Rechnergestützte Beweissysteme

Typen und Typinferenz

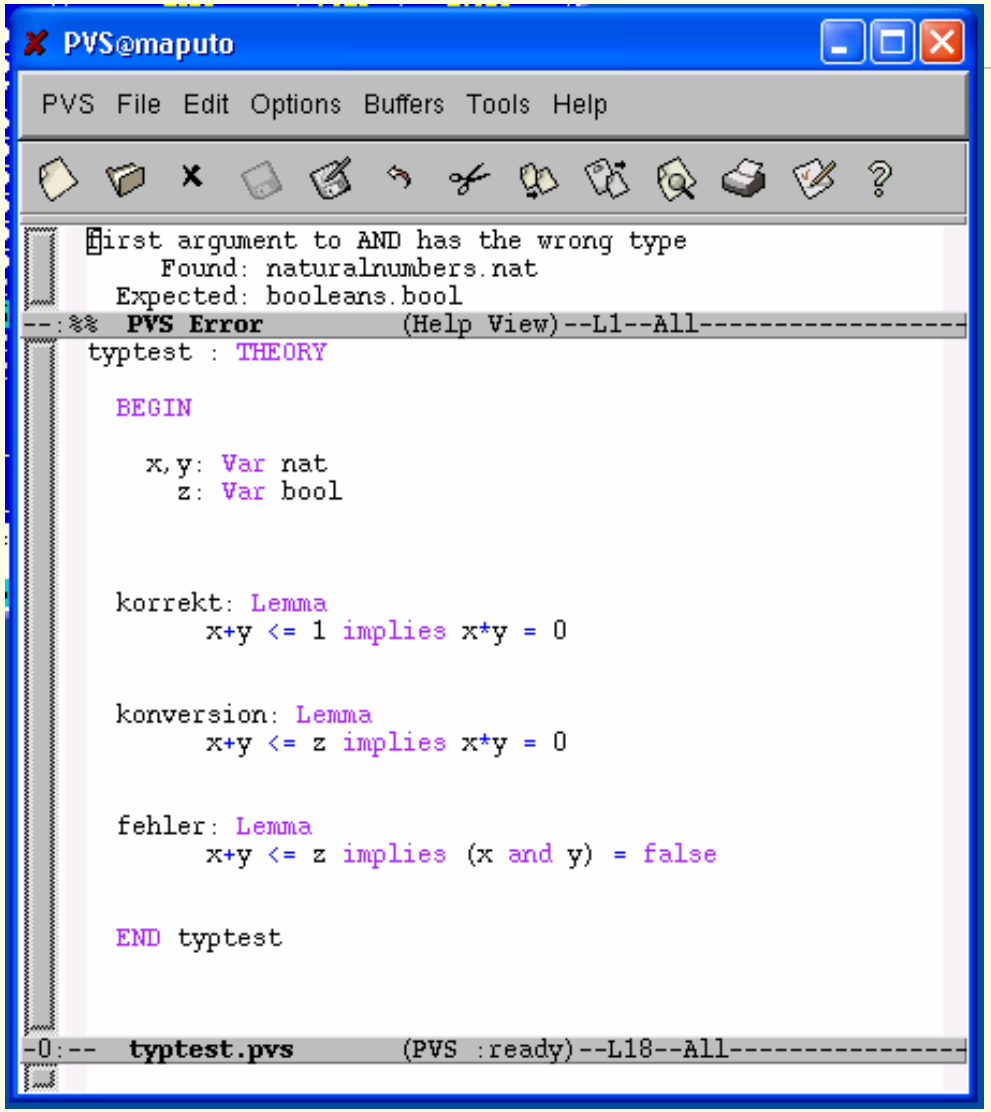
Typen und Typinferenz

1. Kontexte
   - Variablen
   - Kontexte
   - Terme
   - Ausdrücke
   - Typüberprüfung und Konversion
2. Typkonstruktionen mit Kontextregeln
   - Uninterpretierte Typen
   - Aufzählungstypen
   - Funktionstypen
   - Rekursive Funktionen
   - Terminierungsmaße
   - Prädikative Subtypen
   - Abhängige Typen (dependent types)
3. Typüberprüfung
   - Konsequenzen des Typsystems
   - Grenzen der Typüberprüfung
   - TCCs - Type Correctness Conditions

