

Kurzanleitung zu PVS Teil 4

Rekursive Funktionen

Rekursive Funktionen werden in PVS wie normale Funktionen deklariert, wobei dem Typ des Wertebereichs der Funktion das Schlüsselwort **RECURSIVE** vorangestellt werden muss. Ferner muss nach der Definition des Funktionsrumpfs mit dem Schlüsselwort **MEASURE** ein Maß und – optional – eine dem Schlüsselwort **BY** folgende fundierte Ordnungsrelation angegeben werden.

Rekursive Definitionen unterliegen in PVS einigen Einschränkungen. So ist beispielsweise gegenseitige Rekursion nicht erlaubt. Insbesondere müssen die Funktionen aber total, also für jedes Argument des Definitionsbereichs definiert sein. Um dies sicherzustellen, ist die Angabe eines Maßes nötig. Das Maß ist eine Funktion deren Definitionsbereich mit dem der rekursiven Funktion übereinstimmt¹, und deren Wertebereich gleich dem Definitionsbereich der Ordnungsrelation ist. Falls keine Ordnungsrelation angegeben ist, wird standardmäßig die $<$ -Relation auf den natürlichen Zahlen **nat** angenommen.

Rekursive Funktionen haben im allgemeinen einen Basisfall und einen oder mehrere rekursive Aufrufe. Diese werden üblicherweise mit Hilfe eines **IF-THEN-ELSE**-Ausdrucks unterschieden:

```
fakt(n: nat): RECURSIVE nat = IF n = 0 THEN 1 ELSE n * fakt(n - 1) ENDIF
MEASURE (LAMBDA (x: nat): x)
```

Als abkürzende Schreibweise für die Maßfunktion bietet PVS die Möglichkeit, die **LAMBDA**-Bindung wegzulassen:²

```
fakt(n: nat): RECURSIVE nat = IF n = 0 THEN 1 ELSE n * fakt(n - 1) ENDIF
MEASURE n
```

Dies ist z. B. auch bei mehrstelligen rekursiven Funktionen nützlich, wenn die Rekursion nur bezüglich eines Arguments geschieht:

```
expt(r: real, n: nat): RECURSIVE real = IF n = 0 THEN 1 ELSE r * expt(r, n-1) ENDIF
MEASURE n
```

¹PVS erlaubt hier eine Reihe von Sonderfällen, z. B. für Funktionen in *Curry*-Form oder doppelt-rekursive Aufrufe, die aber in dieser Kurzanleitung nicht behandelt werden. Näheres findet sich in der PVS-Sprachbeschreibung ab Seite 19.

²Allerdings müssen die verwendeten Variablen dann durch die Funktionsargumente gebunden sein; eine Maßangabe **MEASURE x** ist hier also nicht erlaubt, da der Name **x** dann frei vorkommt.

In allen obigen Beispielen kann die Ordnungsrelation auch explizit angegeben werden (allerdings wird dann eine zusätzliche Typkorrektheitsbedingung erzeugt, siehe unten):

```
expt(r:real, n:nat): RECURSIVE real = IF n = 0 THEN 1 ELSE r * expt(r, n-1) ENDIF
    MEASURE n BY <;
```

Im Rahmen der Übungen ist die explizite Angabe einer Ordnungsrelation jedoch allenfalls bei manchen rekursiven Funktionen über induktiven Datentypen (siehe später) notwendig.

Typkorrektheitsbedingungen (TCCs)

Vor dem eigentlichen Beweisen muss jede Theorie auf Typkorrektheit hin überprüft werden. Kommandos: `M-x tc` oder `C-c C-t`. Wegen der hohen Ausdrucksmächtigkeit der PVS-Sprache ist die Typkorrektheit eines Terms in PVS im allgemeinen nicht entscheidbar. Es werden deshalb sogenannte *Typkorrektheitsbedingungen*, engl. *Type Correctness Conditions*, *TCCs*, erzeugt, unter denen die Theorie typkorrekt ist. Diese TCCs sind *Beweisverpflichtungen*: Ein Theorem einer Theorie gilt erst dann als (vollständig) bewiesen, wenn alle TCCs der Theorie gezeigt sind.

Es gibt u.a. folgende Arten von TCCs: Terminations-TCCs, Subtyp-TCCs und Existenz-TCCs. Wir konzentrieren uns zunächst auf die Terminierungs-TCCs; die beiden letzteren Arten werden später erklärt. Die Befehle `M-x show-tccs` bzw. `C-c C-q s` zeigen die für eine Theorie erzeugten TCCs in einem eigenen Buffer an. TCCs werden genau wie andere Lemmata oder Theoreme behandelt und können auf die gewohnte Art bewiesen werden.

Viele TCCs sind sehr simpel und können von PVS automatisch bewiesen werden. Der Befehl `M-x tcp` bzw. `M-x typecheck-prove` veranlasst PVS, so viele der erzeugten TCCs wie möglich mit Hilfe von Standard-Strategien zu beweisen.

Den „Status“ aller Formeln einer Theorie – also TCCs und deklarierte Lemmata und Theoreme – kann man mit Befehl `M-x spt` bzw. `M-x status-proof-theory` ermitteln.

Terminations-TCCs

Die Typprüfung einer rekursiven Funktionsdefinition erzeugt für jede rekursive Aufrufstelle eine Terminations-TCC. Dazu wird die Maßfunktion sowohl auf die Argumente des rekursiven Aufrufs als auch auf die ursprünglichen Argumente der Funktionsdefinition angewandt und über die Ordnungsrelation verglichen. Dabei wird der Kontext, in dem der rekursive Aufruf vorkommt, mit einbezogen.

