Folie 1

Angewandte Softwareverifikation mit einem interaktiven Theorembeweiser

Dr. Hendrik Tews

http://www.askra.de/

Version 28. März 2011

Script & Software — siehe
http://www.askra.de/lehre/angewandte-verifikation

Version 28. März 2011

Inhaltsverzeichnis

Folie 2

1 Dokumentation / Hilfe

1.1 Logik höherer Ordnung

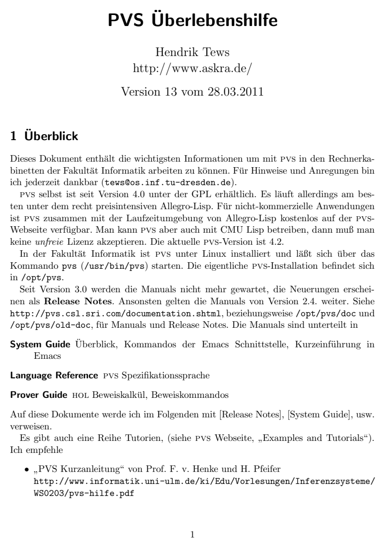
- Dale Miller: *Logic, Higher-order*, Encyclopedia of Artificial Intelligence.

  5-seitige Einführung in Logik höherer Ordnung. Kurz, aber mit weiteren Verweisen.
- J. van Benthem and K. Doets, *Higher-order logic*. Seiten 275 – 329 im Handbook of Philosophical Logic I, herausgegeben von D. Gabbay and F. Guenthner, Reidel, Dordrecht, 1983.
- Daniel Leivant: *Higher-Order Logic* Seiten 230-321 im Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies, herausgegeben von D.M. Gabbay, C.J. Hogger, J.A. Robinson and J. Siekmann, Oxford University Press, 1994.

Folie 3

1.2 PVS

Siehe Webseite!



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`

1

- PVS Überlebenshilfe
- Kurzanleitung von Henke und Pfeifer
- PVS Tutorien

Folie 4

2 Struktur der Quellen

2.1 Quelltextbeispiel

Beispiel

```
more_list_props[T : Type] : Theory
Begin
  take( l : list[T], (n : upto(length(l)))) : Recursive list[T] =
        IF n = 0 Then null
        Else cons( car(l), take( cdr(l), n−1))
        Endif
    Measure n

  takeall : Lemma Forall( l : list[T] ) : take(l, length(l)) = l

  take_length : Lemma Forall( l : list[T], n : nat ) :
        n <= length(l) Implies length(take(l,n)) = n

  nth_take : Lemma Forall( l : list[T], n1, n2 : nat ) :
        n1 <= length(l) And n2 < n1 Implies nth(take(l,n1),n2) = nth(l,n2)
End more_list_props
```

Beispiel

Folie 5

2.2 Dateien

$*$**.pvs** Quelltexte in der Pvs Spezifikationssprache (ASCII)

$*$**.prf** Beweisskripte (ASCII, als Lisp-Konstante)

$*$**.bin** interne Repräsentation (Lispdump; im Unterverzeichnis `pvsbin`)

.pvscontext Statusinformation des Kontexts

~/**.pvsemacs** Konfiguration (Emacs Lisp)

~/**.pvs.lisp** Patches

Kontext = Verzeichnis

- alle zu einen Projekt gehörenden Dateien befinden sich in einem Verzeichnis, dem *aktuellen Kontext*
  (aber siehe Bibliotheken, [Language Reference, S. 34])

Folie 6

2.3 Theorien

Der Quelltext ist in *Theorien* gegliedert (die irgendwie auf die Dateien verteilt sind). Theorien

- haben einen eindeutigen Namen
- enthalten Deklarationen (Typen, Konstanten, Theoreme, und anderes)
- haben Typ- oder Wertparameter

  Parameter werden bei der Benutzung der Theorie mit konkreten Werten oder Typen instanziiert.
- sind durch die Import-Relation Baumartig geordnet. In einer Theorie ist nur das Material der direkt und indirekt importierten Theorien sichbar.

Beispiel

ExTh[Typeparameter_1, Typeparameter_2 : **Type**] : **Theory**
Begin
...
End ExTh

Beispiel

Folie 7

2.4 Bezeichner

Namen bezeichnen Deklarationen (Definitionen, Theoreme, ...), Theorien und Bibliotheken.

Namen können Unterstriche und Fragezeichen (z.B. injective?) enthalten, jedoch keine Bindestriche.

Namen von Theorien müssen innerhalb eines Kontextes eindeutig sein. Theorienamen des PVS-Preludes dürfen nicht wiederverwendet werden.

Der vollständige Name einer Deklaration besteht aus

- dem Theorienamen,
- der Instanziierung (für polymorphe Deklaration) und
- dem Bezeichner der Deklaration.

Theoriename und Instanziierung können (meistens) weggelassen werden.

Beispiele: functions[nat,nat].injective?, injective?[nat,nat], injective?

2.5 Deklarationen

Die meißten Deklarationen folgen der Syntax

name : **Schlüsselwort** ...

Die Grammatik ist nicht LALR(1), manchmal muss man dem Parser aushelfen und eine Deklarationen mit einem Semikolon abschließen (üblicherweise vor Infixdeklarationen).