Typen und Typinferenz

1. Kontexte
   - Variablen
   - Kontexte
   - Terme
   - Ausdrücke
   - Typüberprüfung und Konversion
2. Typkonstruktionen mit Kontextregeln
   - Uninterpretierte Typen
   - Aufzählungstypen
   - Funktionstypen
   - Rekursive Funktionen
   - Terminierungsmaße
   - Prädikative Subtypen
   - Abhängige Typen (dependent types)
3. Typüberprüfung
   - Konsequenzen des Typsystems
   - Grenzen der Typüberprüfung
   - TCCs - Type Correctness Conditions

Typüberprüfung in PVS



- PVS Spezifikationen dürfen
  - keine syntaktischen Fehler und
  - keine Typfehler haben
- Im Kontext der gezeigten Variablendeklaration ist
  - das erste Beispiel typkorrekt
  - das zweite Beispiel nur dank der automatischen Konversion `bool` → `nat`
  - das dritte Beispiel hat einen Typfehler.
- Wie prüft PVS das nach ?

Variablen und Kontexte

- Eine Variable x kann in unterschiedlichen *Kontexten* verschiedene Werte unterschiedlicher Typen bezeichnen – das gilt für PVS wie für Programmiersprachen:

```
complex test(float x, float y) : complex;
{     ...   ... x+y/2 ...   ...
}
```

```
{  int x;
     ..... x+y/2 ... .
}
```

```
VAR x, y : color;
BEGIN
    ... x+y/2 ...
END
```

```
FORALL (x:nat, y: stack):
    ...   push(x,y)   ...
```

Kontexte werden in den verschiedenen Programmier-sprachen durch Blöcke definiert:

In diesem Java-Kontext ist x vom Typ *float*, *x+y/2* ebenfalls.

In diesem C-Kontext ist x vom Typ *int* und *x+y/2* vom Typ *float*.

In diesem PASCAL-Kontext ist x vom Typ *color* und x+y/2 ist *nicht typisierbar* → Typfehler.

In diesem PVS-Kontext ist x vom Typ *nat*, und *push(x,y)* vom Typ *stack*.

Kontexte - formal

- Formal definieren wir einen *Kontext* als eine Abbildung von Variablen („identifier") in Typen.

  $$K = \{ x_1:\tau_1,\ x_2:\tau_2,\ \ldots\ ,\ x_n:\tau_n \}$$

  soll bedeuten:

  $$K(x_1) = \tau_1,\quad K(x_2) = \tau_2,\ \ldots \text{ etc.}$$

  Wir lesen dies als

  „Im Kontext K hat x_1 Typ τ_1, x_2 hat Typ τ_2, ...etc."

- Je nach Kontext ergibt sich auch der Typ eines Terms. Wir schreiben

  $$K \triangleright t:\tau$$

  falls im Kontext K der Term t den Typ τ hat.

  Beispiel: Nach den Typregeln von PASCAL gilt im Kontext

  $K = \{ x_1$`:integer,  `x_2`:real` $\}$:

  $K \triangleright x_1+x_1$`:integer,` aber $K \triangleright$ `x1+x2:real`

Terme

- Sei K ein Kontext und seien Funktionssymbole $(f_j)_{j\in J}$ mit ihren Typen $f_i : \tau_{i1}\times\tau_{i2}\times\ldots\times\tau_{in} \to \tau_i$ gegeben.

- Die folgenden Schlussregeln definieren die Syntax von Termen und gleichzeitig deren Typ in einem Kontext K:

$$\frac{K(x)=\tau}{K \rhd x:\tau}$$

„Jede Variable vom Typ τ ist auch ein Term vom Typ τ"

$$\frac{K \rhd t_1:\tau_1\ ,\ \ldots\ ,\ K \rhd t_n : \tau_n\ ,\qquad f : \tau_1\times\tau_2\times\ldots\times\tau_n \to \tau}{K \rhd\ f(t_1\ ,\ \ldots\ ,t_n) : \tau}$$

- Was besagt der Spezialfall: n=0 ?
- Beispiele für Terme: x, z, x*(y+z), push(x+y,push(3,empty)), ...

