intensiver weiterentwickelt wird. Dies ist vor allem der Tatsache zu verdanken, dass das motivierte und ambitionierte Team nicht länger in politische und Richtungsstreitigkeiten verwickelt ist, sondern seine Produktivität tatsächlich entfalten und sich nunmehr auf das Wesentliche konzentrieren kann: Foswiki.

Wenn Sie die Foswiki-Community mit Know-how oder finanziell unterstützen möchten, finden Sie auf der Website http://foswiki.org umfassende Informationen über Möglichkeiten, sich aktiv oder passiv an diesem Projekt zu beteiligen. Siehe auch: http://foswiki.org/Community/HowYouCanHelp. Und wer einfach nur grüßen und "Alles Gute!" wünschen will, kann den Projektbeteiligten mit einer Mail an foswiki-cheer-and-donate@lists.sourceforge.net eine kleine Freude machen.

Martin Seibert

F*EX: Dateitransfer für Große Jungs [TM]

Hier an der Universität Stuttgart müssen wir häufig sehr große Dateien zwischen Benutzern austauschen. Die Dateien enthalten beispielsweise Datensätze aus wissenschaftlichen Bereichen wie Strömungsmechanik, Multimedia-Daten, DVD- oder BluRay-Images oder komplette virtuelle Maschinen. Die Dateigrößen bewegen sich dabei vom zweistelligen Megabyte- bis in den Terabyte-Bereich.

Wir brauchten daher einen Dateitransfer-Dienst, der folgende Anforderungen erfüllen muss:

- individuell: von Benutzer A an Benutzer B (irgendwo im Internet)
- groß: Dateien im GB oder gar TB Bereich
- robust: Wiederaufsetzen nach Verbindungsabbruch
- verfügbar: keine Probleme mit Firewalls
- universell: Betriebssystem-unabhängig, keine extra Software-Installationen
- asynchron: Empfänger B muss nicht zeitgleich aktiv werden

Auf der Suche nach dem geeigneten Lösungen für den Dateitransfer haben wir schon existierende Dienste betrachtet und dabei diverse Probleme festgestellt:

• Medien-Transport per Post
Kommt nicht in Frage! Wir leben im Jahr 40 nach Erfindung des Internets! Schnelle Datenleitungen existieren seit vielen Jahren und Kurierdienste sind was für Großväter!

• E-Mail
Ursprünglich war E-Mail nur für Text gedacht und nicht für Binärdaten geeignet. Es wurde zwar später durch die MIME-Spezifikation dahingehend aufgebohrt, aber es gibt kaum Mail-Programme, die effizient mit großen Mails umgehen können: Die Limitierung der meisten Mailserver liegt um die 10 MB pro E-Mail und wenige 100 MB Gesamtspeicherplatz pro Benutzer. Viele Mail-Clients hängen sich allerdings bei größeren Mails auf und blockieren damit die Mailbox komplett.

• FTP
Für den Download zwar gut geeignet, aber für den Upload braucht es spezielle Software, die nicht überall vorinstalliert ist. Die Verwaltung der Zugriffsberechtigungen ist ein Alptraum, Verschlüsselung gibt es nur in Verbindung mit SSH (SFTP), was die Sache nochmal verkompliziert, und die meisten Clients haben ein 2-GB-Limit. Zudem muss der Empfänger den Sender informieren, damit er das Zwischenarchiv nach dem Download wieder löschen kann. Wenn Sender und Empfänger Firewalls einsetzen, geht ftp oft gar nicht.

• SSH/SCP
Ist zwar verschlüsselt, aber dafür braucht der Sender das Passwort vom Empfänger oder umgekehrt. Zudem darf keine Firewall den zugehörigen TCP Port 22 blockieren. Aber wer will schon sein persönliches Passwort weitergeben nur für einen simplen Dateitransfer?

• HTTP
Funktioniert Out-of-the-Box nur für anonymen Download, für Upload und authentifizierten Download werden Zusatzprogramme benötigt, da ein User-to-User-File-Transfer vom Protokoll direkt nicht vorgesehen ist. Der Sender benötigt einen Account auf dem Webserver und muss sich um das Aufräumen hinterher kümmern.

• kommerzielle Web-Dienste
Auf HTTP-Basis gibt es zwar einige kommerzielle Web-Dienste wie DropLoad, ALLPeers oder YouSendIt die die oben angesprochene Probleme zumindest teilweise angehen, dafür aber neue aufwerfen:

- meistens sind sie auf 2 GB Dateien limitiert oder gar weit drunter
- sind die Daten dort sicher und vertraulich aufgehoben?
- das ganze System ist proprietär-undurchsichtig, kein Open Source
- der Zugriff erfordert Flash, Java-Gedöhns oder mysteriöse Plugins
- werden die Dienste und damit die dort gespeicherten Daten länger als ein paar Monate existieren (DropLoad und ALLPeers haben inzwischen ihr Geschäft eingestellt)?

• SAFT/sendfile

Das Protokoll SAFT [1] erfüllt zwar die meisten unserer Anforderungen, allerdings benötigt es auf Sender- wie Empfängerseite spezielle Software (sendfile), die nur für UNIX existiert und der TCP Port 487 muss vom Sender zum Empfänger offen sein. Für viele Anwender sind das unüberwindliche Hürden.

Der Quasi-Vorgänger des Internets im wissenschaftlichen Bereich, das BITNET, hatte so einen personalisierten asynchronen Dateitransfer-Dienst. Genauer gesagt, war es DIE zentrale Komponente, auf der alle anderen Dienste aufsetzten. Daher war es für viele BITNET-Internet-Konvertiten unbegreiflich, dass es so etwas im Internet nicht gab. Ein typischer Fall von "not invented here".

Bedarf für so einen Dienst gibt es reichlich, also musste der Autor eben selbst tätig werden. Dabei wurde darauf geachtet, dass die oben genannten Anforderungen eingehalten wurden. Als Weiterentwicklung von SAFT und unter Vermeidung dessen Defizite entstand F*EX: Frams' Fast File EXchange, ein Internet-Dienst für den persönlichen und asynchronen Transfer beliebig großer Dateien zwischen registrierten Sendern und beliebigen Empfängern.

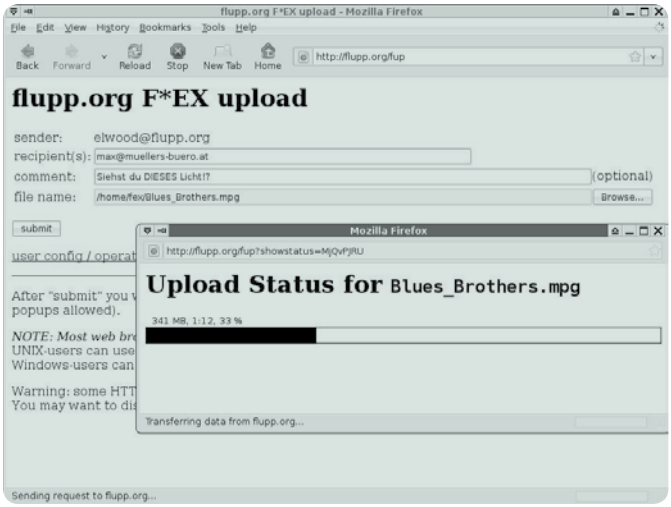


Abbildung 1: Upload mit Firefox als F*EX-Client

Pro Organisation bzw. Installation gibt es einen zentralen F*EX-Server zur Zwischenablagerung der Dateien, den sogenannten F*EX-Spool, vergleichbar einem Mail-Server. Nur eben für (große) Dateien, statt für (viele) kleine E-Mails. An der Universität Stuttgart gibt es z.B. 3 F*EX-Server, die von unterschiedlichen Instituten betrieben werden.