Folie 8

Uninterpretierter Typ Definiert einen neuen Typnamen über den nichts bekannt ist (kann auch leer sein)

Beispiel

T : **Type**

Beispiel

Interpretierter Typ Definiert Alias für Typausdruck (Typgleichheit ist strukturell)

Beispiel

T : **Type** = nat

Beispiel

Uninterpretierte Konstante Definiert einen neuen Namen für eine Konstante oder Funktion. Über den Namen ist nichts bekannt. (vgl. Existenz TCC [Language Reference, S. 15])

Beispiel

n : nat
f : [nat —> nat]

Beispiel

Folie 9

Interpretierte Konstante Definiert Alias für Term

Beispiel

m : nat = 5
g : [nat —> nat] = **Lambda**(i : nat) : i + 1

Beispiel

Infix Operatoren

Beispiel

<(t1, t2 : T) : bool;
<=(t1, t2 : T) : bool = t1 < t2 **or** t1 = t2;

Beispiel

Die Deklaration *vor* einer infix-Deklaration muss häufig durch ein Semikolon abgeschlossen werden.

Folie 10

Axiom Axiome benötigen keinen Beweis. Sie werden bei der Berechnung der Beweisabhängigkeiten ausgelassen.

Beispiel

n_ax : **Axiom** n > 3 **And** f(n) > n

Beispiel

- n_ax ist ein Name, mit dem das Axiom in Beweisen referenziert werden kann
- **Postulate** kann alternativ zu **Axiom** verwendet werden
- der Körper ist ein Ausdruck vom Typ bool

Theorem Theoreme müssen bewiesen werden

Beispiel

g_lemma : **Lemma** g(1) = 2

Beispiel

Alternative Schlüsselworte: **Challenge**, **Claim**, **Conjecture**, **Corollary**, **Fact**, **Formula**, **Law**, **Proposition**, **Sublemma** und **Theorem**

Folie 11

Variablen sparen Schreibarbeit bei Lambdaabstraktion und universeller Quantifizierung

Beispiel

g_1(i : nat) : nat = i + 1

i : **Var** nat
g_2(i) : nat = i + 1

Beispiel

Importing Importiert die Deklarationen einer anderen Theorie. Die Theorieparameter können dabei instantiiert werden.

Beispiel

Importing ExTh, ..., ExTh[nat,nat]

Beispiel

Bibliotheken sind andere Kontexte, deren Material man verwenden möchte. PVS wird mit zwei Bibliotheken ausgeliefert: Bitvektoren (bitvectors) und endliche Mengen (finite_sets). [Language Reference, S. 34]

Beispiel

Importing finite_sets@top

Beispiel

Folie 12

Deklarationen für Experten

Enumeration zur Definition eines Aufzählungstypes, syn. Zucker, [Language Reference, S. 14]

Macros Konstanten, die automatisch expandiert werden. syn. Zucker, [Language Reference, S. 23]

Induktive Prädikate zur Definition aufzählbarer Prädikate und Relationen, [Language Reference, S. 23]

Judgements zur Beflügelung des Typcheckers (und zur Vermeidung von TCC's), [Language Reference, S. 26]

Conversions Typkonvertierungsfunktionen, die vom Typchecker automatisch eingefügt werden, [Language Reference, S. 29]

Auto-rewrite [Language Reference, S. 34]

Folie 13

3 Spezifikationssprache

Beispiel: Transitive Hülle

- Relationen über S sind Funktionen $R : S \times S \longrightarrow \mathsf{bool}$
- $\mathit{trans?}(R) \quad \stackrel{\mathrm{def}}{=} \quad \forall x, y, z : S \,.\, R(x, y) \wedge R(y, z) \Longrightarrow R(x, z)$
- $R_1 \subseteq R_2 \quad \stackrel{\mathrm{def}}{=} \quad \forall x, y : S \,.\, R_1(x, y) \Longrightarrow R_2(x, y)$
- transitive Hülle:

$$\mathit{closure}(R) \quad \stackrel{\mathrm{def}}{=} \quad \lambda x, y : S \,.\, \forall Q : S \times S \rightarrow \mathsf{bool} \,.\, R \subseteq Q \ \wedge \ \mathit{trans?}(Q) \Longrightarrow Q(x, y)$$

Folie 14

Logik höherer Ordnung: Merkmale

- Formeln *sind* Terme vom Typ bool
- zu zwei Typen σ und τ gibt es auch den Funktionstyp $\sigma \to \tau$
- damit lassen sich Produkte (Records, Tupels) und disjunkte Vereinigung (Unions, Varianten) modellieren
- wenn $t : \tau$ ein Term ist, dann auch $\lambda x : \sigma . t \ : \ \sigma \to \tau$ (Abstraktion)
- wenn $f : \sigma \to \tau$ und $t : \sigma$ Terme sind, dann auch $f\,t \ : \ \tau$ (Applikation)
  Damit enhält Logik höherer Ordnung eine reine (*pure*) funktionale Programmiersprache.
- Quantifizierung ist über alle Typen möglich, insbesondere über Prädikate, Relationen und Funktionen
- es gibt Typkonstruktoren, d.h. Operationen die Typen auf Typen abbilden; z.B. $\mathsf{list}[\sigma]$
- kein vollständiges Beweissystem für das Standardmodell
- Unifikation ist nicht berechenbar
- es gibt Theorembeweiser, z.B. HOL, Pvs, Isabelle/Hol, ...

