Proceedings des Deutschen Perl Workshops (Auszug)

WS-Orga-Team (Hrsg.)
Daniel Pfeiffer

Frankfurt am Main
25. bis 27. Februar 2009

Inhaltsverzeichnis

1 Makepp, das bessere make

http://makepp.sourceforge.net/

1.1 Autor

Daniel Pfeiffer (<occitan@esperanto.org>)

1.2 Bio Daniel Pfeiffer

Als Weltenbummler großgeworden, bin ich im Unixland zuhause, besonders in der südafrikanisch-antarktischen Gegend, bei possierlichen schwarz-weißen Vögeln, mit gelben Schnäbeln. Perl betreibe ich seit kurz vor dem Sprung auf Version 5. Der Einsatz reicht von Einzeilern für ad hoc Fragestellungen, über teilweise bei Cpan registrierte Skripte bis hin zu Großprojekten wie makepp oder der Migration umfangreicher hysterisch gewachsener Siemens BS2000 Jobgerüste. Gerade bei makepp, das proportional zur Build-Größe viele Informationen abwägen muß, bin ich ganz altmodisch auf Performanz bedacht. Damit war ich auch im Linux Magazin Programmierwettbewerb (2008-12) Viertschnellster, und Schnellster unter den Skriptsprachen.

1.3 Abstract

Wie übersetzt man die schnellste Grafikkarte der Welt? Mit makepp, genau wie ein C++- oder sonstiges Software-Projekt – wenn man keine Phantomfehler suchen will, die erst nach „make clean“ verschwinden. Nicht nur werden Makefiles kleiner und flexibler für Umorganisationen, mit Graphen behält man den Überblick. Anders als klassische makes, aber doch kompatibel zu Gnu make, analysiert makepp selber verzeichnisübergreifend alle Abhängigkeiten und merkt sie sich auch für die Zukunft, um immer zuverlässig alles Nötige neu aufzubauen. Dank dieses tiefen Wissens bietet es einen Build Cache, um, auch z.B. projektweit, Neuübersetzungen des Selben zu vermeiden. Mit dem Repository-Mechanismus kann man erzeugte Dateien ohne spezielle Vorkehrungen von Quellen trennen, oder einen kompletten Build als Vorlage bereitstellen, wo dann individuelle Entwickler nur noch ihre geänderten Dateien brauchen.
Dieses umfangreiche Werkzeug ist nicht nur in Perl geschrieben, sondern erlaubt auch den Perleinsatz auf allen Ebenen. Man kann Perl-Kode direkt in die Makefile oder als auszuführenden Befehl einer Regel oder als Funktionsausdruck schreiben. Wer das lieber trennt, definiert stattdessen in einer Makefile oder besser in einer Perlbibliothek eigene Direktiven, Befehle oder Funktionen. Für letztere besteht Zugriff auf das Rahmengerüst, welches auch die zahlreichen eingebauten Befehle nutzen. Wo Unix-Klassiker wie cut, expr, grep, sed oder sort daran kranken, daß jeder eine eigene Spezifikationssprache mitbringt, haben ihre in makepp eingebauten Pendants einfach Perl parat.

Wem das nicht weit genug geht, kann makepp auch noch mit Klassen erweitern, die entscheiden, wann eine Datei als geändert gilt (bei C-artigen Sprachen werden z.B. Leerzeichen- oder Kommentaränderungen ignoriert). Andere Klassen analysieren Befehle, um z.B. Includeverzeichnisse oder Bibliotheken zu erkennen. Darauf aufsetzend, parsen weitere Klassen die Eingabedateien, um z.B. nötige Includes zu identifizieren und nötigenfalls im Vorbeigehen aufzubauen.