Zum Ablauf:

Der Administrator legt mit dem Programm fac den F*EX-Useraccount lokal an (falls er nicht LDAP- oder Radius-Authentifizierung einsetzt) und teilt die Login-Daten (E-Mail-Adresse und auth-ID) dem neuen Benutzer mit.

Die F*EX-Endanwender brauchen nur einen Client und eine E-Mail-Adresse. Dieser Client kann ein beliebiger Webbrowser sein oder ein spezieller F*EX-Client.

Mit dem F*EX-Client wird die Datei vom Sender auf den F*EX-Server übertragen. Der Sender gibt dabei auch den Adressaten der Datei und optional noch einen Begleitkommentar mit an.

Nach dem Upload schickt der F*EX-Server automatisch eine Benachrichtigungs-E-Mail an den Empfänger, in der eine URL steht, unter der dieser die Datei herunterladen kann.

Nach dem erfolgreichen Download löscht der F*EX-Server die Datei automatisch. Eine automatische Löschung erfolgt ebenfalls nach Überschreiten der eingestellten Speicherzeit (Expire). Es bleibt also kein Datenmüll zurück.

Leider gibt es keinen einzigen Webbrowser, der mehr als 4 GB Upload (HTTP POST) beherrscht, die meisten brechen sogar schon nach nur 2 GB mit unverständlichen Fehlermeldungen ab. Deshalb gibt es den F*EX-GUI-Client schwuppdiwupp (in Perl/Tk) und die Kommandozeilen-Clients fexsend und fexget. Diese speziellen F*EX-Clients kennen keine Größenbeschränkungen. Mal schnell ein paar Terabyte verschicken? Das geht eben schwuppdiwupp - wenn die Hardware-Infrastruktur mitmacht :-)

Der Download dagegen klappt mit gewöhnlichen Webbrowsern ohne Beschränkung.

Sollte der Up- oder Download unterbrochen werden, besteht die Möglichkeit des Wiederaufsetzens. D.h. der Client [3] kann beim zuletzt übertragenen Byte weitermachen und muss nicht wieder von vorne anfangen.

Der Empfänger erhält eine Benachrichtigung per E-Mail sobald für ihn eine neue Datei zur Abholung bereit liegt. Für die Datei wird ein Ablaufdatum festgelegt bis zu diesem der Empfänger die Datei abgeholt haben muss. Danach wird sie automatisch gelöscht. Die Datei wird auch gelöscht, sobald der Empfänger sie heruntergeladen hat.

Beim Upload können auch mehrere Empfänger angegeben werden. Die Datei wird dabei nur einmal übertragen und verbraucht nur einmal Speicherplatz auf dem Server. Trotzdem bekommt jeder Empfänger eine persönliche Datei angeboten, die unabhängig von den anderen ist (realisiert wird das über UNIX hard links).

F*EX ist rein HTTP (TCP Port 80) bzw. HTTPS (TCP Port 443) basiert, daher gibt es keine Probleme mit Firewalls.

Da F*EX kein anonymer Dienst ist, muss der Sender registriert sein. Für lokale Benutzer lässt sich die Registrierung noch automatisieren, aber diese wollen ja auch Dateien von außen empfangen können. Deshalb können registrierte F*EX-Full-User ohne besondere Admin-Rechte selbst "Sub-User" anlegen, die nur ihnen etwas schicken können, aber sonst niemandem. Somit ist für F*EX ein weitestgehend automatischer Betrieb möglich, der Administrations-Aufwand ist minimal.

Sowohl der Server als auch die Clients sind in Perl geschrieben, wobei darauf geachtet wurde nur Module zu verwenden,

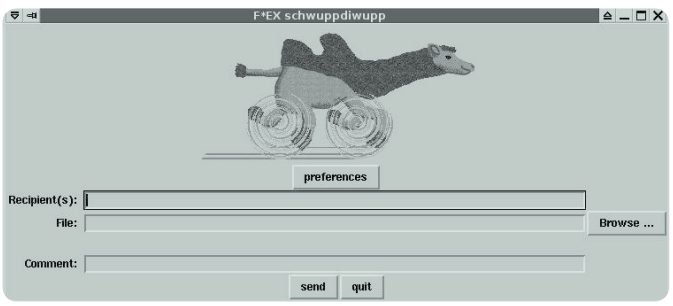


Abbildung 2: der F*EX-Client schwuppdiwupp

die zum Standard-Perl gehören (Ausnahme: SSL). Die Clients laufen sowohl unter UNIX als auch Windows, der Server dagegen benötigt zwingend ein UNIX System.

Implementation

Die erste F*EX Version verwendete `CGI.pm` und einen Apache Webserver. Es stellten sich damit aber sehr schnell Probleme heraus: beide kommen mit großen Uploads schlecht bis gar nicht zurecht. Wegen ineffektivem und nicht abschaltbarem Caching dauerte das Abspeichern ab etwa 100 MB länger als die eigentliche Übertragung, was viele Benutzer stark irritierte: *"Nanu, die Datei ist doch längst übertragen? Hat sich der Server aufgehängt? Ich brech' dann mal ab..."*

Also wurde zuerst auf `CGI.pm` verzichtet und die davon benötigte Funktionalität selbst programmiert, was schon mal einen deutlichen Geschwindigkeitsgewinn erbrachte. Leider stellte sich dann weiter heraus, dass der Apache bei der HTTP POST Verarbeitung ein 2 GB Problem hat: das ist ein hartes Limit, das auch in der 64-bit Variante besteht. Dabei soll der F*EX-Server auch auf 32-bit Systemen laufen.

Alle daraufhin untersuchten alternativen Webserver zeigten ähnliche Probleme beim Umgang mit großen Uploads. So etwas war wohl nie getestet worden. Webserver sind eben hauptsächlich für die Ausgabe von Dateien optimiert, aber nicht für die Annahme.

Also musste auch noch ein eigener Webserver geschrieben werden, was ursprünglich gar nicht geplant war. Im Gegenteil: wenn das von vornherein festgestanden hätte, wäre das Projekt wohl erst gar nicht angegangen worden. Nachdem aber die F*EX CGIs und die Clients schon fertig waren und der erste Beta-Test bei den Benutzern überaus erfolgreich ankam, wurde der Beschluss gefasst:

"Dann programmier' ich halt auch noch einen eigenen Webserver"

Da nur die Funktionalität implementiert werden musste, die F*EX für den Betrieb benötigte und alles andere, was der HTTP-Standard sonst noch spezifiziert, weggelassen werden konnte, entstand dabei mit fexsrv ein Webserver in nicht mal 10 kB Sourcecode, der SEHR effizient und schnell mit wirk-

lich großen Dateien umgehen kann. Faktor 10 Geschwindigkeitsvorteil gegenüber einem apache sind erst der Anfang.

Da jetzt nicht mehr auf Einhaltung der CGI-Spezifikation geachtet werden musste, konnte ohne Kompromisse auf Effizienz und Geschwindigkeit optimiert werden. Abgesehen vom Plattenplatzverbrauch (was in der Natur der Sache liegt: große Dateien sind nunmal groß) ist der Ressourcenverbrauch an CPU und RAM minimal.