Folie 15

3.1 Vordefinierte Typen und Operationen

bool	**False**, **True** : bool;
	Not : [bool −> bool];
	And, **OR**, **Implies**, **When**, **IFF** : [bool, bool −> bool] (infix)
real, rat, int, nat	0,1,.... : nat
	+,−,∗,/,~ : [real, real −> real] ... [nat, nat −> nat] (infix)
	<, <=, >, >= : [real, real −> bool] ... [nat, nat −> bool] (infix)
	min, max : [real, real −> real] ... [nat, nat −> nat]
	rem : [posnat −> [int −> nat]]
	ndiv : [int, posnat −> nat]
PRED[X]	$=$ setof[X] $=$ [X − > bool]

Folie 16

list[X]	null	:	list[X]
	cons	:	[X, list[X] —> list[X]]
	length	:	[list[X] —> nat]
	member	:	[X, list[X] —> bool]
	nth	:	[list[X], nat —> X]
	append	:	[list[X], list[X] —> list[X]]
	reverse :	:	[list[X] —> list[X]]
	map	:	[[X —> Y] —> [list[X] —> list[Y]]]
	every	:	[PRED[X], list[X] —> bool]
lift[X]	up	:	[X —> lift[X]]
	bottom	:	lift[X]

(Für weitere list, lift Funktionen siehe Abschnitt 3.8 auf Folie 27)

bit, bvec

EquivClass

Folie 17

3.2 Typkonstruktionen

Records

Typdefinition:

```
record_typ : Type = [#
  feld_1 : typ_1,
  ...
  #]
```

Wertdefinition:

```
value : record_typ = (#
  feld_1 := term_1,
  ...
  #)
```

Feldselektion: feld_1(t) oder t‘feld_1

Folie 18

Kartesisches Produkt [X, Y, ...]

Tupel: (x, y, ...)

Projektionen: Proj_1, Proj_2, ... oder t‘1, t‘2

Funktionen [X —> Y]

Funktionsapplikation: f(t), g(t_1, t_2, ...) (Klammern sind notwendig!)

Funktionswertige Terme: **Lambda**(x,y : typ, ...) : ...

Abkürzung: [X1,...,Xn —> Y] steht für [[X1,...,Xn] —> Y]

In PRED[[X,Y]] oder [X —> [Y —> Z]] oder [X —> [Y,Z]] sind alle Klammern notwendig.

Folie 19

3.3 Ausdrücke

IF bool_term **Then** term **Elsif** bool_term ... **Else** term **Endif**

Cases l **OF**
null : ...,
cons(hd, tl) : ...
EndCases

Forall(x,y : typ, ...) : ...

Exists(x,y : typ, ...) : ...

Let x = term,
y : typ = term
in ...

(term :: typ)

Folie 20

3.4 Syntaktischer Zucker

{x : typ | term_bool} = **Lambda**(x : typ) : term_bool

f **WITH** [(a) := t, ...] = **Lambda**(x : ...) : **If** x = a **Then** t **Elsif** ... **Else** f(x) **Endif**
f **WITH** [‘a := t, ‘a(1)‘feld := ...]
record **WITH** [(feld_1) := ..., ‘feld_2(44) := ...]

Für Fortgeschrittene: Tabellen & **Cond** [Language Reference, S. 52]

Folie 21

3.5 Abstrakte Datentypen

Abstrakte Datentypen definieren einen aus endlich vielen Konstruktoren frei erzeugten Typ. Abstrakte Datentypen sind das Pendant für rekursive Datentypen in Haskell oder SML oder Ocaml. Siehe [Language Reference, S. 71].

——— Beispiel ———

list [T: **Type**]: **Datatype**
Begin
  null: null?
  cons(car : T, cdr : list) : cons?
End list

——— Beispiel ———

——— Beispiel ———

tree[A , B : **Type**] : **DATATYPE**
Begin
  leaf(leaf_acc : A) : leaf?
  node(left: tree, label : B, right : tree) : node?
end tree

——— Beispiel ———

Folie 22

Allgemeines Format für Konstruktoren

cons(acc_1 : T_1, ..., acc_n : T_n) : rec

Dabei sind

cons der Konstruktor

T's die Typen der Konstruktorargumente

acc's die Zugreifer (accessor)

rec der Erkenner (recognizer)

Folie 23

Für Experten: Wechselseitige Rekursion zwischen verschiedenen abstrakten Datentypen gibt es nicht. Stattdessen verwendet man Teiltypen:

——— Beispiel ———

```
HOL : Datatype With Subtypes Term, Form
Begin
  tt : tt? : Form
  imply(ass : Form, conc : Form) : imply? : Form
  as_form(form : Form) : as_form? : Term

  vari( name : nat ) : vari? : Term
  apply( fun : Term, arg : Term) : apply? : Term
  abstr( name : nat, body : Term) : abstr? : Term
  as_term(term : Term) : as_term? : Form
End HOL
```

——— Beispiel ———

Details: [Language Reference, S. 78]

Folie 24

Semantik abstrakter Datentypen

Die Semantik ist die kleinste durch die Konstruktoren frei generierte Menge:

- unter den Konstruktoren abgeschlossen
- enthält nur Elemente, die sich auf endliche Art und Weise mit den Konstruktoren erzeugen lassen (no junk)
- Elemente unterscheiden sich genau dann, wenn sie unterschiedlich erzeugt wurden. (no confusion; Konstruktoren sind injektiv)
- Semantik ist isomorph zur Menge der Terme, die sich aus den Konstruktoren bilden lassen.