IF und =

- Für jeden Typ τ hat PVS automatisch die Terme
  - IF _ THEN _ ELSE : [bool, τ, τ, $\rightarrow \tau$]
  - = : [τ, τ, $\rightarrow$ bool]
- mit den Typ-Regeln

$$\frac{K \rhd t_1 : \texttt{bool}\ ,\ K \rhd t_2 : \tau,\ K \rhd t_3 : \tau}{K \rhd (\ \text{IF}\ t_1\ \text{THEN}\ t_2\ \text{ELSE}\ t_3\ ) : \tau}$$

$$\frac{K \rhd t_1 : \tau,\quad K \rhd t_2 : \tau}{K \rhd t_1 = t_2 : \texttt{bool}}$$

Atomare Ausdrücke

- Atomare Ausdrücke sind die elementaren Ausdrücke der Prädikatenlogik. Ein atomarer Ausdruck

$$R(t_1, \ldots, t_n)$$

besagt, dass die Terme $t_1, \ldots, t_n$ in der Relation R stehen. Dabei muss natürlich die Typinformation:

$$R : [\tau_1 \times \tau_2 \ldots \times \tau_n \to \text{bool}]$$

berücksichtigt (und von PVS überprüft) werden:

$$\frac{K \rhd t_1 : \tau_1 , \ \ldots \ , K \rhd t_n : \tau_n , \qquad R : [\tau_1 \times \tau_2 \times \ldots \times \tau_n \to \text{bool}]}{K \rhd R(t_1 , \ldots , t_n) : \text{bool}}$$

Beispiele für atomare Ausdrücke:
x+y < z-3, top(s) = 17+y, empty(push(sum(17),s)).

Offene Ausdrücke

- Offene Ausdrücke (*open expressions*) erhält man aus den atomaren Ausdrücken durch Kombination mit den logischen Konnektoren $\wedge,\vee,\neg,\rightarrow,\leftrightarrow$:

$$\frac{K \rhd E : bool}{K \rhd \neg E : bool}$$

$$\frac{K \rhd E_1 : bool \;,\; K \rhd E_2 : bool \;,\; \bullet \in \{\wedge,\vee,\rightarrow,\leftrightarrow\}}{K \rhd E_1 \bullet E_2 : bool}$$

$$\frac{}{K \rhd True : bool} \qquad \frac{}{K \rhd False : bool}$$

Beispiel : $x*x < 7 \wedge x > 2 \rightarrow x \leq 1$

Gebundene Ausdrücke

- Gebundene Ausdrücke sind solche, die mit den Quantoren $\forall$ und $\exists$ gebildet werden.

$$\frac{K, x{:}\tau \ \triangleright E{:}bool}{K \ \triangleright \exists(x{:}\tau).E : bool}$$

$$\frac{K, x{:}\tau \ \triangleright E{:}bool}{K \ \triangleright \forall(x{:}\tau).E : bool}$$

- E bezeichnet man als den Kern des Ausdruckes und x als die gebundene Variable.
- Liest man die Regel rückwärts, so wird der Kontext erweitert.
- Unter $K, x{:}\tau$ versteht man den Kontext der um die Variable x erweitert wird. Genauer gilt für eine beliebige Variable v:

$$K,x{:}\tau\ (v) = \text{IF } v \equiv x \text{ then } \tau \text{ ELSE } K(v)$$

Typüberprüfung

- Typkorrektheit wird durch Rückwärts-Anwendung der Kontext-Regeln überprüft.
- Beispiel:
  - FORALL (x,y:nat): FORALL (z:bool): x+y<=z IMPLIES x*y=0
    - ist typkorrekte logische Formel, falls im leeren Kontext { } gilt:
  - { } ▷ (∀ x:nat.∀ y:nat.∀z:bool. x + y ≤ z ⇒ x*y=0) *: bool*
    - mit der ∀-Typ-Regel reicht es zu zeigen:
  - {x:nat} ▷ (∀ y:nat.∀z:bool. x + y ≤ z ⇒ x*y=0) *: bool*
  - {y:nat x:nat} ▷ (∀z:bool. x + y ≤ z ⇒ x*y=0) *: bool*
  - {z:bool, y:nat x:nat} ▷ (x + y ≤ z ⇒ x*y=0) *: bool*
    - wegen ⇒ : [`bool,bool →bool`] führt das auf die beiden Fragen
  - {z:bool, y:nat x:nat} ▷ (x + y ≤ z) *: bool* und
    {z:bool, y:nat x:nat} ▷ (x*y=0) *: bool*

Typüberprüfung und Konversion

- {z:bool, y:nat x:nat} ▷ (x + y ≤ z) *: bool*
  ist zu überprüfen
  - *wegen* ≤ : [nat,nat → bool]
  - wird das zu
  - {z:bool, y:nat x:nat} ▷ (x + y) *: nat*
    - und
  - {z:bool, y:nat x:nat} ▷ (z) *: nat*

- Diese führen zu den drei Verpflichtungen
  - {z:bool, y:nat, x:nat} ▷ x *: nat*
  - {z:bool, y:nat, x:nat} ▷ y *: nat*
  - {z:bool, y:nat, x:nat} ▷ z *: nat*

- Die ersten sind trivial, die letzte führt zu einem Fehler, es sei denn, es ist eine Konversion (hier von bool zu nat) definiert.