1.4 Warum nicht Gnu make?

Der Platzhirsch stellt sich total dumm:

- Ignoriert Befehle, als wäre `gcc -DTU_DIES` gleich mit `gcc -DTU_JENES`.
- Ignoriert die System- und CPU-Architektur – fatal bei NFS.
- Ignoriert neue Befehlsversion oder geänderte Umgebungsvariablen.
- Abhängigkeit von Verzeichnis, ohne zu wissen, was dort passiert:
  `$(MAKE) -C`*`Verzeichnis`*

Das Symptom: mysteriöse Fehler. Die Arznei: `make clean; make`, was, zusätzlich zur vergeblichen Fehlersuche, in einem großen Projekt Stunden kosten kann. Mein Augenöffner von Peter Miller war: *Recursive make considered harmful*

1.5 Warum mögen Entwickler keine Aufbausysteme?

Die CPU und fast alle Programmiersprachen, egal ob prozedural oder OO, sind imperativ:

1. Tu dies.
2. Danach tu 5x das.
3. Wenn Bedingung erfüllt dann tu solches, sonst jenes.

Aufbauwerkzeuge, egal ob `*make*`, `ant`, `cook`, `*jam`, sind aber (wie Prolog) deklarativ bzw. logikgetrieben.

- Man programmiert nur eigenständige Schnipsel (imperativer Hybrid).
- Ob und wann ein Schnipsel ausgeführt wird entscheidet das Werkzeug.
- Ebenso was parallel ausgeführt werden kann.

Das stellt so manchen Entwickler vor das Rätsel, wann was warum getan wird.

1.6 Ausgiebige Protokollierung

Kleiner Ausschnitt aus dem Protokoll der Erstellung dieses Artikels, der u.A. begründet, warum etwas getan wurde, oder auch nicht:

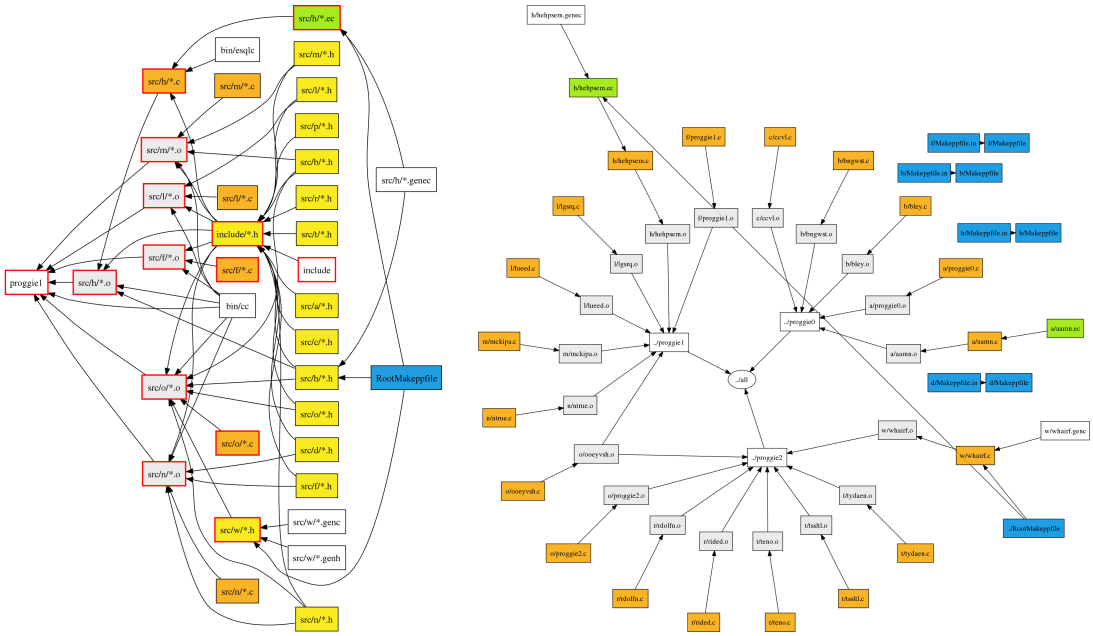
```
    Rebuild ‘article_PFEIFFER.tex’ because ‘article_PFEIFFER.pp’ changed
    ‘Makefile:190’ successfully executed for ‘article_PFEIFFER.tex’
    Trying to build ‘main.tex’
     Using rule ‘default rule’
     Targets ‘main.tex’ depend on
    ‘main.tex’ is up to date
  Rebuild ‘main.dvi’ because ‘article_PFEIFFER.pp’ changed
  ‘Makefile:184’ successfully executed for ‘main.dvi’
Rebuild ‘dvi’ because it is a phony target
```