Der F*EX-Server ist stark an das klassische Webserver-Design angelehnt: der eigentliche Server fexsrv nimmt HTTP Requests an und ruft die zuständigen Server-Programme (CGIs) auf:

- fup : File UPload - Datei Annahme
- fop : File OutPut - Datei Ausgabe
- foc : F*EX Operation Control - Zusatzdienste
- fuc : F*EX User Control - Konfiguration durch den Benutzer
- fur : F*EX User Registration - Benutzer Neuanmeldung
- rup : Redirect UPload - Datei Um- bzw Weiterleitung
- sex : Stream EXchange - siehe Teil 2 dieses Artikels :-)

Außerdem kann fexsrv Dokumente (HTML, JPEG, PDF, etc.) wie ein normaler Webserver selber ohne CGI Hilfsprogramme ausgeben.

Die Clients sind:

- fexsend : UNIX CLI Client zum Versenden von Dateien
- fexget : UNIX CLI Client zum Abholen von Dateien
- schwuppdiwupp : UNIX und Windows GUI Client zum Versenden von Dateien
- sexsend : UNIX CLI Client zum Versenden von Streams
- sexget : UNIX CLI Client zum Abholen von Streams

Zusätzlich gib es noch das Hilfsprogramm fac (F*EX Admin Control) zur Benutzerverwaltung.
Auf UNIX Shell Ebene sieht das Versenden so aus:

```
framstag@moep:/tmp: fexsend vmware.tar
    hoppel@flupp.org
sending vmware.tar to hoppel@flupp.org
transfered: 5346 MB in 467 s with 11448 kB/s
```

Der Empfänger bekommt dann diese E-Mail - siehe Listing 1.

Server und Clients zusammen umfassen gerade mal 150 kB Sourcecode und sind frei verfügbar (GPL Open Source) unter http://fex.rus.uni-stuttgart.de/ (Hinweis: hier bedient Sie ein fexsrv!)

Ulli Horlacher
framstag@rus.uni-stuttgart.de

[1] Simple Asynchronous File Transfer
http://de.wikipedia.org/wiki/
Simple_Asynchronous_File_Transfer

[2] Because It's There Net
http://www.livinginternet.com/u/ui_bitnet.htm

(3] Webbrowser beherrschen "resume" nur beim Download, jedoch nicht beim Upload.

```
From: framstag@rus.uni-stuttgart.de via F*EX service fex.rus.uni-stuttgart.de
 To: hoppel@flupp.org
 Subject: F*EX-upload: vmware.tar

 framstag@rus.uni-stuttgart.de has uploaded the file
   "vmware.tar"
 (5346 MB) for you. Use the link

 http://fex.rus.uni-stuttgart.de/fop/3TyYH6YX/vmware.tar

 to download this file within 5 days.

 WARNING: After download (or view with a web browser!), the file will be deleted!

 Remember: F*EX is not an archive, it is a spooling system.

 Questions? ==> F*EX admin: fexmaster@rus.uni-stuttgart.de
```

Listing 1

ANWENDUNG

Ein Videomixer in Perl

Videoinstallationen faszinieren mich. Bilder von bekannten Abfolgen, die sich augenscheinlich immer wiederholen, aber nicht immer gleich sind. Ich möchte Videos aus Versatzstücken zusammensetzen und verfremden, so dass die zugrundeliegende Struktur nicht gleich sichtbar ist. Eine profanere Anwendung derselben Technik ermöglicht das Einblenden eines Logos in bestehende Videos, der Zusammenschnitt mit Übergängen und nicht zuletzt die Ausgabe des Ergebnisses auf den Bildschirm meines Computers.

Perl mag nicht als erste Wahl zur Bearbeitung von Videoquellen erscheinen, diese gilt als Domäne von maschinennahen Sprachen wie C. Durch die Fortschritte in der Grafikhardware und die Module *OpenGL* und *OpenGL::Shader* steht aber auch Perl ein mächtiges Rechenwerk zur Verarbeitung von Videodaten in Form der Grafikkarte zur Verfügung.

Warum OpenGL?

Das einfache Zusammenschneiden von Videodaten kann bereits das Programm `ffmpeg` [1]. Das Programm bietet aber keine Effekte zur Nachbearbeitung wie Verfremdung oder Zusammenmischung der Daten an. Auch eine Ausgabe auf den Bildschirm wird nicht "live" unterstützt.

Mit OpenGL steht eine Schnittstelle zur Verfügung, die ursprünglich rein zur Darstellung von Daten im dreidimensionalen Raum gedacht war. Es lassen sich mit dieser Schnittstelle aber auch zweidimensionale Daten darstellen, wie es die Videodaten sind. Insbesondere das einfache Mischen von mehreren Videoquellen kann man sich vorstellen als die Überlagerung von Filmen durch mehrere Projektoren auf dieselbe Leinwand (siehe Abbildung 1).

Derselbe Effekt ließe sich auch in Perl direkt programmieren, aber dann unter Verzicht auf die Unterstützung durch die brachliegende Hardware. Die Leistungsfähigkeit der spezialisierten Hardware und von OpenGL übersteigt die Leistung von Perl und die Fähigkeiten des Autors um ein Vielfaches.

Funktionen von OpenGL

OpenGL in den Versionen vor 2.0 unterstützte nur wenige, vorgegebene Funktionen zur Kombination von Bildern. Diese Operationen beschränken sich im Wesentlichen auf die auch in Perl bekannten Operatoren. Zwei Punkte aus zwei Bildern können im Zielbild addiert, multipliziert oder subtrahiert werden. Alternativ kann der Wert im Zielbild als bitweise "oder" (`OR`), "und" (`AND`) bzw. "exklusiv-oder" (`XOR`) Verknüpfung der Ursprungspunkte ermittelt werden.

Mit diesen festen Funktionen lässt sich immerhin bereits eine Maskierung für die Einblendung eines Logos durch eine Kombination von `AND` und `OR` erreichen. Mit `AND` wird der Bereich des Logos aus dem Film ausgestanzt und dann mit `OR` der ausgestanzte Bereich mit dem Logo überschrieben. Ein

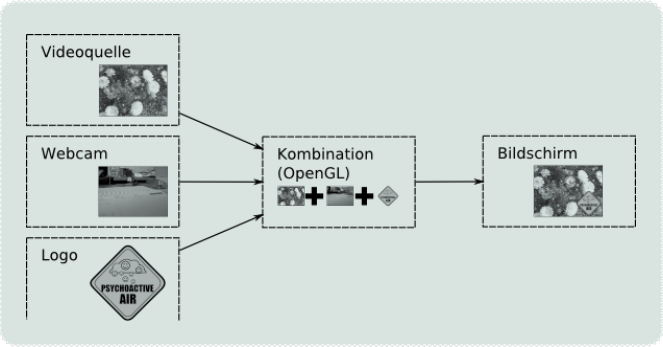


Abbildung 1: Kombination von Bildern

Programm, welches unter anderem eine PNG-Datei als Logo in einen Filmclip einbindet, findet sich, wie aller Quellcode zu diesem Artikel, in der Distribution *App::VideoMixer* auf dem CPAN.

