Mit Stellenmarkt

bfr 207,– hfl 10,95 lfr 205,– öS 70,– sfr 8,50 **DM 8,50** H 10554

7
Juli 2000

iX

MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK

IMAP-fähige Mail-Clients im Vergleich:

E-Mail im Griff

Signieren, verschlüsseln, organisieren

Neues Tutorial:
JavaServer Pages

Rechnerkopplung:
Linux-Cluster Hochgeschwindigkeitsnetze

Objektorientierung:
CORBA mit Visual Basic

Web-Programmierung:
PHP 4.0 Signed Java-Applets

Sichere Namensdienste:
DNS-Cache

Neues Sun-Unix:
Solaris 8

Linux-GUI:
KDE 2.0

Anschluss für die Firma:
Internet-Provider

Juli 2000
Magazin für professionelle Informationstechnik
http://www.heise.de/ix/
HEISE
4 399148 508506 07

Neues vom Perl-Compiler

Reise ins Ich

Michael Schilli

Lassen sich aus Perl-Skripts binäre Programme erstellen, die ganz ohne den Perl-Interpreter laufen? Der Perl-Compiler hilft Skripts, ihr eigenes Selbst zu röntgen und sich in allerlei Zielformate zu transformieren. Auch eine Schnittstelle für den C-Compiler ist dabei.

Kaum eine Frage erscheint so oft in den einschlägigen Newsgroups wie die nach einem Perl-Compiler. Dabei liefert das Programm den Compiler frei Haus: Bei jedem Aufruf eines Skripts durchforstet der Interpreter den Code nach Syntaxfehlern, kompiliert Perl-Konstrukte in internen Bytecode und führt diesen dann mit einer virtuellen Maschine aus.

Das Interesse an einem Perl-Compiler zielt aber darauf ab, ob es nicht möglich sei, ein Skript in ein ausführbares Programm auf der jeweiligen nativen Hardwareplattform umzuwandeln. Nun – im Prinzip ja.

Freilich muss das fertige ausführbare Programm hierzu die benötigten Teile der virtuellen Perl-Maschine enthalten – schließlich soll beispielsweise Perls praktische Garbage Collection auch in einem Binary funktionieren.

Der Perl-Compiler, den ursprünglich Malcolm Beattie aus der Taufe hob, lässt *perl* ein Skript laden und nutzt das Selbstinspektionsmodul *B*, um die interne Darstellung des Skripts zu analysieren. Daraus generiert er Anweisungen beispielsweise in der Programmiersprache C. Die C-Datei übersetzt dann der C-Compiler auf der jeweiligen Plattform in eine Object-Datei und linkt sie mit allerlei Bibliotheken, unter anderem mit einer virtuellen Perl-Maschine. Der Begriff 'Perl-Compiler' trifft daher nicht ganz den Punkt – 'Code-Inspektor' oder 'Transformierer' wäre eigentlich passender.

Dieses Verfahren führt zu fettleibigen Dateien: Ein Executable für ein 'Hello World!' misst fast 1 MByte, und kommen Erweiterungsmodule hinzu, steigt die Dateigröße schnell weiter an. Auch die Ausführungsgeschwindigkeit des kompilierten Programms unterscheidet sich nicht wesentlich von der eines interpretierten Skripts, denn nach wie vor rattert die gleiche virtuelle Maschine vor sich hin.

Lange Skripts laden schneller

Einzig die Ladegeschwindigkeit längerer Skripts, gemessen vom Aufruf bis zu dem Zeitpunkt, an dem das Skript mit der ersten Instruktion loslegt, verbessert sich. Während *perl* jedes Mal vom Source- auf den Bytecode schließen muss, steht in einem Binary alles schon bereit, und der Prozessor kann sofort mit dem Ausführen beginnen.

Doch Vorsicht: Ein kurzes 'Hello-World'-Skript startet trotzdem schneller als ein übersetztes Binary, denn *perl* schluckt und transformiert eine 20-Byte-Datei schneller als das Betriebssystem ein 1-MByte-Programm von der Platte ausführungsbereit in den Speicher laden kann. Besonders wenn das Betriebssystem den Interpreter noch von einem vorausgehenden Aufruf her im Speicher stehen hat, gewinnt das Perl-Skript mit Längen.

Das mit Perl 5.6.0 ausgelieferte *perlcc* abstrahiert die komplexen Vorgänge der Umwandlung, und ein kurzes Skript *hello.pl* des Inhalts

```
#!/usr/bin/perl -w
print "Und es geht doch!", "\n";
```

erfährt mit dem Aufruf

```
perlcc hello.pl
```

die magische Wandlung. Allerdings enthält *perlcc* aus perl5.6.0 einen kleinen Fehler, siehe Kasten 'Kleine Fehlerkorrektur'. Was bei der Konvertierung passiert: Das Modul *B*, fester Bestandteil der Perl-Distribution, kann seit Version 5.005 den so genannten Syntaxbaum, die interne Struktur eines Skripts, analysieren. Dieses Backend-Modul *B* wird dann von einem Frontend-Modul *O* genutzt, um den Baum in neue Formen zu verwandeln. Im obigen Fall erzeugt *perlcc* zunächst eine C-Datei *hello.pl.c*, und ruft den C-Compiler mit allen notwendigen Include-Pfaden und Bibliotheken auf,

um das Executable *hello* zu generieren. Das auf einem Linux-System 698 159 Byte große Programm läuft dann auch ohne installiertes Perl.

