

Einschub: Prädikative Untertypen in PVS

PVS erlaubt es, einen Untertyp eines existierenden Typs zu definieren durch Angabe eines Prädikats, das die dem Untertyp zugrundeliegende Teilmenge bestimmt:

```
p: [T -> bool]
sub: TYPE = { x:T | p(x) }
```

Die Definition eines Untertyps ist immer relativ zu einem Basistyp ('Obertyp').
sub ist der Typ all derjenigen Elemente des Typs T, die das Prädikat p erfüllen.

Alternative, abkürzende Schreibweise:

(p) bezeichnet denselben Untertyp wie $\{x:T \mid p(x)\}$

d.h. *jedes* Prädikat kann zur Definition eines Untertyps herangezogen werden.

Untertypen können in PVS wie Typen benutzt werden.

Sie sind insbesondere nützlich bei der Spezifikation von Funktionen, die andernfalls nur partiell wären.

Untertypen (2)

Beispiele:

`nznat: TYPE = { n: nat | n > 0 }`

`evennat: TYPE = { n: nat | even(n) } [= (even)]`

`non_zero(x: num): boolean = (x /= 0)`

`div(x: num, y: (non_zero)): num = ...`

Untertypen (3)

Konsequenzen der Einführung von prädikativen Untertypen:

- Typüberprüfung ist nicht mehr rein statisch (“syntaktisch”) durchführbar, sondern ist im allgemeinen Fall unentscheidbar und erfordert Beweise; ‘Typ’ wird zu einem *semantischen* Begriff.
 \rightsquigarrow Integration von Typüberprüfung und Beweisen in PVS
- Es werden *Typkorrektheitsbedingungen* (TCCs) erzeugt, sobald z.B.
 - die Existenz eines Elements im Untertyp behauptet wird;
 - als Funktionsargument ein Element eines Untertyps gefordert wird, für das aktuelle Argument aber nur Zugehörigkeit zum Basistyp sichergestellt ist.

TCCs müssen wie andere Lemmata, auf denen ein Beweis aufbaut, bewiesen werden, damit ein Beweis vollständig ist (bzw. damit eine Spezifikation als konsistent akzeptiert werden kann).

Parametrisierte Typen in PVS

Typen können von Werten abhängen, d.h. parametrisiert sein.
Beispiele aus dem prelude:

```
m: VAR nat
upto(m): NONEMPTY_TYPE = {n: nat | n <= m} CONTAINING m
below(m): TYPE = {n: nat | n < m}
```

auch mit mehreren Parametern:

```
i, j: VAR int
subrange(i, j): TYPE = {k: int | i <= k AND k <= j}
```

Ein subrange-Typ kann offensichtlich auch leer sein.

Einschub II: Abhängige Typen in PVS

Ein Typ ist ein *abhängiger Typ* (engl. *dependent types*), wenn eine Komponente vom *Wert* einer vorherigen Komponente abhängt.

Beispiel:

```
datum: TYPE = [# jj: jahr, mm: monat,  
               tt: {t: posnat | t <= tage(mm,jj) } #]
```

Der Typ der Tag-Komponente `tt` des Datums hängt von den Werten für das Jahr und den Monat ab, hier ausgedrückt durch eine (geeignet definierte) Funktion `tage`.

Abhängige Typen können beitragen zur Verschärfung bzw. genaueren Spezifikation von Definitions- und Wertebereich von Funktionen:

~> Vermeidung von Unbestimmtheit (“Partialität”) von Funktionen

Abhängige Typen (2)

Eine endliche Folge der Länge n aus Elementen des Typs T kann repräsentiert werden durch eine Funktion über dem Index-Bereich $\{0, \dots, n-1\}$; als Typ in PVS:

```
seq: [below(n) -> T]
```

Typ endlicher Folgen *beliebiger* Länge (engl. *finite sequences*):

```
finseq: TYPE = [# length: nat, seq: [below(length) -> T] #]
```

Beispiel: Zugriff auf n -tes Element einer Folge

```
nth(s:finseq, n: below(s'length)): T = s'seq(n)
```

Typüberprüfung stellt sicher, daß nicht versucht wird, auf nicht-existierende Elemente der Folge zuzugreifen.

↪ Einschränkung des Definitionsbereichs der Funktion `nth` durch geeignete Typisierung

Abhängige Typen (3)

Operationen auf endlichen Folgen: z.B. Verkettung

```
o(fs1, fs2: finseq): finseq =  
    LET l1    = fs1'length,  
        lsum = l1 + fs2'length  
    IN (# length := lsum,  
        seq      := (LAMBDA (n:below(lsum)):  
                      IF n < l1  
                        THEN fs1'seq(n)  
                        ELSE fs2'seq(n-l1)  
                      ENDIF) #);
```

Typkorrektheit der rechten Seite?

Abhängige Typen (4)

Weiteres Beispiel:

```
mod: THEORY
  i,k: VAR int
  j:   VAR nonzero_int
  ...
  mod(i,j): { k | abs(k) < abs(j) } = ...
```

- Wertebereich ist eingeschränkt in dem Sinn, dass der *Rumpf* der Funktionsdefinition einen Wert bestimmen muß, der dem angegebenen Typ angehört.
- Bei Aufruf der Funktion `mod` wird dann garantiert, daß der Funktionswert die angegebene Eigenschaft hat.
- Typinformation für `mod` bzw. einen Aufruf `mod(i,j)` kann dem Beweiser übermittelt werden mit dem Kommando `TYPEPRED`.

Parametrisierte Theorien

Eine Theorie kann parametrisiert werden mit

- Typen
- Konstanten (einschließlich Funktionskonstanten)

Theorie-Parameter ergeben zum Teil denselben Effekt wie *polymorphe* oder *generische Typen*

Beispiel: generischer Typ von endlichen Folgen über einem beliebigen Elementtyp T

```
finseqs [T: TYPE]: THEORIE
BEGIN
  finseq: TYPE = [# length: nat, seq: [below(length) -> T] #]
  ...
END
```

Parameter können im Theorie-Rumpf genauso benutzt werden wie “normal” deklarierte Typen und Konstanten.

Parametrisierte Theorien (2)

In PVS ist die Parametrisierung von `finseq` nur über den Umweg der parametrisierten Theorie möglich, nicht direkt.

Streng genommen gibt es in PVS keine *Typ-Variablen*, in dem Sinne wie es Variable für Grundobjekte, Funktionen, Prädikate usw. gibt; es gibt auch keine *Quantifizierung* über Typen.

Dies erfordert eine mächtigere und komplexere Logik: eine *Typentheorie*.

Instanziierung einer parametrisierten Theorie: durch Import

~> wird in Kürze behandelt