OpenGL Shader und GLSL

Seit OpenGL 2.0 (und jetzt OpenGL 3.0) kann man die Funktionen zur Verarbeitung von Bilddaten selber programmieren. Diese Programme sind in einer eigenen Sprache ("GLSL") verfasst, und heißen "Shader". GLSL ist der Programmiersprache `C` nachempfunden, macht aber zugunsten der parallelen Verarbeitung der Bildpunkte einige Einschränkungen. In dieser Sprache lassen sich sehr interessante Verfremdungs- und Effektfilter programmieren, deren Parameter sich wiederum von Perl aus steuern lassen. Zum Beispiel lassen sich damit Filter zur Kantenerkennung, Green Screen bzw. Chroma-Key Filter und Überblendungen programmieren, die ohne weitere Belastung der CPU für die Ausgabe aktiviert werden können.

Programmstruktur

Perl funktioniert gut als Verbindungskleber zwischen den spezialisierten Werkzeugen `ffmpeg` und OpenGL. Das Programm übernimmt den Datentransport zwischen Hauptspeicher und Grafikkarte und die Steuerung der Parameter der einzelnen Filter.

Die Daten werden durch das Programm `ffmpeg` dekomprimiert und an Perl übergeben. Hier besteht eine erste Möglichkeit, die Daten im Zugriffsbereich von Perl zwischenzuspeichern, bevor sie an eine Reihe von OpenGL-Shadern übergeben werden. Die OpenGL-Shader kombinieren und verfremden die Daten und stellen sie zu guter letzt auf dem Bildschirm dar. Das Ergebnis oder auch Zwischenergebnisse können auch wieder an Perl zurückgeliefert werden und von dort mittels `ffmpeg` als Datei gespeichert werden (siehe Abbildung 2).

Beispielfilter

Als Anwendungsmöglichkeiten für die Verfremdung von Filmclips werden im Folgenden drei Filter kurz vorgestellt. Diese Filter arbeiten auf jeweils verschiedenen Ebenen, ausschließlich auf der Grafikhardware, ausschließlich in Perl oder in einer Mischung aus beidem.

Filter "Delay"

Der Filter "Delay" ist ein Beispiel für einen in Perl implementierten Filter. Der Filter speichert 20 Bilder zwischen und

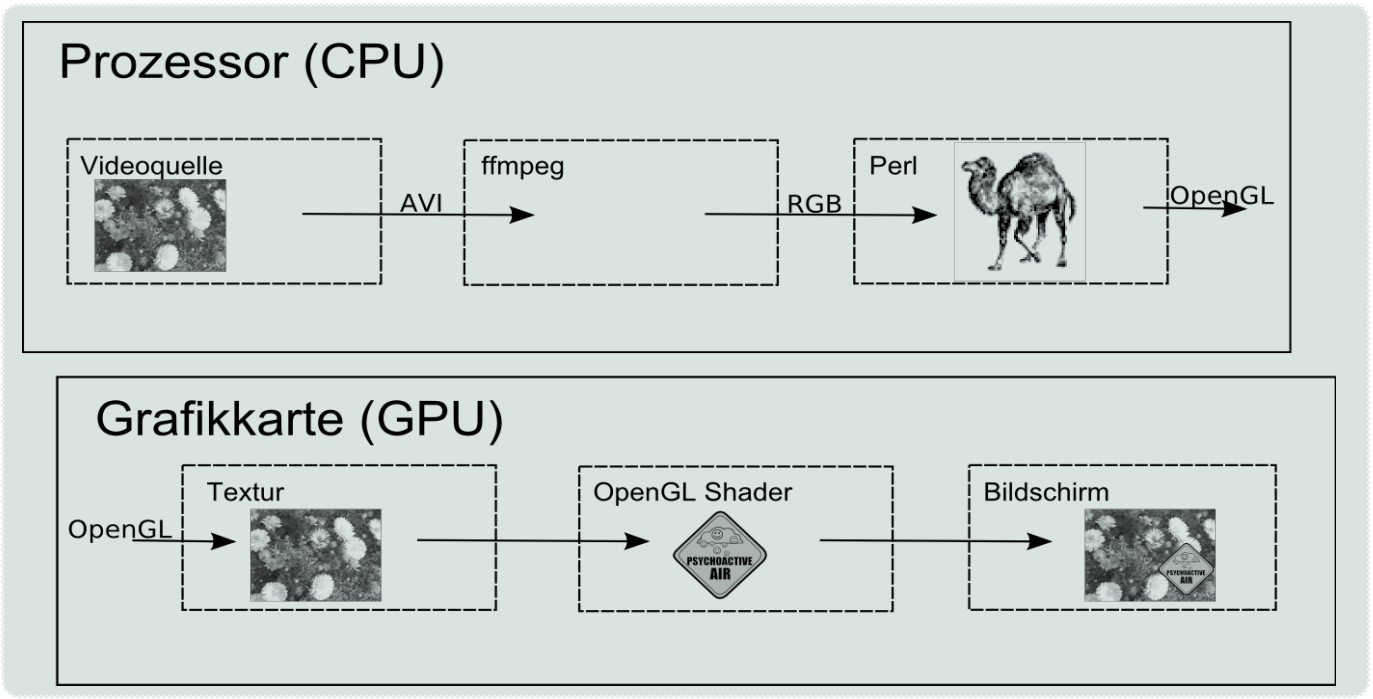


Abbildung 2: Datenfluss zwischen ffmpeg, Perl und Grafikkarte

bewirkt dadurch eine Verzögerung der Ausgabe. Das Kopieren der Daten ist schnell, wenn sie aus `ffmpeg` gelesen werden und verhältnismäßig langsam, wenn die Daten von der Grafikkarte gelesen werden. Die Grafikkarte ist optimiert für schnellen Schreibzugriff. Leseoperationen sind deutlich langsamer (siehe Abbildung 3).

Filter "Nervös"

Ein weiterer Filter, der einen etwas beunruhigenden Effekt der Ausgabe erwirkt, ist der Filter "Nervös". Dieser Filter mischt die Reihenfolge der Bilder. Dadurch wirken Bewegungen sehr ruckhaft und hektisch.

Halbton Filter

Der "Halbton" Filter verfremdet die Eingabe ähnlich dem Verfahren, wie es in der Druckreproduktion verwendet wird. Das Bild wird grob gerastert. Die Helligkeit eines Bildpunktes aus dem Raster wird durch dessen Größe abgebildet. Die Farbe eines Bildpunktes bleibt unverändert. Dadurch wirkt das Bild aus der Ferne unverändert, aus der Nähe verschwindet das Motiv in den separaten Punkten.

Das Programm

Das Programm verteilt die Arbeit auf mehrere Module [4]. Alle Module sind über das CPAN installierbar. Entwickelt wurde der Code mit Strawberry Perl 5.10 [1] und mit dem Augenmerk auf Windows. Andere Betriebssysteme sollten mit geringen Anpassungen ebenfalls unterstützt werden können. Das Modul Imager ist nicht trivial über CPAN zu installieren, da es weitere Bibliotheken benötigt, die nicht auf CPAN liegen (`zlib`, `libjpeg`, `libpng`). Hier empfiehlt sich die Installation eines vorkompilierten Pakets mittels `PPM`.