Zum Geschwindigkeitsvergleich: Der Code in Listing *benchmark.pl* vergleicht Source- und kompilierte Version zweier Skripts. Das oben vorgestellte *hello.pl* tritt gegen das folgende kurze CGI-Skript *cgi.pl* an:

```
#!/usr/bin/perl -w
use CGI qw(:all);
print header(), h1("Hallo!");
```

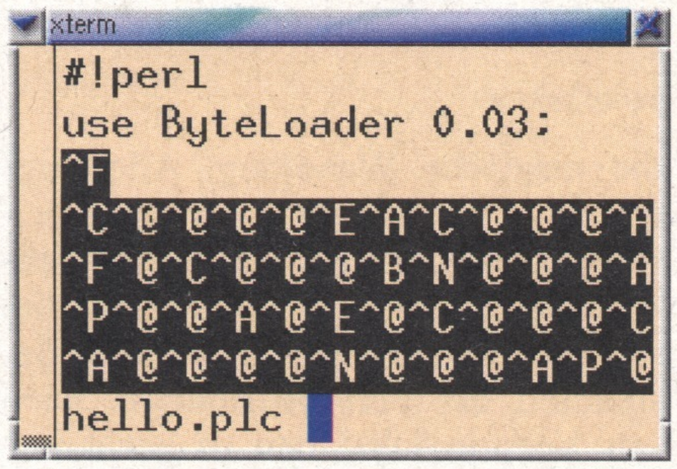
Letzteres lädt das mächtige CGI-Modul, bevor es einen Header und eine kurze Nachricht ausgibt. Das Ergebnis nach den eingestellten 100 Durchgängen auf einem RedHat-6.1-Linux-System:

```
Skript:  hello.pl .....1.69 sec
Exe:     hello .......3.30 sec
Skript:  cgi.pl .....22.86 sec
Exe:     cgi .......12.61 sec
```

Während also das kurze *hello.pl* in der Sourceversion sogar schneller durchläuft als die übersetzte Variante, bringt die Übersetzung im Falle eines großen Zusatzmoduls wie CGI Geschwindigkeitsvorteile – wie gesagt, die Anlauf- und Ladezeit machen den Unterschied, nicht die eigentliche Ausführung der Skript-Befehle.

Zugriff auf Perls Bytecode

Perl-Skripts liegen traditionell immer im Source-Code vor. Der zwischenzeitlich zur Ausführung erzeugte Bytecode war bislang nicht zugänglich. Während Java oder Python diesen Bytecode in Dateien ablegen und bei folgenden Aufrufen den Transformationsschritt vom Source- zum Bytecode sparen, ging dies mit Perl bislang nicht.



***hello.plc*, das Perl-Skript *hello.pl* im Bytecode**

Mit dem Perl-Compiler kann man nun auch den temporären Perl-Bytecode speichern. Ein so halb übersetztes Skript startet ohne Verzögerung durch, es lädt lediglich den Bytecode und wirft die virtuelle Maschine an. Der Bytecode ist unabhängig von der verwendeten Hardware- oder Betriebssystemplattform. Auch hier erspart es *perlcc* dem Anwender, sich mit den Eigenwilligkeiten der *B*- und *O*-Module herumzuschlagen. Der Aufruf

```
perlcc -b hello.pl
```

von der Kommandozeile erzeugt die Bytecode-Datei *hello.plc*. Deren Innereien sind wegen des binären Bytecodes unleserlich, es lohnt sich aber dennoch, einen Blick hineinzuwerfen (Abbildung 1). Wie man sieht, handelt es sich bei *hello.plc* tatsächlich um ein Perl-Skript. Es nutzt das Modul *ByteLoader*, einen so genannten Source-Filter (siehe Kasten 'Source-Filter'), der den nach der *use*-Zeile folgenden Bytesalat liest und dem Perl-Interpreter zu Fressen gibt. Der Aufruf

```
perl hello.plc
```

startet das Skript direkt in der Bytecode-Repräsentation. Mit den Compiler-Modulen *B* und *O* lassen sich noch allerhand Tricks anstellen: So ist es möglich, aus der internen Darstellung eines Perlskripts wieder eines zu generieren.

Dies führt zu überraschenden Effekten: Codestücke, die *perl* aus Optimierungsgründen eliminiert hat, fehlen plötzlich in der Quelle, und meistens entsteht aus dem Bytesalat sogar wieder ein schön formatiertes Skript, auch wenn das Original schlampig hingeschrieben wurde.