Induktion

- jeder Datentyp hat ein eigenes Induktionsprinzip
- gesichert durch freie Erzeugung
- Induktion zerfällt in Definitionsprinzip und Beweisprinzip

Folie 25

3.6 Rekursive Definitionen

Rekursive Funktionen müssen immer total sein. Um das abzusichern muss die Definition mit einem Maß versehen werden.

Beispiel

```
member(x, l): Recursive bool =
  Cases l OF
    null: False,
    cons(hd, tl): x = hd OR member(x, tl)
EndCases
Measure length(l)
```

Beispiel

Im Normalfall reicht es (so wie im Beispiel) anstelle der Maßfunktion einen Ausdruck vom Typ nat anzugeben. Dieser Ausdruck darf auf die Argumente vor **Recursive** bezug nehmen. Details: [Language Reference, S. 19].

Folie 26

3.7 Teiltypen

Für ein Prädikat P : [T —> bool] bezeichnet (P) den Teiltyp von T, der nur die Elemente von P enthält.

Beispiel

i, j: **VAR** int
even?(i): bool = **Exists** j: i = j ∗ 2
even_int: **Nonempty_Type** = (even?) **Containing** 0

Beispiel

Folie 27

3.8 Zugreifer und Erkenner für Abstrakte Datentypen

Für einen Konstruktor cons(acc_1 : T_1, ...) : rec eines Datentyps T werden definiert:

rec : [T —> bool]	Teilmenge der mit cons erzeugten Elemente
acc_1 : [(rec) —> T_1]	Zugriff auf Konstruktorargument

Für list[X] und lift[X] sind das

null? : [list[X] —> bool]
cons? : [list[X] —> bool]
car : [(cons?) —> X]
cdr : [(cons?) —> list[X]]

bottom? : [lift[X] —> bool]
up? : [lift[X] —> bool]
down : [(up?) —> X]

Folie 28

3.9 Abhängige Typen

Typdefinitionen können mit Termen parametrisiert werden.

Beispiel

```
i : VAR int
upfrom(i): NONEMPTY_TYPE = {s: int | s >= i} CONTAINING i
above(i): NONEMPTY_TYPE = {s: int | s > i} CONTAINING i + 1

i : VAR nat
upto(i): NONEMPTY_TYPE = {s: nat | s <= i} CONTAINING i
below(i): TYPE = {s: nat | s < i} % may be empty
```

Beispiel

In komplexen Typausdrücken können Typausdrücke von anderen Elementen abhängen.

Beispiel

```
seq : Type = [#
  size : nat,
  content : [below(size) -> T]
#]
```

Beispiel

Folie 29

In abhängigen Typen kann sich beim *override* der Typ ändern.
Dann ist ->| anstelle von := zu verwenden:

Beispiel

```
append(s : seq, t : T) : seq =
  s WITH ['size := s'size +1,
          'content := s'content WITH [(s'size) |-> t]]
```

Beispiel

Folie 30

3.10 Vorbedingungen für Theorieparameter: Assumptions

Beispiel

Th[...]: **Theory**
Begin
  Assuming
    ass_1 : **Assumption** ...
  EndAssuming
  ...
End Th

Beispiel

- Innerhalb der Theorie Th wirkt ass_1 wie ein Axiom.
- Bei jeder Instanziierung von Th wird ass_1 als TCC Beweisobligation generiert und muss für die spezifische Instanziierung bewiesen werden.

Folie 31

3.11 Warnungen, Nachrichten und Konvertierungen

Beim Typcheck können verschiedene Meldungen erzeugt werden

Warnungen (warnings) werden für bestimmte, häufig fehlerhaft benutzte syntaktische Konstrukte generiert.

Anzeige mit `M-x show-pvs-file-warnings`.

Nachrichten (messages) werden generiert für TCC's, die durch einfache syntaktische Checks als wahr oder ableitbar erkannt und somit nicht generiert werden. Der abgeleitete Typ von Let-Konstanten wird auch als Nachricht generiert.

Anzeige mit `M-x show-pvs-file-messages`.

Konvertierungen (conversions) Konvertierungen werden vom Typchecker automatisch eingefügt, falls die Typen nicht übereinstimmen (so wie in C). Oft verbirgt sich hinter einer Konvertierung ein Typ- oder Spezifikationsfehler.

Anzeige mit `M-x show-pvs-file-conversions`.

Folie 32

3.12 TCC's: Type Correctness Conditions

- Typrichtigkeit ist unentscheidbar (abhängiger Typen, Teiltypen und Assuming's)
- TCC's sind Beweisobligationen, die vom Typchecker generiert werden. Sie sind mit einem Punkt im Quelltext verbunden.
- TCC's müssen vom Nutzer bewiesen werden.
- Bestimmte Programmierfehler äußern sich als unbeweisbare TCC's.
- Zum Beweis: Zuerst automatisch versuchen (mit `M-x tcp`). Dann TCC's anzeigen mit `C-c C-q s`. Für unbewiesene TCC's einen Beweis im TCC Puffer starten.