- `OpenGL` - OpenGL Anbindung (CPAN)
- `OpenGL::Shader` - OpenGL Shader/Filter (CPAN)
- `Imager` - Bilder laden (CPAN)
- `App::VideoMixer` - das Hauptprogramm (CPAN)
- `App::VideoMixer::Source::FFmpeg` - der Decoder für die Videostreams (CPAN)
- `App::VideoMixer::Filter::Buffer` - Ringpuffer für Feedback/Delay (CPAN)

Das Hauptprogramm

Der Ablauf des Hauptprogramms ist schlicht. Die einzelnen Filme werden zum Lesen mit *Source::FFmpeg* geöffnet und die GLSL-Shader eingelesen und kompiliert. Danach tritt das Programm in eine Schleife ein, die alle Bilder aus allen Filmen verarbeitet.

Die einzelnen Filter sind Funktionen, die ein oder mehrere Bilder ("Texturen") als Eingabe annehmen und wiederum ein Bild als Ergebnis liefern. Die Ausgabetextur eines Filters wird als Eingangstextur für den nächsten Filter weitergereicht, so dass alle Filter hintereinander das Bild verfremden. Da der Bildspeicher auf der Hardware begrenzt ist, arbeitet die Filterkette auf zwei Speichern, die im Wechsel als Quellspeicher und Ergebnisspeicher dienen.

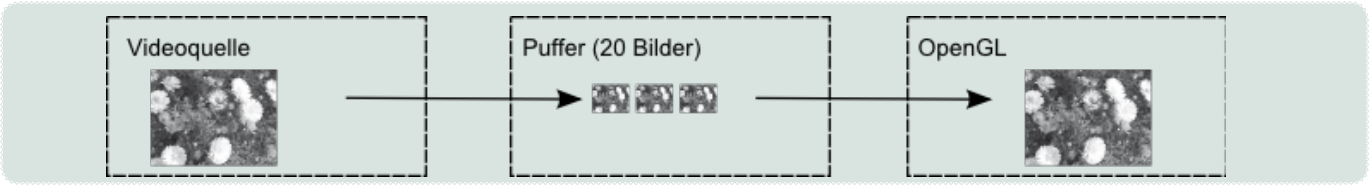


Abbildung 3: Filter "Delay"

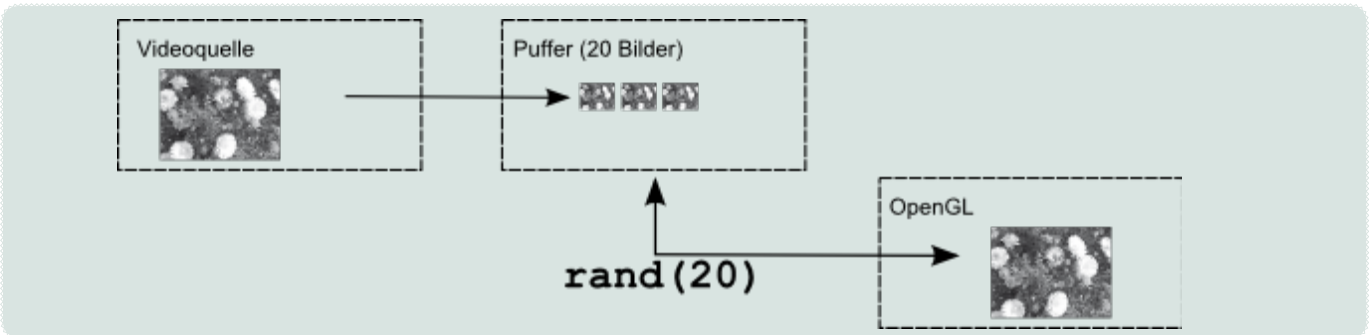


Abbildung 4: Filter "Nervös"

```
my $curr_texture;
for my $movie (@movies) {
    # Bild in Grafikkarte laden
    $curr_texture = $movie->tick();
    # Bild in Ringpuffer laden
    ...

    # Textur-Ping-Pong
    for my $filter (@active_filters) {
        $curr_texture =
            $filter->render($texture);
    };
};
```

App::VideoMixer::Source::FFmpeg

Das Modul *App::VideoMixer::Source::FFmpeg* kapselt die Steuerung des Programms `ffmpeg`. Zwar gibt es die Bibliothek `libffmpeg` auch als Perl-Kapselung auf dem CPAN (*FFmpeg*), aber das Kompilieren dieser Bibliothek ist aufwendig, da viele externe Bibliotheken benötigt werden.

Der Code an sich soll hier nicht näher diskutiert werden. Das zentrale Ergebnis von `App::VideoMixer::Source::FFmpeg` ist ein Dateihandle, aus dem die einzelnen Bilder als "Zeilen" gelesen werden.

Vor- und Nachteile von FFmpeg+OpenGL

Die Kombination von OpenGL und `ffmpeg` kann keine Soundausgabe. Durch das Auslesen über eine Pipe ist kein Zurückspulen des Films möglich. Ein Vorspulen ist nur durch sehr schnelles Lesen aus der Pipe möglich. Die Verwendung von OpenGL schliesst den Einsatz in einer Multiuserumgebung aus. Genauso ist der Einsatz in Servermaschinen aufgrund der fehlenden Hardware(beschleunigung) ausgeschlossen.

Die Verwendung von `ffmpeg` hat aber den Vorteil, dass keine Kompilierung der `ffmpeg`-Bibliotheken notwendig ist. Es reicht, eine ausführbare Version von `ffmpeg` zu finden.

OpenGL ist ebenfalls leicht zu installieren, sofern ein zu Perl passender C Compiler vorhanden ist.

Links zum Artikel

[1] ffmpeg Homepage: *http://ffmpeg.mplayerhq.hu*
[2] Strawberry Perl *http://strawberryperl.com*
[3] Imager: *http://cpan.uwinnipeg.ca/dist/Imager*
[4] Quellcode: *http://search.cpan.org/dist/App-VideoMixer/*

Max Maischein

PERL'S TRY-CATCH

Mangelnde Fehlerbehandlung ist bei Perl häufig ein Problem. Immer wieder vermissen Programmierer Konstrukte wie das try-catch bei Java. Bei Perl wird dann stattdessen das Block-`eval` empfohlen. Das ist zwar nicht so komfortabel wie Java's try-catch, aber es ist ganz nützlich. Bei einem Block-`eval` steht normaler Perl-Code im Block, der an `eval` übergeben wird. Tritt ein Fehler innerhalb dieses Codes auf, dann bricht das Perl-Programm nicht gleich ab, sondern läuft weiter. Es gibt dabei aber kein "Rollback", das heißt: werden Variablenzuweisungen gemacht, dann werden die alten Zustände nicht wieder hergestellt.

```
$ perl -le 'my $var = "test";
eval{ $var = "hallo"; die;
};
print $var'
hallo
```

Doch wie kann ich das jetzt als try-catch verwenden? Wie das obige Beispiel zeigt, bricht das Programm nicht ab, wenn es in einem Block-`eval` auf ein `die` stößt. Wenn der Code im `eval` stirbt, wird die Spezialvariable `$@` gesetzt (siehe auch `perldoc perlvar`). Diese kann man nach dem `eval` überprüfen.

```
eval{
    # code hier;
};
if( $@ ){
    print "Code im eval
           wurde abgebrochen: $@";
}
```

Hier ist also das `eval` das "try" in Java und der `if`-Block ist das "catch". Allerdings muss man hier etwas aufpassen: Durch sogenannte "Race Conditions" kann es passieren, dass die Variable `$@` gesetzt wird, obwohl es in dem `eval`-Block direkt davor zu keinen Problemen kam. Aus diesem Grund kann man das `eval` und die Fehlerausgabe verbinden:

```
eval{
    # code hier;
    1;
} or do {
    # Ausgabe von $@ hier
};
```

Die `1;` im `eval`-Block ist wichtig, da es sonst passieren kann, dass der Code im Block etwas "unwahres" zurück gibt und dann die Ausgabe von `$@` erfolgt:

```
eval{
    0
} or do {
    print "Problem";
};
```

Obwohl hier im `eval` nicht wirklich ein Problem aufgetaucht ist, wird trotzdem "Problem" ausgegeben.

