PVS Überlebenshilfe

Hendrik Tews
http://www.askra.de/

Version 13 vom 28.03.2011

1 Überblick

Dieses Dokument enthält die wichtigsten Informationen um mit PVS in den Rechnerkabinetten der Fakultät Informatik arbeiten zu können. Für Hinweise und Anregungen bin ich jederzeit dankbar (`tews@os.inf.tu-dresden.de`).

PVS selbst ist seit Version 4.0 unter der GPL erhältlich. Es läuft allerdings am besten unter dem recht preisintensiven Allegro-Lisp. Für nicht-kommerzielle Anwendungen ist PVS zusammen mit der Laufzeitumgebung von Allegro-Lisp kostenlos auf der PVS-Webseite verfügbar. Man kann PVS aber auch mit CMU Lisp betreiben, dann muß man keine *unfreie* Lizenz akzeptieren. Die aktuelle PVS-Version ist 4.2.

In der Fakultät Informatik ist PVS unter Linux installiert und läßt sich über das Kommando `pvs` (`/usr/bin/pvs`) starten. Die eigentliche PVS-Installation befindet sich in `/opt/pvs`.

Seit Version 3.0 werden die Manuals nicht mehr gewartet, die Neuerungen erscheinen als **Release Notes**. Ansonsten gelten die Manuals von Version 2.4. weiter. Siehe `http://pvs.csl.sri.com/documentation.shtml`, beziehungsweise `/opt/pvs/doc` und `/opt/pvs/old-doc`, für Manuals und Release Notes. Die Manuals sind unterteilt in

System Guide Überblick, Kommandos der Emacs Schnittstelle, Kurzeinführung in Emacs

Language Reference PVS Spezifikationssprache

Prover Guide HOL Beweiskalkül, Beweiskommandos

Auf diese Dokumente werde ich im Folgenden mit [Release Notes], [System Guide], usw. verweisen.

Es gibt auch eine Reihe Tutorien, (siehe PVS Webseite, „Examples and Tutorials“). Ich empfehle

- „PVS Kurzanleitung“ von Prof. F. v. Henke und H. Pfeifer
  `http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/Inferenzsysteme/WS0203/pvs-hilfe.pdf`

mit Ergänzung zu rekursiven Funktionen
(`http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/Inferenzsysteme/WS0203/pvs-hilfe-app.pdf`)

- „A Tutorial Introduction to PVS“,
  `http://www.csl.sri.com/papers/wift-tutorial/`
- „A Less Elementary Tutorial“,
  `http://www.csl.sri.com/papers/csl-95-10/`

PVS ist ein hybrides System. Die Nutzerschnittstelle ist Emacs oder XEmacs. Im Hintergrund läuft ein Allegro-Lisp Prozess. Das Shellskript `pvs` startet alles zusammen. Voreingestellt ist GNU Emacs. XEmacs bekommt man mit `pvs -emacs xemacs`. Mit Emacs Version 23 gibt es verschiedene Probleme, ich benutze für PVS nur Version 22 (`pvs -emacs emacs22`). Auf den Rechnern der Fakultät ist Emacs Version 22 für PVS voreingestellt. Die Emacs Statuszeile enthält immer Informationen darüber, was PVS gerade macht. Für Details und weitere Optionen siehe [System Guide, S. 59].

2 Emacs

Kontakt mit Emacs (oder XEmacs) ist bei der Arbeit mit PVS unvermeidlich. Es reichen aber einfachste Kenntnisse. Siehe zum Beispiel [System Guide, Anhang A] oder die ersten 1000 Zeilen des Emacs Tutoriums (siehe Help-Menü im Emacs). Im Folgenden verwende ich die Emacs Notation für Tastatureingaben. Dabei steht „C–n“ für „Control–n“, „C–x C–n“ für „Control–x“ gefolgt von „Control–n“, „M-x save-buffer“ für „Meta–x“[1] gefolgt von den 11 Zeichen „save-buffer“, gefolgt von Enter, usw.