Jede Meldung hat intern einen Schlüssel, nach denen man in dem riesigen Wust selektieren kann. Als Auswahlhilfe gibt es eine Ansicht die diese zeigt:

```
BUILD_CHANGED Rebuild ‘article_PFEIFFER.tex’ because ‘article_PFEIFFER.pp’ changed
SUCCESS ‘Makefile:190’ successfully executed for ‘article_PFEIFFER.tex’
TRY Trying to build ‘main.tex’
USE Using rule ‘default rule’
DEPEND Targets ‘main.tex’ depend on
UP_TO_DATE ‘main.tex’ is up to date
BUILD_CHANGED Rebuild ‘main.dvi’ because ‘article_PFEIFFER.pp’ changed
SUCCESS ‘Makefile:184’ successfully executed for ‘main.dvi’
BUILD_PHONY Rebuild ‘dvi’ because it is a phony target
```

1.7 Aufbau visualisieren

Man kann das, was im Aufbau geschieht, auf vielfache Weise visualisieren, wobei man, um der Fülle Herr zu werden, Dateinamen mit Perl substituiert um zu sagen was man sehen will, z.B. nur Suffixe, oder nur Verzeichnisse... Nicht so hübsch, aber oft übersichtlicher, kann man dasselbe auch als aktives HTML (mit zuklappbaren Untergraphen) oder als nackten Text begutachten.



1.8 Bißchen Makefile Syntax

```
include Datei                  # Anweisung, hier z.B. zum Einfügen

A = 1 2 3 \
    4 5                        # Zuweisung, Leerzeichen bilden Liste
DATECMD =  $(shell date)       # Zeichenkette literal (samt '$'!)
DATENOW := $(DATECMD)          # Jetzt auswerten und danach nicht mehr
DATENOW1 ;= $(DATECMD)         # Nur einmal aber nicht sofort

Ausgabe: Eingabe               # Einzelregel, erzeugt Ausgabe aus Eingabe
        Shell Befehle

%.o: %.c                       # Musterregel: C Quelle zu Modul
       $(CC) $(CFLAGS) $(input) -o $(output)
```

1.9 Perl Versionen und Plattformen

Makepp wird mit allen Perl Versionen ab 5.6.0 getestet.
Makepp läuft:

- auf allen Unixen

- auf allen Windows Perls (Cygwin, MinGW, ActiveState, Strawberry)
  von Unix-nah mit `fork/exec` und symbolischen Verweisen, bis hin zu -fern
- Auf Großrechnern mit Ebcdic (BS2000, z/OS)

Diese Vielfalt zwingt einerseits portabel zu programmieren, andererseits sind leider auch Umgehungen diverser kleiner Bugs, die verschiedene Versionen auf mancher Plattform haben, notwendig.

1.10 Perl in Makefiles

```
# als Funktion, diverse Klammerung, um Ende zu finden
# Doppelte Klammerung erlaubt if(){} und mehrzeilig ohne \
$(perl ...) $(makeperl ...) ${perl ...} $((perl ...)) ${{perl ...}}

# Blocks oder Regelaktionen
perl { $x = $y }
    makeperl { $$x = '$(CFLAGS)' }
perl {
    ...
}                           # Muß in 1. Spalte
    perl {{
        ...
    }}                      # Beliebige Spalte
```

Statt `perl`, `makeperl`: `$(Ausdrücke)` werden durch makepp ersetzt, `$$` für Perl `$`.
Obskures Perl Feature erlaubt korrekte Zeilennummern in `eval`:

```
#line 27 "/abc/def/xyz/Makeppfile"
```