Module bieten auch ein try-catch

Es gibt auf CPAN ein paar Module, die eine Fehlerbehandlung und auch try-catch zur Verfügung stellen. Diese sind aber auch nicht unbedingt das Wahre, da `Error::TryCatch` ein Source-Filter ist und `Exception::Class::TryCatch` bietet kein echtes try-catch an.

Insgesamt muss man sagen, dass das Exception-Handling in Perl noch stark verbesserungswürdig ist. `eval` ist aber immerhin ein Anfang.

Renée Bäcker

CPAN News IX

Module::ScanDeps

Wer schon immer wissen wollte, welche Module durch seinen Code geladen werden (auch von Modulen, die nicht aus der eigenen Tastatur stammen), sollte sich mal `Module::ScanDeps` anschauen. Damit bekommt man aufgelistet, welche Abhängigkeit von welchem Modul geladen wird.

```
use strict;
use warnings;
use Module::ScanDeps;
use Data::Dumper;

my $deps = scan_deps(
    files   => [$0],
    recurse => 1,
);

print Dumper $deps;
```

Acme::Tiroler

Für die Sprachfreunde unter den Perl-Programmierern gibt es viele Module. Für Freunde des Tiroler Dialekts gibt es jetzt `Acme::Tiroler`. Damit kann man in Tiroler Dialekt programmieren.

```
#!/usr/bin/env perl

use warnings;
use strict;
use Acme::Tiroler;

sagsch "tirol isch lei oans\n" odr

mei $a isch 7 odr
sagsch "$a\n" odr
```

Log::Dispatch::Twitter

Jetzt ist das Logging von Programmen bei Web 2.0 angekommen. Für `Log::Dispatch`- und `Log::Log4perl`-Nutzer gibt es dank `Log::Dispatch::Twitter` die Möglichkeit, Log-Ausgaben direkt an Twitter zu senden.

```
use Log::Dispatch;
use Log::Dispatch::Twitter;

my $logger = Log::Dispatch->new;

$logger->add(Log::Dispatch::Twitter->new(
    username  => "foo",
    password  => "bar",
    min_level => "debug",
    name      => "twitter",
));

$logger->log(
    level   => 'error',
    message => 'We applied the cortical
                electrodes but were unable
                to get a neural reaction
                from either patient.',
  );
```

File::CountLines

In Foren taucht immer wieder die Frage auf, wie man am besten die Anzahl von Zeilen in einer Datei bestimmen kann. Jetzt fällt die Antwort noch leichter: Nimm `File::CountLines`.

```
#!/usr/bin/perl

use strict;
use warnings;
use File::CountLines qw(count_lines);

print count_lines( $0 );
```

Time::y2038

Kaum haben wir das Jahr-2000-Problem hinter uns, kommt das nächste "Zeitproblem" auf uns zu: Mit 32-bit-Zahlen lassen sich Epochensekunden nicht mehr ins richtige Datum übersetzen wenn es über das Jahr 2038 hinausgeht. Mit `Time::y2038` kann man das Problem lösen.

```
use Time::y2038;

# Sat Dec  6 03:48:16 142715360
print scalar gmtime 2**52;
```

Net::Twitter::Search

`Log::Dispatch::Twitter` wurde bereits vorgestellt. Wer jetzt nach seinen Logausgaben suchen will, kann `Net::Twitter::Search` verwenden.

```
my $twitter = Net::Twitter::Search->new();

my $results = $twitter->search('hello');
foreach my $tweet (@{ $results }) {
  my $speaker =  $tweet->{from_user};
  my $text = $tweet->{text};
  my $time = $tweet->{created_at};
  print "$time <$speaker> $text\n";
}
```

Neues von TPF

Grant-Updates

Alberto Simões hat Neuigkeiten zu den laufenden Grants:

Extending BSDPAN

Colin hat sich Ende November/Anfang Dezember wieder mit dem BSDPAN-Code auseinandergesetzt und ein paar unterschiedliche Ansätze entworfen.

Embedding perl in C++ Anwendungen

Leon hat libperl nach Perl 5.10 portiert, allerdings noch nicht ausgiebig getestet. Er hat Globs und Filehandles hinzugefügt und arbeitet gerade an Perl 5.8 Regulären Ausdrücken. Das Projekt ist bei Google Code zu finden: http://code.google.com/p/libperl/. Weiterhin hat Leon Timmermans die TAP-Ausgaben verbessert und als nächstes stehen Beispielcodes auf dem Programm. Leon hat Unit Tests hinzugefügt, ein paar Bugs gefixt und zwei neue Methoden hinzugefügt. Weiterhin hat er am Export-Code Änderungen vorgenommen und mit der Dokumentation des Exportings begonnen (http://code.google.com/p/libperl/wiki/Exporting). Außerdem hat Leon Tests für Skalar-Referenzen und Subroutinen-Referenzen hinzugefügt. Darauf basierend hat er das Handling von Referenzen und Variablen geändert. Das ganze wird jetzt unter GCC kompiliert.

Tcl/Tk access for Rakudo

Vadim hat einen ersten Patch an die Parrot-Entwickler geschickt.

Perl cross-compilation für linux und WinCE

Vadim Konovalov sieht sich in der Mitte des Grants. Einige Patches hat er an die Perl 5 Porters geschickt. Als Ziel des Abschlusses wurde Januar 2009 genannt.

Portierung von PyYAML nach Perl

Ingy döt net hat einige der neuen Module in der letzten Woche auf CPAN veröffentlicht. Mehr gibt's unter http://www.socialtext.net/yaml/index.cgi?the_new_yaml_pm . Bis Weihnachten soll der Grant abgeschlossen sein.

Test::Builder 2

Michael Schwern arbeitet daran, sonst keine weiteren Informationen

localtime() und gmtime() 2038-sicher machen

Der y2038-Grant für Michael Schwern wurde soweit abgeschlossen und wartet nur noch auf die Integration in die 5.10.x-Reihe.

Bugfixing in Archive::Zip

Alan Alavi ist weiterhin dabei, Bugs zu fixen. Der Fortschritt kann unter http://blog.alanhaggai.org/ verfolgt werden.

SMOP: Simple Meta Object Programming

Die letzten zwei Wochen wurde viel am Design gearbeitet, z.B. wie "for" und "map" umgesetzt werden. Zusätzlich hat Daniel Ruoso hat sehr viel in diesem Grant getan. Er hat zum Beispiel ein XS-Binding erstellt, mit dem SMOP in Perl 5 verwendet werden kann. Er hat in Aussicht gestellt, dass Perl 5.12 Perl 6 vollkommen unterstützt. Mehr Berichte sind im Blog der Perl Foundation zu finden.

Perl on a Stick

Dieser Grant ist so gut wie beendet. Der abschließende Bericht von Adam Kennedy folgt noch.