Folie 33

Beispiel

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

Beispiel

erzeugt die folgenden TCC's:

Beispiel

nth_TCC1: **OBLIGATION**
  FORALL (l: list[T], (n: below[length(l)])): n = 0 **IMPLIES** cons?[T](l);

% Subtype TCC generated (at line 4144, column 38) for l
  % expected type (cons?[T])
 % proved − complete
nth_TCC2: **OBLIGATION**
  FORALL (l: list[T], (n: below[length(l)])): **NOT** n = 0 **IMPLIES** cons?[T](l);

Folie 34

```
% Subtype TCC generated (at line 4144, column 42) for n - 1
    % expected type below[length(cdr(l))]
  % proved - complete
nth_TCC3: OBLIGATION
  FORALL (l: list[T], (n: below[length(l)])):
    NOT n = 0 IMPLIES n - 1 >= 0 AND n - 1 < length(cdr[T](l));

% Termination TCC generated (at line 4144, column 30) for
    % nth(cdr(l), n - 1)
  % proved - complete
nth_TCC4: OBLIGATION
  FORALL (l: list[T], (n: below[length(l)])):
    NOT n = 0 IMPLIES length(cdr[T](l)) < length(l);
```

Beispiel

Folie 35

3.13 Das Auswahlaxiom: choose, epsilon

Siehe Theorie epsilons und die Definition von choose in der Theorie sets!

Die leere Funktion

Beispiel

```
empty_fun : [below(0) -> A] = Lambda(x : below(0)) :
  choose(lambda(a : A) : true)
```

Beispiel

generiert die folgende TCC:

Beispiel

```
empty_fun_TCC1: OBLIGATION
  FORALL (x: below(0)): nonempty?[A](LAMBDA (a: A): TRUE);
```

Beispiel

Siehe Beispiel in `choose.tar.gz`!

4 Beweiser

4.1 Sequenzenkalkül

Sequenz $\Gamma \vdash \Delta$

- Γ und Δ sind Multimengen von Formeln
  $\Gamma = \varphi_1, \varphi_2, \ldots, \varphi_m \qquad \Delta = \psi_1, \psi_2, \ldots, \psi_n$

Folie 36

Bedeutung $\quad \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m \quad \Longrightarrow \quad \psi_1 \vee \psi_2 \vee \cdots \vee \psi_n$

oder $\quad \neg\varphi_1 \vee \neg\varphi_2 \vee \cdots \vee \neg\varphi_m \quad \vee \quad \psi_1 \vee \psi_2 \vee \cdots \vee \psi_n \quad = \quad \mathsf{true}$

oder $\quad \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m \quad \wedge \quad \neg\psi_1 \wedge \neg\psi_2 \wedge \cdots \wedge \neg\psi_n \quad = \quad \mathsf{false}$

ASCII Notation

```
[-1] phi_1
[-2] phi_2
  |-------
{1} psi_1
[2] psi_2
```

Tautologien / Axiome

- $\Gamma \cap \Delta \neq \emptyset$
- $\mathsf{false} \in \Gamma$
- $\mathsf{true} \in \Delta$

Folie 37

Regeln (Beispiele)

$$\frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta} \wedge - L \qquad \frac{\Gamma \vdash \Delta, \varphi \quad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \varphi \wedge \psi} \wedge - R$$

$$\frac{\Gamma, \varphi \vdash \Delta, \psi}{\Gamma \vdash \varphi \Rightarrow \psi, \Delta} \qquad \frac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}$$

4.2 Beweiskommandos

Statistik von 112 .prf Dateien mit 30760 Kommandos

Folie 38

Kommando	prozentualer Anteil	Kommando	prozentualer Anteil
expand	19.30	flatten	2.38
rewrite	9.70	hide-all-but	2.17
skosimp*	9.69	auto-rewrite	1.85
assert	9.54	prop	1.59
hide	4.52	reduce	1.39
inst	4.51	decompose-equality	0.81
inst?	3.95	forward-chain	0.81
use	3.76	lemma	0.77
smash	3.70	subtype-tcc	0.75
grind	3.65	skolem-typepred	0.70
apply-extensionality	2.61	split	0.59
case	2.53	postpone	0.55
replace	2.40	induct	0.48

Folie 39

Kommando	prozentualer Anteil	Kommando	prozentualer Anteil
typepred	0.46	iff	0.06
copy	0.32	stop-rewrite	0.05
expand*	0.32	generalize	0.04
skolem	0.25	bddsimp	0.04
skosimp	0.20	name	0.03
skolem!	0.19	detuple-boundvars	0.03
apply-eta	0.17	cases-tcc	0.03
name-replace	0.16	auto-rewrite-theories	0.02
apply	0.14	assuming-tcc	0.02
replace-eta	0.14	case*	0.02
termination-tcc	0.13	existence-tcc	0.01
lift-if	0.09	bash	0.01
delete	0.09	simplify	0.01
induct-and-simplify	0.08	extensionality	0.00
auto-rewrite-theory	0.07	reveal	0.00