Im Beispiel der Funktion `fakt` ist der rekursive Aufruf im `ELSE`-Zweig der Fallunterscheidung; daher wird die Negation der Bedingung als Antezent in die Beweisverpflichtung

eingefügt. Es wird somit folgende Terminations-TCC erzeugt:

```
% Termination TCC generated (at ...) for fakt(n - 1)
% untried
fakt_TCC2: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```

Falls in der Definition der rekursiven Funktion explizit eine Ordnungsrelation angegeben wird, muss diese wohl-fundiert auf dem Wertebereich der Maßfunktion sein. Für das Beispiel auf S. 2 würde somit folgende TCC erzeugt werden:

```
% Well-founded TCC generated (at ...) for
% restrict[[real, real], [nat, nat], boolean](<)
% for expt
% unfinished
expt_TCC1: OBLIGATION
well_founded?(LAMBDA (x: nat, y: nat):
    restrict[[real, real], [nat, nat], boolean](<)(x, y));
```

PVS Beweisregeln (Fortsetzung)

Beweisregel: (INDUCT)

(INDUCT *var*) initiiert einen Induktionsbeweis mit der Induktionsvariablen *var* durch Instanzierung des zum Typ von *var* gehörenden Induktionsaxioms. Bei Bedarf kann letzteres auch explizit durch das Schlüsselwort *NAME* angegeben werden.

```
(INDUCT/$ VAR &OPTIONAL (FNUM 1) NAME) :
  Selects an induction scheme according to the type of VAR in FNUM and uses
  formula FNUM to formulate an induction predicate, then simplifies yielding
  base and induction cases. The induction scheme can be explicitly supplied as
  the optional NAME argument.
  (induct "i"):
  If i has type nat and occurs outermost universally quantified in formula
  FNUM, the nat_induction scheme is instantiated with a predicate constructed
  from formula FNUM, and beta-reduced and simplified to yield base and
  induction subcases. If i has type that is a datatype, then the induction
  scheme for that datatype is used by default.
  (induct "x" :fnum 2 :name "below_induction[N]"):
  The below_induction scheme is instantiated with an induction predicate
  constructed from fnum 2.
```

Beispiel: Induktion bei natürlichen Zahlen.

Mit (INDUCT "n" *fnum*) kann Induktion über die Variable *n* geführt werden, wenn die Formel *fnum* die Form (FORALL (n:nat) P(n)) hat. Es werden dabei die zwei Unter-

ziele $P(0)$ und $(\text{FORALL } (n:\text{nat}): P(n) \text{ IMPLIES } P(n+1))$ generiert.

```

beispiel :
  |-----
{1}  FORALL (n:nat): even?(n) AND even?(m!1) => even?(n + m!1)

Rule? (induct "n")
Inducting on n on formula 1,
this yields 2 subgoals:
beispiel.1 :
  |-----
{1}  even?(0) AND even?(m!1) => even?(0 + m!1)

Rule? (postpone)
Postponing beispiel.1.

beispiel.2 :
  |-----
{1}  FORALL j:
      (even?(j) AND even?(m!1) => even?(j + m!1)) IMPLIES
      even?(j + 1) AND even?(m!1) => even?(j + 1 + m!1)
    
```

Beweisregel: (INDUCT-AND-SIMPLIFY)

Einfache Induktionsbeweise können oft mit der vordefinierten Strategie (INDUCT-AND-SIMPLIFY *var*) „erledigt“ werden. Die Strategie wendet zunächst das zu *var* passende Induktionsschema an und vereinfacht daraufhin die entstehenden Unterziele.

```

closed_form :
  |-----
{1}  FORALL (n: nat): sum(n) = (n * (n + 1)) / 2

Rule? (induct-and-simplify "n")
sum rewrites sum(0)
  to 0
sum rewrites sum(1 + j!1)
  to 1 + sum(j!1) + j!1
By induction on n, and by repeatedly rewriting and simplifying,
Q.E.D.
    
```

```
(induct-and-simplify/$ var &optional (fnum 1) name (defs t)
(if-match best) theories rewrites exclude (instantiator inst?)) :
  Inducts on VAR in formula number FNUM using the induction
scheme named NAME, then simplifies using rewrite rules taken
from THEORIES and REWRITES.
DEFS is either
  NIL: defs in the statement are not installed as auto-rewrites
  T: All defs are installed as conditional rewrites
  !: All defs are installed, but with explicit (non-recursive) defs as
    unconditional rewrites
  explicit: Only explicit defs installed as conditional rewrites
  explicit!: Only explicit defs installed as unconditional rewrites.
IF-MATCH is either all, best, or T, as in INST?,
    or NIL meaning don't use INST?.
THEORIES is a list of theories to be rewritten in format expected by
  AUTO-REWRITE-THEORY,
REWRITES is a list of rewrite rules in AUTO-REWRITE format.
EXCLUDE is a list of rewrite rules on which rewriting must be stopped.
The INSTANTIATOR argument can be used to specify use of an alternative
instantiation mechanism. This defaults to the (INST?) strategy.
```

```
(induct-and-simplify "i" :defs ! :theories "real_props"
:rewrites "assoc" :exclude ("div_times" "add_div")):
  If i has type nat, then the natural number induction
scheme is instantiated with a predicate constructed from sequent
formula 1, and the resulting cases are simplified using
definitions in the given sequent (unconditionally expanding
explicit definitions), the rewrites in the prelude theory
real_add_div but excluding div_times and add_div,
and the rewrite rule assoc.
```