Barcode support in Act

Andrew Shitov hat den Grant abgebrochen

Abschlussbericht Smolder-Grant

Michael Peters hat den Smolder-Grant beendet. Die Ziele des Grants waren:

• Das eigene XML-Format durch TAP und TAPx::Parser ersetzen

• Smolder soll mit Modulen im CPAN-Stil einfacher umgehen.

• Aufsetzen eines Smolder Servers für die CGI::Application Community als Test-Backend für die mehr als 110 Module aus der CGI::Application-Familie

• RSS-Feeds für jedes Projekt und jeden Entwickler als Alternative zu den E-Mail-Benachrichtigungen

Die Punkte 3 und 4 wurden leicht abgeändert. Der Smolder Server wurde nicht für die CGI::Application Community sondern für Parrot aufgesetzt und statt RSS-Feeds gibt es Atom-Feeds.

Bewilligte Grants im 4. Quartal 2008

Die Grants für das 4. Quartal 2008 wurden bewilligt. In dieser Runde werden nur zwei Projekte von der Perl Foundation unterstützt, da in diesem Jahr schon 25.000 Euro für Grants ausgegeben wurden und die Spenden an die Perl Foundation die Kosten nicht decken.

Glücklicherweise fördert Vienna.pm zwei weitere Projekte mit insgesamt 4.000 Euro.

Von der TPF unterstützte Projekte:

• Moose Documentation von Dave Rolsky

• The Mojo Documentation Project von Sebastian Riedel

Von Vienna.pm unterstützte Projekte:

• The Perl Survey von Kieren Diment

• Integrating Padre with Parrot and Rakudo von Gabor Szabo

Erster 'Hague'-Grant geht an Patrick Michaud

Die Perl Foundation hat einen ersten Grant aus der "Ian Hague"-Spende bewilligt.

Patrick Michaud wird damit für eine Fortsetzung des Mozilla-Grants bezahlt. Dies beinhaltet die Arbeit an den Regulären Ausdrücken und anderen Internas.

Zweiter "Hague"-Grant: Eine Kommandozeile für Perl 6

Ein zweiter Grant aus dem Topf der Ian Hague-Spende wurde von der Perl Foundation bewilligt:

Jerry Gay wird an die S19 Synopsis definieren, die die Kommandozeile von Perl 6 betrifft. Zusätzlich wird er die Synopsis für Rakudo (Perl 6 auf Parrot) umsetzen.

Abschlussbericht von Patrick Michaud (Mozilla/TPF-Grant)

Schon vor ein paar Wochen hat Patrick Michaud seinen Abschlussbericht über den Mozilla/TPF-Grant veröffentlicht.

Richard Dice von der Perl Foundation hebt besonders hervor, dass dieser Bericht sehr deutlich zeigt, wie die Grants der Perl Foundation bei der Weiterentwicklung von Perl helfen können. Der abschließende Bericht von Michaud ist in seinem use.perl.org-Journal zu finden (http://use.perl.org/~pmichaud/journal)

Grant für David Mitchell für das Release von Perl 5.10.1

Die Perl Foundation gewährt David Mitchell einen Grant für das Release von Perl 5.10.1.

Die Arbeiten beinhalten das Einspielen von über 400 Patches, Überarbeiten des "Smart-matching" und das Release an sich. Das Geld für den Grant wurde von Dijkmat BV bereitgestellt.

Perl 5.8.9 veröffentlicht

Nick Clark hat gestern eine neue Version der erfolgreichen 5.8.x-Serie veröffentlicht, die viele Verbesserungen beinhaltet.

Verbesserungen am Interpreter
Das neue Release beinhaltet auch einige Verbesserungen am Interpreter. So wurde die "Unicode Character Database" für das Unicode-Handling von der Version 4.1.0 auf 5.1.0 aktualisiert.

Der interne Caching Code für UTF-8 wurde verbessert, so dass es stabiler und schneller läuft.

Die Perl-Installation kann nun beliebig im Dateisystem verschoben werden – ein Segen für Systemadministratoren.

Die komplette Liste an Verbesserungen ist unter perl589delta.pod zu finden: http://search.cpan.org/~nwclark/perl-5.8.9/pod/perl589delta.pod

Perl 5.8.9 wird das letzte "große" Release der 5.8.x-Reihe sein. Danach wird es nur noch Releases mit Bugfixes für Sicherheitsprobleme oder Plattformspezifische Bugs geben.

Den Anwendern wird geraten auf Perl 5.10.x umzusteigen.

Booking.com spendet 50.000 US-Dollar

Das Unternehmen Booking.com hat 50.000 US-Dollar an die Perl Foundation gespendet. Damit soll die Weiterentwicklung von Perl vorangetrieben werden.

Neben der 50.000-Dollar-Spende stellt Booking.com auch Hardware und Arbeitszeit für das neue git-Repository von Perl zur Verfügung.

Perl 5: Wechsel zu git

In den letzten Tagen wurde der Perl 5 Code auf ein git-Repository umgezogen. Damit wird das bisherige System "Perforce" abgelöst.

Der Umzug zu git bietet einige Vorteile für die Perl-Entwickler:
- Durch das öffentliche Repository und der git-Unterstützung von verteiltem Arbeiten, wird die Arbeit am Perl 5 Code einfacher.
- Da git Open Source ist, haben alle Entwickler den gleichen Zugang zu den für die Arbeit am Perl 5 Code notwendigen Tools.
- Für die Core-Entwickler ist es weniger administrative Arbeit, Änderungen zu integrieren.
- Entwickler außerhalb des Kernteams kann einfacher an experimentellen Änderungen arbeiten bevor sie die Änderungen für das nächste Release vorschlagen.
- Eine große Liste an verbesserten Tools zu Analyse des Repositories und der Änderungen sind jetzt verfügbar.
- Das git Repository enthält jede einzelne Version von Perl 5, die jemals veröffentlicht wurde.

Interessierte Entwickler können eine Kopie des Perl 5 git Repository unter http://perl5.git.perl.org/perl.git bekommen.

Hague Grant: Jonathan Worthington und "Rakudo Dispatch and Role Enhancements"

Jonathan Worthingtons Grant-Antrag ist von der Perl Foundation angenommen worden. Damit wird er in den nächsten Wochen und Monaten an folgenden Sachen arbeiten (Auszug aus dem Grant-Antrag):

D1. Register symbols in the namespace at compile time, allowing elimination of current "types must start with upper-case letters" hack and detection of re-defined routines, as specified in S06 and S12
D2. Implementation of junction auto-threading, as specified in S09
D3. Implementation of submethods, as specified in S12
D4. Finish implementation of delegation ("handles" trait verb), as specified in S12
D5. Implementation of parametric roles, as specified in S12

chromatic wird hierbei als Grant-Manager fungieren.

Neuer Grant Manager

Die Perl Foundation hat einen neuen Grant Manager gesucht. Das Grant Komitee hat daraufhin einige Bewerbungen erhalten:

* Nuno Carvalho
* Renée Bäcker
* Scott "KONOBI" McWhirter
* Andrew Shitov
* Steve Peters
* Robin Berjon
* Makoto Nozaki
* David Nicol

Über Weihnachten hat das Grant Komitee abgestimmt und Renée Bäcker zum neuen Grant Manager gewählt.

Mehr zum Grant Komitee gibt es auf der Seite der Perl Foundation.

Microsoft hostet zentrales Testsystem für CPAN-Module

Microsoft und (Strawberry-)Perl gehen eine Partnerschaft ein, wie man so schön sagt: Adam Kennedy berichtet, dass Microsoft jedem CPAN-Autor kostenlosen Zugang zu zentral gehosteten Virtuellen Maschinen mit allen wichtigen Windows-Versionen ermöglicht.