Typkonversion

$$\frac{K \triangleright v : bool}{K \triangleright v : nat}$$

Typen und Typinferenz

1. Kontexte
   - Variablen
   - Kontexte
   - Terme
   - Ausdrücke
   - Typüberprüfung und Konversion
2. Typkonstruktionen mit Kontextregeln
   - Uninterpretierte Typen
   - Aufzählungstypen
   - Funktionstypen
   - Rekursive Funktionen
   - Terminierungsmaße
   - Prädikative Subtypen
   - Abhängige Typen (dependent types)
3. Typüberprüfung
   - Konsequenzen des Typsystems
   - Grenzen der Typüberprüfung
   - TCCs - Type Correctness Conditions

Eingebaute PVS-Typen

- Eingebaute Typen sind `bool`, `nat`, `int`, `rat`, `real`
  - `bool` : die Wahrheitswerte TRUE, FALSE
  - `nat`: die natürlichen Zahlen 0,1,2,3,...
  - `int`: die ganzen Zahlen ... -3, -2, -1, 0, 1, 2, 3, ...
  - `rat`: die rationalen Zahlen: 1, $\frac{1}{2}$, 2/3, -22/7, ...
  - `real`: die reellen Zahlen: 1, 0.7777, $\sqrt{2}$, π, ...

Uninterpretierte PVS-Typen

- Uninterpretierte Typen werden durch das Schlüsselwort `TYPE` eingeführt

  Beispiel: `H:Type  % H sei eine beliebige Menge`

- Nichtleere Typen durch `TYPE+`

  Beispiel: `G:Type+  % G sei eine nichtleere Menge`

  - Wenn ein Typ eine Konstante haben soll, muss er nichtleer sein

    Beispiel: Gruppen

```
G   : TYPE+
e   : G
o   : [G,G->G]
inv : [G->G]
```

Uninterpretierte Teil-Typen

- Uninterpretierte Teil-Typen können zu jedem vorhandenen Typ erklärt werden.
- Sie werden durch die Schlüsselwort `TYPE FROM` eingeführt

Beispiele:

```
H : Type            % H sei eine beliebige Menge
S : TYPE FROM H     % S ist ein Teiltyp von H
F : TYPE FROM nat   % F sei eine Teilmenge von nat
```

- Nichtleere Teil-Typen durch `NONEMPTY_TYPE FROM`

Beispiele:

```
G : Type+                      % G sei eine nichtleere Menge
H : NONEMPTY_TYPE FROM G       % H sei nichtleere Teilmenge von G
M : NONEMPTY_TYPE FROM nat     % M sei nichtleere Menge nat.Zahlen
```

Aufzählungstypen

- Aufzählungstypen werden durch Angabe der möglichen Werte definiert

Beispiele:

```
Farbe:TYPE = { rot,gruen,blau }
```

`bool` ist ein Aufzählungstyp mit den Werten TRUE, FALSE

Typregeln für *Farbe* : `TYPE { rot,gruen,blau }`

$$\frac{}{K \triangleright \text{rot} : \text{Farbe}} \qquad \frac{}{K \triangleright \text{gruen} : \text{Farbe}} \qquad \frac{}{K \triangleright \text{blau} : \text{Farbe}}$$

Kartesisches Produkt

- $T_1, T_2, \ldots, T_n$ seien beliebige Typen
- Kartesisches Produkt (tuple type):

  `[T1,T2,...,Tn]`
- Bedeutung: $T_1 \times T_2 \times \ldots \times T_n$
- Konstruktor: `( , … , )`
- Selektoren: `proj_1, …, proj_n` alternativ: `´1, …, ´n`
- Beispiele:
  - `Rational : TYPE = [nat,nat]  % ein Rational ist ein Paar von nats`
  - `(3,5) : Rational  % (3,5) ist ein Rational`
  - `IF x=(3,5) THEN proj_1(x)=3`
  - `(3,5)´2 = 5`

Kontextregeln

Kontextregeln aus Rational : `TYPE = [nat,nat]`:

$$\frac{K \triangleright \texttt{p:nat, q:nat}}{K \triangleright (p,q) : \texttt{Rational}}$$