1.11 Eingbaute Befehle

Heißt es `echo -n ...` oder `echo "...\c"`? Windows echo ist noch etwas anders...
Wir haben volle Gnu Optionen `-n` oder `--no-newline`, Ende mit `--`
Es gibt: `&cat`, `&chmod`, `&cp`, `&cut`*, `&echo`, `&expr`*, `&grep`*, `&install`, `&ln`, `&mkdir`, `&mv`, `&perl`*, `&preprocess`* (Templatesprache mit Makefile Syntax), `&printf`, `&rm`, `&sed`*, `&sort`*, `&template`* (@XYZ@ Ersetzung), `&touch`, `&uninstall`, `&uniq`*, `&yes`
Statt vieler Subsprachen, *alles Perl.

```
    &cut -c 10-20,-5,25- $(input)          # Klassische Liste, 0-basiert!
    &cut -c 'grep $$_ % 3, 0..99' $(input) # Nicht durch 3 teilbare Spalten
```

Alle Filter können `#line` Direktiven erzeugen.

1.12 Eigene Erweiterungen

All das oben genannte sind einfach Perl Funktionen nach folgendem Schema, das man auch selber definieren darf:

- Anweisung (**s**tatement)

  `sub s_include { }`

- **F**unktion, z.B. $(shell ...)

  `sub f_shell { }`

- Befehl in Regel (**c**ommand), kann auch als Funktion `$(&cat ...)` oder Anweisung verwendet werden. Meist sehr nützlich ist das `frame` Konstrukt, welches nicht nur automatisch die `-v/-verbose` Option bereitstellt, sondern neben ggf. eigenen auch Zugriff auf alle Standardoptionen gibt. Dazu gehören insbesondere Ein-/Ausgabeoptionen wie `-o/-output`, die man braucht da es keine Shell gibt. Dafür kann man Rückgabewerte einzelner Pipe-Stufen als kritisch behandeln und direkt in einer Datei editieren.

```
sub c_chmod {           # Makepp setzt local $0 = 'chmod'
  local @ARGV = @_;
  frame {               # Behandelt Optionen und ggf. Ein-/Ausgabe
    my $mode = oct shift @ARGV;
    perform { chmod $mode, $_ } {\ss}et mode for `$_'"
      for @ARGV;        # Behandelt -v/--verbose, bzw. Fehlermeldung
  };
}
```

- Sonstige Hilfsfunktionen, um Argumente nicht zu überfrachten

```
sub transformation { ... }

&sed &transformation datei
```

1.13 Erweiterung der Arbeitsweise

Die folgenden erweiterbaren Klassenhierarchien bilden von der Befehlszerlegung über das Befehlsverstehen bis hin zur Dateianalyse den Kern von makepp und begründen dessen Mächtigkeit:

- Mpp::ActionParser::MyParser

  erlaubt andere Shells zu unterstützen (`;`, `<`, `>`, ` `...`)

- Mpp::CommandParser::MyParser

  erlaubt neue Befehle zu unterstützen (`gcc -I... -L... -l...`)

- Mpp::Scanner::MyScanner

  erlaubt neue Dateiformate zu verstehen (`#include "datei.h"`)

- Mpp::Signature::MyFile

  erlaubt neue Dateiformate zu verstehen (`/* Kommentar ignorieren */`)

- Mpp::BuildCheck::MyCheck

  erlaubt je Ausgabe zu entscheiden ob neu gebaut wird (z.B. etwas zweifelhaft: sehr teurer Befehl nur nachts)

Gerne nehmen wir Eure Klassen für weitere Sprachen oder Werkzeuge mit auf!

1.14 Yaph – Yet Another Perl Hacker!

So schreibt man keine Makefile:

```
$(phony default): Yet Another Perl Hacker!
    @&echo $(&cat $(inputs 1 3 2 -1))

r=n
Yet: e=a
Perl: e=o
Perl: r=c

Another:
    @&echo jemand -o $(output)

%: :last_chance
    @&expr "$$_ = '$(output)'; tr/!HPYacklr$(e:%=e)/?FnHragh$r$e/; $$_" \
        -o $(output)
```

Hat noch jemand Fragen?

Index

T

U

V

Y

Z