

Kurzanleitung zu PVS Teil 5

Subtypen

Alle in PVS definierten Funktionen müssen total sein. Eine Möglichkeit auch partielle Funktionen (wie z. B. die Division) in PVS zu behandeln ist die Funktion durch eine Einschränkung des Definitionsbereichs mit einem Subtyp zu „totalisieren“.

Beispiel: Subtyp der reellen Zahlen ungleich 0.

```
nzreal : NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1
/: [real, nzreal -> real]
```

Die Anwendung einer Funktion, deren Argumente auf einen Subtyp eingeschränkt wurden, führt bei der Typüberprüfung meist zu Typkorrektheitsbedingungen.

Prädikate und Subtypen: Für ein Prädikat P über einem Elementtyp T bezeichnet (P) den zugehörigen Subtyp von T all derjenigen Elemente, die P erfüllen:

```
P : pred[nat] = LAMBDA (n:nat): n >= 100000
BigNumbers : TYPE = (P) % -- Typ der natürlichen Zahlen ab 100000
```

PVS Beweisregeln (Fortsetzung)

Beweisregeln: (TYPEPRED) und (SKOSIMP* :PREDS? T)

(TYPEPRED *exprs*) führt die Subtypeinschränkungen für die Ausdrücke in *exprs* als Antezedentformeln ein.

```
(TYPEPRED &REST EXPRS):
  Extract subtype constraints for EXPRS and add as antecedents.
  Note that subtype constraints are also automatically recorded by
  the decision procedures.
```

Dieser Befehl ist nützlich, um Typbedingungen von Skolemvariablen sichtbar zu machen:

```
|-----  
{1}   FORALL (x: {y: nat | y > 2}): P(x)  
  
Rule? (skosimp*)  
|-----  
{1}   P(x!1)  
  
Rule? (typepred "x!1")  
{-1}  x!1 > 2  
|-----  
[1]   P(x!1)
```

Um Typinformation von quantifizierten Variablen direkt bei der Skolemisierung einzuführen, steht der Befehl (SKOLEM-TYPEPRED) zur Verfügung:

```
(skolem-typepred/$ &optional (fnum *)) :  
    Skolemizes and then introduces type-constraints of the Skolem  
    constants.  
See also SKOLEM!, TYPEPRED.
```

Ebenso kann bei (SKOSIMP*) die Schlüsselwortoption :PREDS? T benutzt werden, um Typprädikate bei der fortgesetzten Skolemisierung mit einzuführen:

```
|-----  
{1}   FORALL (x: {y: nat | y > 2}): P(x)  
  
Rule? (skosimp* :preds? T)  
{-1}  x!1 > 2  
|-----  
[1]   P(x!1)
```

Typkorrektheitsbedingungen (Fortsetzung)

Subtyp-TCCs

Subtyp-TCCs werden dann erzeugt, wenn ein Ausdruck vom Typ T an einer Stelle verwendet wird, wo ein Ausdruck von einem Subtyp S von T erwartet wird.

```
distance(x,y:int) : nat =  
  IF x > y THEN x - y ELSE y - x ENDIF
```

Der Abstand zweier ganzen Zahlen ist offensichtlich nicht-negativ, die Funktion `distance` hat also den Resultattyp `nat`. Der Typ der Subtraktionsfunktion ist jedoch `[real -> real]` und damit sind die Ausdrücke `x-y` und `y-x` zunächst vom Typ `real` und nicht vom deklarierten Subtyp `nat`. Es werden folglich zwei entsprechende Subtyp-TCCs erzeugt:

```
% Subtype TCC generated (at line 6, column 18) for x - y
% untried
distance_TCC1: OBLIGATION FORALL (x, y: int): x > y IMPLIES x - y >= 0;

% Subtype TCC generated (at line 6, column 29) for y - x
% untried
distance_TCC2: OBLIGATION FORALL (x, y: int): NOT x > y IMPLIES y - x >= 0;
```

Existenz-TCCs

Existenz-TCCs werden immer dann erzeugt, wenn Elemente eines Typs *deklariert* werden, von dem PVS nicht ohne weiteres feststellen kann, dass er überhaupt Elemente besitzt (also nicht-leer oder „bewohnt“ ist).