Konstruktor

$$\frac{K \triangleright \texttt{v:Rational}}{K \triangleright \text{proj\_1}(v) : \texttt{nat}}$$

$$\frac{K \triangleright \texttt{v:Rational}}{K \triangleright \text{proj\_2}(v) : \texttt{nat}}$$

Selektoren

Noch ein Kartesisches Produkt

- *Recordtypen*

  $[\#\ name_1 : T_1,\ name_2 : T_2,\ \ldots\ ,\ name_n : T_n\ \#]$

- Bedeutung: $T_1 \times T_2 \times \ldots \times T_n$ mit sebstgewählten Namen für den Zugriff auf die Komponenten

- Konstruktor: `(# , … , #)`
- Selektoren: $name_1, \ldots, name_n$ als einstellige Funktionen

- Beispiele:
  - `Rational: TYPE = [# zaehler:nat, nenner: nat]`
  - `(# zaehler := 3, nenner := 5 #) = (# nenner := 5, zaehler := 3 #)`
  - `IF x = (# zaehler := 3, nenner := 5 #) THEN zaehler(x) = 3`
  - `nenner((# zaehler := 3, nenner := 5 #))=5`

Funktionstypen

- Funktionstypen (function type)
  - $[T_1 \rightarrow T_2]$
- Bedeutung: $[T_1 \rightarrow T_2]$ ist die Menge aller Abbildungen von T_1 nach T_2. Abbildungen mit mehreren Argumenten in Verbindung mit Tupeln:

  $[T_1, T_1 \rightarrow T_2]$ ist Kurzform für $[[T_1, T_1] \rightarrow T_2]$
  und identisch mit ARRAY$[T_1, T_1 \rightarrow T_2]$
- Konstruktor: (LAMBDA (v : T_1) : t),
  - v Variable, t Term vom Typ T_2
- Selektor: Funktionsapplikation f(t)
  - f : $[T_1 \rightarrow T_2]$ und t Term vom Typ T_1
- Beispiele:
  - `(LAMBDA (x:nat): x+1) : [nat -> nat]`
  - `(LAMBDA (x,y:nat): x*x+y*y) : [nat,nat -> nat]`
  - `IF f=(LAMBDA (x,y:nat): x*x+y*y) THEN f(2,3)= 13`
  - `(LAMBDA (n:nat) even(n))(7) = FALSE`

Kontextregeln für Funktionstypen

Kontextregeln für f : TYPE = [T_1 -> T_2] :

$$\frac{K,\ x{:}T_1 \rhd\ expr : T_2}{K \rhd (\ LAMBDA\ (x{:}T_1) : expr\ ) : [T_1 \to T_2]}$$

Abstraktion

$$\frac{K \rhd p{:}\ [T_1 \to T_2],\ K \rhd t{:}T_1}{K \rhd p(t) : T_2}$$

Applikation

Statt `f:[X -> Z] = (Lambda (x:X):<expr> )`
kann man in PVS auch schreiben :

`f(x:X): Z = <expr>`

Kontextregel für binäre Operation

Zu jedem Funktionstyp ergibt sich eine entsprechende Kontextregel
Beispiel: binOp : TYPE = `[X,Y -> Z]` :

$$\frac{K,\ x{:}X,\ y{:}Y \ \triangleright\ \ expr : Z}{K \ \triangleright\ (\ \mathit{LAMBDA}\ (x{:}X,y{:}Y) : expr\ \ ) : binOp}$$

Konstruktor

$$\frac{K \ \triangleright\ p{:}\ binOp\ \ ,\ \ K \ \triangleright\ t_1{:}X,\ \ K \ \triangleright\ t_2{:}Y}{K \ \triangleright\ p\ (t_1,t_2) : Z}$$

Selektor

Statt **`f :[X,Y -> Z] = (Lambda (x:X, y:Y): <expr> )`**
kann man in PVS auch schreiben :

`f(x:X, y:Y): Z = <expr>`

Funktionsdefinitionen

- Funktionsdefinitionen erzeugen Elemente von Funktionstypen:

```
isPrim (m:nat) : bool =
    FORALL (p,q:nat): p*q=n IMPLIES p=m OR q=m
```

  - steht für

```
istPrim : [nat -> bool] =
    (LAMBDA (m:nat):
          FORALL (p,q:nat): p*q=n IMPLIES p=m OR q=m
```

  - und analog

```
cdr(l:list[X]) : list[X] =
   CASES l OF
     NULL          :  NIL
     cons(a,rest)  : rest
   ENDCASES
```