Menüpunkte „Status → status-proof“ beziehen sich auf das `PVS` Menü in Emacs.

2.1 Emacs im VI-Modus

Mit `M-x viper-mode` aktiviert man die vi-Emulation. Dokumentation gibt es unter dem Info Punkt „Viper“ (Info starten mit `C-h i`, Punkt „Viper“ mit mittlerer Maustaste auswählen oder `m viper`, dann mit `n` (nächste Seite) und `p` (vorherige Seite) navigieren).

3 Arbeiten in PVS

PVS verwaltet Dateien in *Kontexten*. Ein Kontext ist die PVS Sicht auf ein Verzeichnis und enthält alle PVS-Dateien die sich in diesem Verzeichnis befinden. Der gegenwärtige Kontext steht in der Begrüßungsnachricht im Puffer `PVS Welcome`. Der Kontext kann mit `M-x cc` geändert werden.

PVS-Quellkode steht in Ascii Dateien ∗.pvs. Beweisskripte speichert PVS in ∗.prf Dateien. Die ∗.bin Dateien im Unterverzeichnis `pvsbin` enthalten die interne Darstellung

[1] Auf PC's ergibt oft die Alt-Taste Meta, auf Sun Tastaturen gibt es eine richtige Metataste: ⋄.

der ∗.pvs Dateien. Die Datei `.pvscontext` und das `pvsbin` Verzeichnis können gelöscht werden (aber PVS vorher beenden).

Der Arbeitszyklus in PVS sieht so aus:

- Quelltext (Definitionen und Lemmata bzw. Theoreme) in eine `.pvs` Datei schreiben.
- Typecheck (Menü [Parsing and Typechecking → typecheck] oder `C-c C-t`)
- Beweisen der TCC's, siehe Abschnitt 7
- Beweisen von Lemmata und Theoremen, siehe Abschnitt 5
- Abhängigkeiten/Vollständikeit prüfen, siehe Abschnitt 6

4 PVS Quelltext

Eine `.pvs` Datei besteht aus beliebig vielen Theorien. Eine Theorie dekariert Typparameter, definiert Typen, Konstanten und Funktionen und enthält Theoreme, siehe Abbildung 1.

4.1 Kommentare

Kommentare beginnen mit % und reichen bis Zeilenende.

4.2 Importings

Importings (Zeilen 3 und 7) importieren das Material anderer Theorien. Details: [Language Referencs, 6.2]

4.3 Typdeklarationen

Zeile 5 definiert den neuen Typen Self über eine Typgleichung. Beispiel für einen Recordtypen:

```
MapSignature : TYPE =
  [#
    add : [[Self , Key , Data] -> [Self , bool]] ,
    del : [[Self , Key] -> Self] ,
    get : [[Self , Key] -> Lift[Data]]
  #]
```

Details: [Language Referencs, 3.1 und 4].

```
MapFunModel[Key, Data : Type] : Theory
Begin
  IMPORTING Lift

  Self : Type = [Key -> Lift[Data]]                                    5

  IMPORTING MapBasic[Self,Key,Data]

  % adding a new association
  add(m : Self, k : Key, d : Data) : [Self,bool] =                     10
    if up?(m(k))
    then (m, false)
    else (m with [(k) := up(d)], true)
    endif
                                                                       15
  del(m : Self, k : Key) : Self = m With [(k) := bot]

  get(m : Self, k : Key) : Lift[Data] = m(k)

  c : MapSignature = (#                                                20
    add := add,
    del := Lambda(m : Self, k : Key) : del(m,k),
    get := Lambda(m : Self, k : Key) : get(m,k)
  #)
                                                                       25
  fun_new : MapConstructors = (#
    new_map := Lambda(k : Key) : bot
  #)

  fun_model : Lemma MapModel?(c,fun_new)                               30
End MapFunModel
```