Folie 40

4.2.1 Allgemeines

- Beweiserschnittstelle: Allegro-Lisp Top-Level **(Alles (schön) (klammern) (!))**
- ... in comint-Modus: Navigation mit `M-p`, `M-n`, `M-r`, `C-M-k`, `C-M-backspace`
- Teilformeln kopieren mit *secondary selection* (`mouse-yank-at-point` ändern!) oder mit virtuellem Kursor (Paket `vcursor`)
- Beispiel für optionale und markierte Argumente:
  `(replace fnum &optional ((fnums *) dir hide? actuals?))`

Hilfe

- Prover Guide
- `M-x x-prover-commands` und dann rechte Maustaste
- `C-c C-h c`

Folie 41

4.2.2 Aussagenlogik

assert Vereinfachung mit β-Reduktion, Faktendatenbank und automatischen Rewrites: `TAB a`

smash Vereinfachung und Zerlegung in Subgoals (mit IF-Lifting)

- arbeitet mit BDD's (*binary decision diagrams*)
- manchmal (scheinbare) Subgoal-Vervielfachung
  - vor smash Formeln löschen
  - Alternativen prop oder bddsimp probieren
  - Zerlegung per Hand (lift-if und split oder case)

replace Ersetzung mit Gleichung: `TAB r`

Beispiel: `(replace -1)`, `(replace -1 (-2 2 3) :dir RL)`,
`(replace -1 :actuals T)`

Folie 42

case Fallunterscheidung

Beispiel: `(case "null?(l!1)")`

Varianten: case*, case-replace

flatten Propositionale Vereinfachung: `TAB f`

split Zerlegung in Subgoals: `TAB s`

split zerlegt nur top-level Konjunktionen, Disjunktionen und Implikationen.

lift-if Liften von IF, Cases und Upates: `TAB l`

Folie 43

4.2.3 Quantoren

skosimp$*$ Iterativ skolemisieren und flatten

Skolemisierung ersetzt $\exists$ in den Voraussetzungen und $\forall$ in den Folgerungen mit Metavariablen.

Beispiel: `(skosimp* :preds? t)`

Varianten:

- skosimp (nur eine Ebene)
- `(skolem -1 ("f!1" "n!1" "_"))`

inst? Automatische Instanziierung von Quantoren: `TAB ?`

Beispiel: `(inst? -5)`

Varianten: `(inst -5 "5" "7" "f!1") inst-cp`

generalize Generalisierung von Skolem-Konstanten

Folie 44

4.2.4 Gleichheit

apply-extensionality Gleichheit von komplexen Typen zerlegen: `TAB E`

decompose-equality `TAB =`

Folie 45

4.2.5 Theoreme und Definitionen

expand Definition expandieren: `TAB e`

Beispiel: `(expand "append")`, `(expand "append" 1)`, `(expand "append" 1 3)`

lemma Theorem als Voraussetzung einführen

Beispiel: `(lemma "append_assoc")`, `(lemma "list_props[nat].append_assoc")`

use Lemma mit automatischer Instanziirung

Beispiel: `(use "append_null")`, `(use "append_null" :subst ("l" "l!1"))`, `(use "append_null" :if-match all)`

Folie 46

rewrite Lemma instanziieren und Ersetzung durchführen

Beispiel: `(rewrite "append_null")`,
`(rewrite "append_null" :subst ("l" "l!1") :dir RL)`

forward-chain Lemma instanziieren

Folie 47

4.2.6 Automatisches Beweisen

auto-rewrite Lemma als Auto-Rewrite installieren

Varianten: auto-rewrite-theorie, auto-rewrite-theories

stop-rewrite Auto-Rewrite abschalten

Variante: stop-rewrite-theory

grind Super-duper Strategie: `TAB G`

lädt alle Definitionen und Lemmata als Auto-Rewrite und iteriert dann Vereinfachung, instanziierung und Zerlegung in Subgoals.

Beispiel: `(grind :if-match all)`

reduce Grind ohne zusätzliche Auto-Rewirtes

induct Induktion

Beispiel: `(induct "i")`

Folie 48

4.2.7 Steuerungskommandos

hide Formeln verstecken: `TAB C-h`

Beispiel: `(hide -1 -2 3 4)`

hide-all-but

Beispiel: `(hide-all-but (-1 2))`, `(hide-all-but 1)`

reveal versteckte Formeln einfügen

Beispiel: `(reveal 1)`

Anzeigen der versteckten Formeln mit `M-x show-hidden-formulas`

postpone Beweisziel vertagen: `TAB P`

copy Formel verdoppeln

quit Beweis abbrechen

Folie 49

4.2.8 Strategien

Strategien heißen Kommandos zum Programmieren des Beweisers.

Beispiel: `(apply (branch (grind :if-match nil) ((grind))))`

4.2.9 Weiteres

typepred Einführen von Teiltypbedingungen

Beispiel: `(typepred "l!1")`

name-replace komplexen Ausdruck ersetzen: `TAB N`

Folie 50

5 Aufgaben

1. Korrektheit von Quicksort †
2. Korrektheit von Heapsort †
3. Korrektheit von Bitonic Sort †
4. ausbalancierte binäre Bäume
5. rot-schwarze Bäume (red-black trees)
6. das Lampen-Frosch Problem †
7. Charakterisierung des reflexiven, symmetrischen und transitiven Abschlusses einer Relation mittels Zickzacks
8. Knaster-Tarski-Fixpunkttheorem
9. finale Sequenzen-Coalgebra †
10. Endliche Mengen: Externe Charakterisierung †
11. Endliche Mengen: Interne Charakterisierung

Siehe auch http://www.askra.de/lehre/angewandte-verifikation/#aufgaben, die mit † markierten Aufgaben sind dort ausführlicher formuliert.