Steht in einem Perl-Skript *calc.pl* etwa die Zeile

```
print "2 mal 2 ist ", 2*2, "\n";
```

so wird *perl* den arithmetischen Ausdruck 2*2 in der internen Darstellung schnell zu 4 vereinfachen, wie man ganz leicht durch den Decompiler-Aufruf

```
perl -MO=Deparse calc.pl
```

zeigen kann. Dieser weist *perl* an, das *O*-Modul zu laden und dessen *Deparse*-Schnittstelle zu aktivieren. Die Ausgabe zeigt dann tatsächlich

```
print '2 mal 2 ist ', 4, "\n";
calc.pl syntax OK
```

Auch unerreichbare Codeblöcke verwirft *perl* sofort nach dem Kompilieren: Ein Skript *zeroloop.pl*, das

```
while(0) {
    print "Nie erreicht!\n";
}
```

enthält, steht nach dem Entschlacken mit

```
perl -MO=Deparse zeroloop.pl
```

einfach als

```
;
```

Dieses Skript misst die Zeiten für übersetzte und interpretierte Programme.

LISTING benchmark.pl

```
#!/usr/bin/perl -w
use Benchmark;
timethese(100, { 'Exe: hello' => 'system("./hello >/dev/null 2>&1")',
                 'Script: hello.pl' => 'system("./hello.pl >/dev/null 2>&1")',
                 });
timethese(100, { 'Exe: cgi' => 'system("./cgi >/dev/null 2>&1")',
                 'Script: cgi.pl' => 'system("./cgi.pl >/dev/null 2>&1")',
});
```

Wie funktioniert ein Source-Filter?

Mit dem ByteLoader kann *perl* in Bytecode übersetzte Perl-Skripts ausführen. Er arbeitet als Source-Filter, einer trickreichen, aber recht selten genutzten Perl-Funktion: Ein mit *use Filter::MyFilter* eingebundener Source-Filter veranlasst *perl*, den folgenden Code der aktuellen Datei nicht mehr direkt an den Perl-Parser weiterzugeben, sondern vorher einer Transformation zu unterwerfen.

Ein Beispiel: In der Filtersammlung *PMQS/Filter-1.18.tar.gz* auf dem CPAN befindet sich der *Filter::exec*, der ein externes Programm über die Quellen des gegenwärtigen Perl-Skripts laufen lässt, bevor der Parser es sich schnappt.

Das Prograämmchen

```
use Filter::exec qw( rot13 );
cevag "Uryyb, jbeyq!\a";
```

aktiviert zunächst das externe Unix-Programm *rot13* (falls dieses nicht vorhanden ist, tuts auch die Zeile *qw(tr n-za-mN-ZA-M a-zA-Z)*), das alle Buchstaben des Alphabets um 13 Stellen nach rechts verschiebt. Dieser Vorgang ('Rot13-Transformation' genannt) verwandelt *cevag "Uryyb, jbeyq!\a";* flugs in den String *print "Hello, world!\n";* – eine Sprache, die der Perl-Interpreter sofort versteht und ausführt. Die Manual-Seite *perldoc perlfilter* erläutert die Funktionsweise von Source-Filtern im Detail.

da, ein einzelnes Semikolon, ein leeres Perl-Skript, das nichts tut. Auch zur Codeverschönerung taugt *Deparse* bedingt: Ein hektisch als

```
my $a = 42;
while( $a--){print "$a ";}print "\n";
```

hingepfeffertes Skript modelliert das Deparse-Frontend mit Hingabe:

```
my $a = 42;
while ($a--) {
    print "$a ";
}
print "\n";
```

Mehr zum Thema Perl-Compiler bietet die Manual-Seite *perlcompile*, die auf das Kommando *perldoc perlcompile* zum Vorschein kommt. *perldoc perlcc* zeigt die Optionen für den Kompiliervorgang. Die Moduldokumentationen zu *B, B::C, B::CC, B::Deparse, B::Bytecode*, und *O* geben nützliche Hinweise, auch wenn alles noch etwas wackelig dasteht – schließlich handelt es sich um experimentelle Software, die keinesfalls für den Produktionseinsatz gedacht ist. Auch der Autor von *perlcompile* sieht das so, denn die Statuswerte für die einzelnen Module, die sich zwischen 0 wie 'noch nicht implementiert' bis 10 für 'Stabil, Fehler würden mich wundern' bewegen, schwanken zwischen 6 und 8 – es bleibt spannend. (ck)

MICHAEL SCHILLI

arbeitet als Web-Engineer für AOL/Netscape in Mountain View, Kalifornien. Er ist Autor des 1998 bei Addison-Wesley erschienenen Buches 'GoTo Perl 5'.

Kleine Fehlerkorrektur

Der Perl-Compiler in Version 5.6.0 enthält einen kleinen Fehler, der sich aber leicht beheben lässt. Im Skript *perlcc*, das die Distribution im Perl-Ausführungspfad installiert (üblicherweise in */usr/bin* oder */usr/local/bin*), muss in Zeile 393 die Anweisung *use DynaLoader;* einfügt werden, sodass der Code an dieser Stelle folgendermaßen aussieht:

```
print $fd <<"EOF";
    use DynaLoader;
    use FileHandle;
```

Den Fehler werden die fleißigen Perl-Porter mit neuerenVersionen sicherlich beheben.