Abbildung 1: PVS Beispieltheorie

4.4 Konstanten & Funktionen

Zeile 10 definiert die Funktion add : Self × Key × Data ⟶ Self × bool. Für rekursive Funktionen muss der Terminationsbeweis mit einer Measure–Funktion erfolgen. Beispiel:

```
nth(l :list, (n:below[length(l)])): Recursive T =
  IF n = 0 Then car(l) Else nth(cdr(l), n−1) Endif
Measure length(l)
```

Diese Definition erzeugt bei typecheck die folgende Terminations-TCC:

```
% Termination TCC generated (at line 15, column 31) for nth(cdr(l), n − 1)
  % untried
nth_TCC4: OBLIGATION
  FORALL (l: list[T], (n: below[length[T](l)])):
    NOT n = 0 IMPLIES length[T](cdr[T](l)) < length[T](l);
```

4.5 Lemmata

Zeile 30 deklariert ein Theorem. Anstelle von **Lemma** sind diverse andere Schlüsselworte möglich: **Challenge**, **Claim**, **Conjecture**, **Corollary**, **Fact**, **Formula**, **Law**, **Proposition**, **Sublemma** oder **Theorem**. Hinter diesen Schlüsselwörtern muss ein boolscher Wert stehen.

5 Beweisen

Zum Beweisen eines Lemmas positioniert man den Kursor auf das Lemma (oder die TCC) und startet `M-x prove` oder Menu [Prover Invocation → prove]. Im neuen Fenster kommuniziert man direkt mit dem Allegro Lisp Prozess (deshalb alle Kommandos in Lisp Syntax eingeben und mit einem Klammerpaar umschließen!).

Wichtige Beweiskommandos:

assert Simplifikation. Löst Goals die „offensichtlich“ wahr sind.

flatten boolsche Vereinfachungen

replace Ersetzen. Die linke Seite einer Gleichung in den Voraussetzungen wird durch die rechte Seite ersetzt. Z.B. `(replace -1)`, ersetzen von rechts nach links: `(replace -1 :dir RL)`, ersetzen nur in den Sequenzen 1 und 2: `(replace -1 :fnums (1 2))`

skosimp Führt Metavariablen für universell quantifizierte Formeln ein.

expand Expandieren einer Definition: `(expand "name")`

inst Instanziierung von Quantoren; z.B. instanziieren von Sequenz Nummer -3, die die Form **Forall**(a : A, b : B) : ... hat: `(inst -3 "term-für-a" "term-für-b")`

smash Simplifikation mit Zerlegung in Subgoals.

lemma Einfügen eines Lemmas mit `(lemma "lemma-name")`. Eventuell müssen die Theorieparameter in eckigen Klammern hinter `lemma-name` instanziiert werden.

induct Induktion: `(induct "variable")`.

grind Stärkste Strategie von PVS.

undo Macht das letze Beweiskommando rückgängig. *Nur* das allerletzte undo läßt sich mit `(undo undo)` rückgängig machen.

postpone Weiterschalten auf das nächste offene Subgoal.

quit Beweis abbrechen.

Alle Kommandos nehmen diverse optionale Argumente. Details sind in [Prover Guide]. Online kann man `M-x x-prover-commands` starten und erhält dann die Hilfe mit der rechten Maustaste.

Für viele Kommandos existieren Emacs Keybindings, z.B. für `expand`: Cursor auf Funktionsnamen positionieren und `TAB e`. Details: [System Guide, 3.5.9, Seiten 36, 37].

Ein Beweis kann jederzeit mit `(quit)` abgebrochen werden. Der Beweis kann dann mit einem Namen versehen werden und in der `.prf` Datei gespeichert werden. So kann man unter verschiedenen Namen verschiedene Beweise zu einem Theorem speichern. Eine Übersicht über die vorhandenen Beweise liefert `M-x display-proofs-formula`, siehe [Release Notes, Multiple Proofs].