Folie 51

5.1 Korrektheit von Quicksort

Aufgabe

1. Programmieren Sie Quicksort für beliebig lange Felder beliegigen Typs in Pvs als rekursive Funktion.
2. Beweisen Sie die Korrektheit der Sortierfunction.

5.2 Korrektheit von Heapsort

Heapsort ist etwas unbekannter als Quicksort, hat aber die gleiche Komplexität von $O(n \log(n))$. Siehe http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/heap/heap.htm.

Aufgabe

1. Programmieren Sie Heapsort für beliebig lange Felder in Pvs als rekursive Funktion.
2. Beweisen Sie die Korrektheit der Sortierfunction.

5.3 Korrektheit von Bitonic Sort

Folie 52

Bitonic Sort ist ein etwas unbekannterer Sortieralgorithmus der Komplexität $O(n\log(n)^2)$. Das besondere an Bitonic Sort ist, dass die Anzahl zu vergleichenden Elemente unabhängig von den Daten sind. Bitonic Sort läßt sich deshalb in Hardware realisieren.
Siehe http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonic.htm.

Aufgabe

1. Programmieren Sie Bitonic Sort für beliebig lange Felder in Pvs als rekursive Funktion.
2. Beweisen Sie die Korrektheit der Sortierfunction.

5.4 Balancierte binäre Bäume

Folie 53

Mit balancierten binären Suchbäumen vermeidet man, dass ein Baum zu einer Liste degeneriert. Perfekt ausbalancierte Bäume sind dabei meist nicht erstrebenswert. Es reicht aus, wenn die Bäume nicht zu unbalanciert sind.

Aufgabe

1. Definieren Sie nahezu balancierte binäre Bäume als predikativen Teiltyp eines geeigneten Typs für binäre Bäume.
2. Definieren Sie Funktionen zum Einfügen, Löschen und Suchen in nahezu balancierten binären Suchbäumen.
3. Beweisen Sie, dass diese Funktionen die entsprechenden Invarianten erhalten (der Baum bleibt ein nahezu balancierter binärer Suchbaum).

Folie 54

5.5 Rot-schwarze Bäume

Rot-schwarze Bäume sind eine einfachere Form balancierter binärer Bäume. In rot-schwarzen Bäumen haben die Knoten ein, zwei oder drei Nachfolgeknoten. Dadurch kann man einen beliebigen Knoten immer leicht erweitern. Implementiert werden rot-schwarze Bäume mit binären Bäumen, in denen die Knoten gefärbt sind (nämlich rot oder schwarz). Jeweils zwei oder drei solche gefärbter Knoten entsprechen dann einem Knoten im abstrakten rot-schwarzen Baum.

Aufgabe

1. Definieren Sie rot-schwarze Bäume als predikativen Teiltyp eines geeigneten Typs für binäre Bäume.
2. Definieren Sie Funktionen zum Einfügen, Löschen und Suchen in rot-schwarzen Suchbäumen.
3. Beweisen Sie, dass diese Funktionen die entsprechenden Invarianten erhalten (der Baum bleibt ein rot-schwarzer Suchbaum).
4. Es ist ausreichend rot-schwarze Bäume über natürlichen Zahlen zu betrachten.

Folie 55

5.6 Das Lampen–Frosch–Problem

Eine unendliche Schlange von Fröschen steht vor einer unendlichen Kette von Lampen mit Ein-Aus-Tastschaltern. Sowohl Frösche als auch Lampen sind mit den positiven natürlichen Zahlen (1,2,3,...) durchnummeriert.

Zu Beginn der Unix-Epoche (0:00 Uhr am 1. Januar 1970 GMT) springt Frosch Nummer 1 auf jeden Schalter und schaltet damit alle Lampen an. Danach springt Frosch Nummer 2 auf jeden zweiten Schalter, so dass nur die Lampen mit ungeraden Nummern brennen. Frosch Nummer drei springt auf jeden dritten Schalter und schaltet dabei zum Beispiel Lampe 3 aus und Lampe 6 an. Alle weiteren Frösche folgen nach und nach, dabei springt Frosch Nummer n immer auf jeden n-ten Schalter.

Aufgabe

1. Welche Lampen brennen, nachdem alle Frösche gesprungen sind?
2. Formalisieren Sie das Lampen-Frosch-Problem in Pvs und beweisen Sie Ihre Lösung!

Folie 56

5.7 Charakterisierung des reflexiven, symmetrischen und transitiven Abschlusses einer Relation mittels Zickzacks

Der reflexive, symmetrische und transitive Abschluss (im folgenden kurz Abschluss) einer Relation R ist die kleinste Relation, die R enthält und reflexiv, symmetrisch und transitiv ist. Den Abschluss kann man wie folgt konstruieren:

Ein *Zickzack in einer Relation* R ist eine endliche Folge $x_1, x_2, \ldots, x_n$ mit wenigstens einem Element, so dass $x_i \, R \, x_{i+1}$ oder $x_{i+1} \, R \, x_i$ gilt (für $i < n$). Zwei Elemente x und y sind im Abschluss genau dann in Relation gesetzt, wenn es einen Zickzack von x nach y gibt.