Damit wird der Perl-Community ermöglicht, ohne Kosten und Installations-Aufwand Module auf diversen Windows-Versionen zu testen und debuggen. Es ist also nicht nur wie bei den Smoke-Tests möglich, die beim Modul mitgelieferten Tests durchlaufen zu lassen, sondern auch ein lokales Entwickeln und Debuggen wird möglich. Obwohl ich nach Möglichkeit nicht mit Windows arbeite, halte ich dies für eine sehr gute und sehr wichtige Sache.

Im ersten Schritt werden nur eine Hand voll Windows-Versionen aus der XP- und Vista-Reihe zur Verfügung stehen, aber laut Adam sollen es mehr werden. In den nächsten Tagen sollen ihm die Admin-Passwörter übergeben werden, und dann kann es losgehen. Potentiell 7000 Autoren können das System nutzen, es ist also kein Kleinkram, auch wenn nur ein Teil davon dieses Angebot tatsächlich nutzen wird.

Und letztendlich können beide Seiten davon profitieren: Perl wird gestärkt, wenn möglichst viele Module problemlos auch unter Windows laufen, denn in vielen Firmen herrscht zumindest auf dem Desktop immer noch eine Windows-Monokultur. Und Microsoft profitiert von einer besseren Windows-Unterstützung, da dies langfristig möglicherweise weniger Nutzer zu Linux, *BSD oder OS X treibt.

Hoffen wir, dass das Angebot von den CPAN-Autoren auch angenommen wird.

Alvar Freude

(http://www.perl-blog.de/2008/12/microsoft-hostet-cpan-testserver.html)

Termine

Februar 2009

02. Treffen Vienna.pm
03. Treffen Frankfurt.pm
05. Treffen Dresden.pm
06.-08. Frozen Perl 2009 (Minneapolis)
09. Treffen Ruhr.pm
11. Treffen Hamburg.pm
12. Treffen Cologne.pm
16. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
24. Treffen Bielefeld.pm
25. -27. Deutscher Perl-Workshop 11.0
25. Treffen Berlin.pm
26.-27. Spanischer Perl-Workshop
28. Belgischer Perl-Workshop

März 2009

02. Treffen Vienna.pm
03. Treffen Frankfurt.pm
05. Treffen Dresden.pm
07. 2. Ukrainischer Perl-Workshop
09. Treffen Ruhr.pm
11. Treffen Hamburg.pm
12. Treffen Cologne.pm
16. Treffen Erlangen.pm
Treffen München.pm
18. Treffen Darmstadt.pm
24. Treffen Bielefeld.pm
25. Treffen Berlin.pm

April 2009

02. Treffen Dresden.pm
06. Treffen Vienna.pm
07. Treffen Frankfurt.pm
08.. Treffen Hamburg.pm
09. Treffen Cologne.pm
13. Treffen Ruhr.pm
15. Treffen Darmstadt.pm
16.-17. Nordic Perl-Workshop 2009
20. Treffen Erlangen.pm
28. Treffen Bielefeld.pm
29. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

Kennen Sie weitere Termine, die in der nächsten Ausgabe von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine EMail an:

termine@foo-magazin.de

LINKS

Unter Perl-Nachrichten.de sind deutschsprachige News rund um die Programmiersprache Perl zu finden. Jeder ist dazu eingeladen, solche Nachrichten auf der Webseite einzureichen.

http://www.perl-nachrichten.de

Perl-Community.de ist eine der größten deutschsprachigen Perl-Foren. Hier ist aber nicht nur ein Forum zu finden, sondern auch ein Wiki mit vielen Tipps und Tricks. Einige Teile der Perl-Dokumentation wurden ins Deutsche übersetzt. Auf der Linkseite von Perl-Community.de findet man viele Verweise auf nützliche Seiten.

http://www.perl-community.de

Das Online-Zuhause der Perl-Mongers. Hier findet man eine Übersicht mit allen Perlmonger-Gruppen weltweit. Außerdem kann man auch neue Gruppen gründen und bekommt Hilfe...

http://www.perlmongers.de/
http://www.pm.org/

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern. Der 11. Deutsche Perl-Workshop findet 2009 in Frankfurt statt.

http://www.perl-workshop.de

Die Perl-Foundation nimmt eine führende Funktion in der Perl-Gemeinde ein: Es werden Zuwendungen für Leistungen zugunsten von Perl gegeben. So wird z.B. die Bezahlung der Perl6-Entwicklung über die Perl-Foundationen geleistet. Jedes Jahr werden Studenten beim "Google Summer of Code" betreut.

http://www.perl-foundation.org

Auf Perl.org kann man die aktuelle Perl-Version downloaden. Ein Verzeichnis mit allen möglichen Mailinglisten, die mit Perl oder einem Modul zu tun haben, ist dort zu finden. Auch Links zu anderen Perl-bezogenen Seiten sind vorhanden.

http://www.Perl.org

// SEIBERT / MEDIA

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

//SEIBERT/MEDIA besteht aus den vier Kompetenzfeldern Consulting, Design, Technologies und Systems und gehört zu den erfahrenen und professionellen Multimedia-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 60 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte. Beratungen im Bereich Online-Marketing und Usability runden das Leistungsportfolio ab.

Ihre Aufgabe wird sein, in unserer Entwicklungsabteilung im Team komplexe E-Business Applikationen zu entwickeln. Dabei ist objektorientiertes Denken genauso wichtig, wie das Auffinden individueller und innovativer Lösungsansätze, die gemeinsam realisiert werden.

Wir freuen uns auf Ihre Bewerbung unter www.seibert-media.net/jobs.
// SEIBERT / MEDIA GmbH, Rheingau Palais, Söhnleinstraße 8, 65201 Wiesbaden
T. +49 611 20570-0 / F. +49 611 20570-70, bewerbung@seibert-media.net

„Statt mit blumigen Worten umschreiben unsere Programmierer den Job so:

Apache, Catalyst, CGI, DBI, JSON, Log::Log4Perl, mod_perl, SOAP::Lite, XML::LibXML, YAML“

Warum Fachleute auf Schulungen gehen sollten

Externe Schulungen sind wie eine Studienfahrt mit Fremden, die am gleichen Thema arbeiten. 5 Tage lang 'mal nicht mit den eigenen Kollegen im immer gleichen Brei schwimmen. Es ist nämlich nicht richtig, alles im Selbststudium lernen zu wollen: Jede(r) von uns hat die Fundamente seines Könnens in Schulen und Universitäten gelernt, und gerade wer im Betrieb stark belastet ist, hat keine Chance, schwierige Themen „nebenbei“ am Arbeitsplatz zu erlernen oder neue Kollegen anzulernen.

Schauen Sie 'mal: www.Linuxhotel.de

Wer sich wirklich intensiv auf eine Arbeit einläßt, will auch Spaß dabei haben! Im Linuxhotel kombinieren wir deshalb ganz offen eine äußerst schöne und luxuriöse Umgebung mit höchst intensiven Schulungen, die oft bis in die späten Abendstunden zwanglos weitergehen. Natürlich freiwillig! Wer will, zieht sich zurück! Und weil unser Luxushotel ganz weitgehend per Selbstbedienung läuft, sind wir gleichzeitig auch noch sehr preiswert.