Ist schon ein Beweis gespeichert, dann kann man beim Start des Beweisers die gespeicherten Kommandos automatisch ablaufen lassen. Man kann auch alle gespeicherten Beweise einer Datei oder Theorie wiederholen: Menü [Prover Invocation → proof-theory] usw.

Das aktuelle Beweisskript wird mit `M-x edit-prove` angezeigt. Mit `C-c C-p s` (`M-x step-prove`) startet man einen Beweis und zeigt das Skript an. Dann kann das Skript mit `TAB 1` und `TAB #` im Einzelschrittmodus abgearbeitet werden.

Ein Tcl/Tk Fenster mit dem aktuellen Beweisbaum erhält man mit
`M-x x-show-current-proof`. Will man einen Beweis in den Einzelschritten genau studieren, startet man ihn mit `C-c C-p X` (`M-x x-step-proof`). Das zeigt den Beweisbaum im Tcl/Tk Fenster und das aktuelle Beweisscript an.

6 Complete & Incomplete

Wenn ein Beweis zwar fertig ist, aber von einem noch unbewiesenen Lemma abhängt, dann ist er *incomplete*. Erst wenn alle Abhängigkeiten vollständig bewiesen sind ist das Lemma *complete*. Nicht vollständig bewiesene Lemmas können falsch sein. Das Kommando `M-x spc` zeigt die Abhängigkeiten.

7 TCC: Typecheck Condition

Beim Typecheck erzeugt PVS hin und wieder sogenannte *typecheck conditions*. Das sind logische Formeln, die mit einem Punkt im Quelltext verbunden sind, den der Typchecker nicht vollständig checken konnte. Nach dem Typchecken sollte man immer erst alle TCC's beweisen oder sich zumindest überzeugen, dass sie beweisbar sind!

Mit dem Kommando `M-x tcp` (Menü [Parsing and Typechecking → typecheck-prove]) versucht PVS alle TCC's automatisch zu beweisen.

Anzeigen aller TCC's: `C-c C-q s` oder `M-x show-tccs`.

Manuell beweisen: Zuerst lässt man sich die TCC's anzeigen. Im neuen TCC Puffer können ganz normal Beweise gestartet werden.

8 Prelude

Der Prelude enthält vordefinierte Typen und Konstanten. Er kann mit `M-x view-prelude-file` angezeigt werden. Im Prelude sucht man am besten mit den normalen Emacs Suchfunktionen (`C-s` und `C-r`). Die Theoreme im Prelude haben echte Beweise, die man (ohne den Kontext zu wechseln) ablaufen lassen kann.

9 Abbrechen & Fehler & Bugs

Es kann passieren, dass das Rewrite System von PVS in eine Schleife gerät oder dass ein Beweis mit grind zu lange dauert. Manchmal trifft man auch auf einen Bug in PVS. Für solche Fälle gibt es verschiedene Methoden, die Situation wieder unter Kontrolle zu bringen:

Beweiskommando unterbrechen Im ∗pvs∗ Puffer: `C-c C-c` oder `M-x pvs-interrupt-subjob`. Das bringt einen in den Allegro List Exception Handler. Dort gibt man `(restore)` ein.

Emacs hängt Mit (wiederholtem) `C-g` unter Kontrolle bringen.

Müll im *pvs* Puffer solange die Ausnahmebehandlung von allegro-Lisp aktiv ist hilft `(restore)`. Ansonsten M-x reset-pvs und Beweis neu beginnen. Fall das auch nicht hilft, den nächsten Punkt wählen.

Neustart, Emacs Session behalten In den `*pvs*` Puffer wechseln und im `Signals` Menü `EOF` auswählen. Das terminiert den pvs-allegro Prozess (könnte wahlweise auch gekillt werden). Kontext und bin-Dateien löschen. Dann einen neuen allegro Prozess mit `M-x pvs` starten.

Neustart Emacs auf normalem Wege beenden, Kontext und bin-Dateien löschen und pvs neustarten.

Letzte Rettung Emacs und pvs-allegro Prozesse killen, Kontext und bin-Dateien löschen und neu starten.