Aufgabe

1. Formalisieren Sie die Zickzackkonstruktion in Pvs.
2. Beweisen Sie, dass man so tatsächlich den Abschluss einer beliebigen Relation erhält.

Folie 57

5.8 Knaster-Tarski Fixpunkttheorem

Ein vollständiger Verband ist eine geordnete Menge in der Infima (größte untere Schranken) und Suprema (kleinste obere Schranken) für beliebige (insbesondere auch unendlich große) Teilmengen existieren. Eine Funktion f auf einem Verband ist monton, wenn aus $x \leq y$ auch $f(x) \leq f(y)$ folgt. Ein Fixpunkt ist ein Element x mit $f(x) = x$.

Das Knaster-Tarski Fixpunkttheorem besagt, dass die Fixpunkte einer beliebigen monotonen Funktion auf einem vollständigen Verband wieder einen vollständigen Verband auf der gleichen (eingeschränkten) Ordnung bilden (das ist die vollständige Version, praktisch interessiert man sich meist nur für den größten oder kleinsten Fixpunkt).

Aufgabe

1. Formalisieren Sie vollständige Verbände, monotone Funktionen und Fixpunkte für beliebige Ordnungsrelationen (auf beliebigen Trägermengen).
2. Beweisen Sie das Knaster-Tarski Fixpunkttheorem in Pvs.

5.9 Finale Sequenzen-Koalgebra

Eine Sequenzen-Koalgebra über dem Zustandsraum X und dem Alphabet A ist eine Funktion $c : X \longrightarrow (X \times A) \uplus \{\bot\}$. Eine Element $x \in X$ ist dann eine Sequenz. Sie ist leer, falls $c\,x = \bot$. Falls jedoch $c\,x = (x', a)$ dann ist a das erste Element der Sequenz x und x' ist die Restsequenz. Ein Sequenzenhomomorphismus ist eine Abbildung zwischen den Zustandsräumen zweier Sequenzen, die die Struktur beider Sequenzen erhält.

Folie 58

Eine Sequenzen-Koalgebra d ist final, falls es für jede Sequenzenkoalgebra c *genau einen* Sequenzenhomomorphismus von c nach d gibt. (Insbesondere muss es auch genau einen Sequenzenhomomorphismus von d nach d geben.)

Aufgabe

1. Formalisieren Sie Sequenzen-Koalgebren, Sequenzenhomomorphismen und den Begriff der finalen Sequenzen-Koalgebra in PVS.
2. Konstruieren Sie die finale Sequenzen-Koalgebra (für ein beliebiges aber festes Alphabet) und beweisen Sie die Finalität in Pvs.

5.10 Endliche Mengen: Externe Charakterisierung

Eine algebraische Spezifikation für endliche Mengen enthält eine Konstante $nil : set$ und eine Operation $add : set \times A \longrightarrow set$ für die die folgenden Axiome gelten:

$$\begin{aligned} \forall s \in set, a \in A\,.\quad & add(add(s,a),a) = add(s,a) \\ \forall s \in set, a, b \in A\,.\quad & add(add(s,a),b) = add(add(s,b),a) \end{aligned}$$

Ein Modell dieser Spezifikation besteht aus einer Grundmenge X (der Menge aller endlichen Mengen) und entsprechenden Operationen nil_X und add_X, die die Axiome erfüllen.

Folie 59

Ein Mengenhomomorphismus ist eine strukturerhaltende Abbildung zwischen den Grundmengen zweier Modelle. Ein Modell X ist initial, wenn es *genau einen* Mengenhomomorphismus zu jedem anderen Modell gibt (insbesondere gibt es auch nur einen Homomorphismus $X \longrightarrow X$).

Aufgabe

1. Formalisieren Sie die Begriffe des Modells (der Spezifikation endlicher Mengen), des Mengenhomomorphismus und des initialen Modells in PVS.
2. Beweisen Sie, dass das initiale Modell isomorph zu den in PVS definierten endlichen Mengen ist.

Folie 60

5.11 Endliche Mengen: Interne Charakterisierung

Ein algebraisches Modell für endliche Mengen enthält eine Konstante $nil : set$ und eine Operation $add : set \times A \longrightarrow set$. Ein Modell endlicher Mengen kann in folgenden Schritten konstruiert werden:

1. Zuerst bildet man den absolut freien Datentyp T mit den beiden Konstruktoren nil und add.
2. Auf diesem Datentyp betrachtet man die kleinste Kongruenzrelation R, für die folgendes gilt:

$$\begin{array}{rl} \forall t \in T, a \in A\,. & add(add(t,a),a) \;\; R \;\; add(t,a) \\ \forall t \in Ta, b \in A\,. & add(add(t,a),b) \;\; R \;\; add(add(t,b),a) \end{array}$$

3. Jede Äquivalenzklasse von R beschreibt genau eine endliche Menge. Man erhält deshalb das Modell der endlichen Mengen durch eine Faktorisierung: T/R.

Aufgabe

1. Konstruieren Sie den Quotienten T/R in PVS!
2. Konstruieren Sie eine bijektive Abbildung zwischen den in PVS eingebauten endlichen Mengen und T/R!

Index für Begriffe und Beweiskommandos

Index für PVS Funktionen und Typen