Beispiel: Deklaration einen beliebigen natürlichen Zahl kleiner oder gleich 5.

```
n : {x:nat | x <= 5}
```

Die Typprüfung erzeugt folgende Existenz-TCC:

```
% Existence TCC generated (at line 8, column 2) for n: {x: nat | x <= 5}
% untried
n_TCC1: OBLIGATION EXISTS (x1: {x: nat | x <= 5}): TRUE;
```

Die Wichtigkeit solcher TCCs wird an folgendem Beispiel klar:

```
n : {x:nat | x < 0}

quatsch : THEOREM
  FORALL (P:pred[nat]): P(n)

absoluter_unsinn : THEOREM
  TRUE = FALSE
```

Hier wird eine Konstante deklariert, die es offensichtlich nicht geben kann. Diesen Widerspruch kann man dazu ausnutzen, die beiden „Theoreme“ zu „beweisen“. Die (nicht beweisbare) Existenz-TCC verhindert, dass solche Beweise als zulässig angesehen werden.

Induktive Datentypen

Definition induktiver Datentypen

Zur abstrakten Modellierung natürlicher Zahlen kann folgender induktiver Datentyp definiert werden:

```
Nat : DATATYPE
BEGIN
  zero      : zero?
  succ(pred:Nat) : succ?
END Nat
```

Im allgemeinen hat ein Datentyp folgende Form:

```
adt: DATATYPE [T:TYPE]
BEGIN
  cons1(acc11: T11, ..., acc1n1: T1n1): rec1
  ⋮
  consm(accm1: Tm1, ..., acc1nm: T1nm): recm
END adt
```

wobei cons_i die Konstruktoren, acc_{ij} die Akzessoren und rec_i die Klassifikatoren (recognizer) sind.

Im Gegensatz zu gewöhnlichen Deklarationen können Datentypdeklarationen auch außerhalb von Theorien auftreten (*warum?*). In diesem Fall wird die entsprechende ADT-Theorie auch in eine eigene PVS-Datei geschrieben. Wird ein Datentyp innerhalb einer Theorie deklariert, so kann die ADT-Theorie mit `M-x ppe` (*pretty-print-expanded*) betrachtet werden.

Rekursive Funktionen über induktiven Datentypen

Eine Additionsfunktion über dem induktiven Datentyp `Nat` kann durch Fallunterscheidung mit Hilfe eines `IF-THEN-ELSE`-Ausdrucks definiert werden:

```
add(n,m:Nat) : RECURSIVE Nat =
  IF m = zero THEN n
  ELSE succ(add(n,pred(m)))
ENDIF
MEASURE m BY <<
```

Im `ELSE`-Zweig dieser Definition wird auf das Vorgängerelement von `m` mit einem expli-

ziten Aufruf der Selektor-Funktion `pred` verwiesen.

Für Funktionen auf Elementen eines induktiven Datentyps (und nur für diese!) bietet PVS jedoch auch eine syntaktische Variante zum IF-THEN-ELSE-Ausdruck an: den CASES-Ausdruck, mit dessen Hilfe ein einfaches *pattern matching* erreicht werden kann. Die obige Additionsfunktion lässt sich mit CASES auch wie folgt definieren:

```
add(n,m:Nat) : RECURSIVE Nat =  
  CASES m OF  
    zero      : n,  
    succ(x)   : succ(add(n,x))  
  ENDCASES  
  MEASURE m BY <<
```

Hier werden die verschiedenen Fälle für die einzelnen Konstruktoren des Datentyps nacheinander aufgezählt (durch Kommata getrennt). Es ist darüberhinaus möglich, einen ELSE-Zweig zu benutzen, der dann alle noch nicht betrachteten Konstruktoren abdeckt. Zu beachten gilt, dass die Fallunterscheidung nur bezüglich einer Variablen (und nicht bezüglich eines Ausdrucks) geführt werden kann, deren Typ darüberhinaus durch eine Datentyp-Deklaration gegeben sein muss.

Ebenso dürfen in den Mustern der einzelnen Fälle als Argumente von Konstruktoren (z. B. `x` im Fall `succ`) nur *Variablen* (und keine Ausdrücke) verwendet werden. Der Gültigkeitsbereich der so eingeführten Variablen ist auf den jeweiligen Zweig beschränkt.

Näheres zum CASES-Ausdruck findet sich in der PVS-Sprachbeschreibung.