  - steht für

```
cdr : [ list[X] -> list[X] ] =
  (LAMBDA (l:list[X]):
    CASES l OF NULL: NIL cons(a,rest):rest  ENDCASES
```

- Mit der Konstruktor-Regel für Funktionstypen folgt, dass die Objekte in den entspr. Funktionsräumen liegen

Beweis für

cdr:[list[X]->list[X]]

```
cdr : [ list[X] -> list[X] ] =
      (LAMBDA (l:list[X]):
        CASES l OF
            NULL          :NIL
            cons(a,rest)  :rest
        ENDCASES
```

Konstruktorregel für `[list[X]-> list[X]]` **und für** `CASES` **führt zu**

K, `l:list[X]` $\triangleright$ IF `l=Null` THEN `Nil:list[X]`
K, `l:list[X]` $\triangleright$ IF `l=cons(a,rest)` THEN `rest:list[X]`

$K \triangleright$ `cdr : [list[X] -> list[X]`

Rekursive Funktionen

- Probleme entstehen mit rekursiven Definitionen:

```
sum(n:nat) : nat =
   IF (n=0) THEN 0 ELSE n+sum(n-1) ENDIF
```

steht für

```
sum = (LAMBDA (n:nat):
 IF (n=0) THEN 0 ELSE n+sum(n-1) ENDIF)
```

- Die Frage, ob `sum:[nat -> nat]` gilt, führt auf die Frage:

  `sum(n-1): nat` ?

- dies wiederum führt auf die beiden Fragen, ob
  - `n:nat ≠ 0 IMPLIES (n-1):nat` und
  - `sum:[nat -> nat]` ?

Rekursive Funktionen

- Wenn die Funktion f im Ausdruck expr vorkommt
- und wenn man zeigen kann, dass die zugehörige Maßfunktion m eine Schranke für die Anzahl der Aufrufe von f angibt
  - (Die Berechnung von f(n) ruft insgesamt höchstens m(n)-mal f auf) :
- dann gilt

$$\frac{K,\ \mathtt{x}:T_1,\ \mathtt{f}:[T_1 \to T_2] \ \triangleright\ \text{expr} : T_2}{K \ \triangleright\ (\ \text{LAMBDA}\ (x : T_1) : \text{expr}\ ) : [T_1 \to T_2]}$$

Konstruktor

Prädikative Subtypen

- Für jedes Prädikat `p:[T→bool]` ist

  `(p) = {x∈T | p(x)}`

  ein prädikativer Subtyp von `T`

  Beispiele: `nonEmptyStack: TYPE = (nonempty?)`

```
primZahl:TYPE = {n:nat|FORALL(x,y:nat):
                          x*y=n IMPLIES x=n OR y=n}
```

das ist identisch mit

```
( (LAMBDA(n:nat):FORALL(x,y:nat):
          x*y=n IMPLIES x=n OR y=n ) )
```

Kontextregeln für Subtypen

- Für das Prädikat `p:[T→bool]` und den Subtyp

  `(p) = {x∈T | p(x)}`

  benötigt man die Regeln

$$\frac{K \triangleright t : T, \quad K \triangleright p(t)=true}{K \triangleright t : (p)}$$

Konstruktor

$$\frac{K \triangleright t : (p), \quad K \triangleright p : T \rightarrow bool}{K \triangleright t : T}$$

Vererbung

Geht man von einem einelementigen Subtyp True von bool aus, so hat man statt $K \triangleright p(t)=true$ die übliche Notation $K \triangleright p(t) : True$.

Abhängige Typen (dependent types)

- Dependent Type – eine Typkomponente ist abhängig von den Werten anderer Komponenten

Beispiele:

```
interval:TYPE =[# unten:real,oben:{r:real|unten<r}#]

date:TYPE = [ y,m:nat, {d:nat| d <= days(m,y)} ]
```

Typen und Typinferenz

1. Kontexte
   - Variablen
   - Kontexte
   - Terme
   - Ausdrücke
   - Typüberprüfung und Konversion
2. Typkonstruktionen mit Kontextregeln
   - Uninterpretierte Typen
   - Aufzählungstypen
   - Funktionstypen
   - Rekursive Funktionen
   - Terminierungsmaße
   - Prädikative Subtypen
   - Abhängige Typen (dependent types)
3. Typüberprüfung
   - Konsequenzen des Typsystems
   - Grenzen der Typüberprüfung
   - TCCs - Type Correctness Conditions

Konsequenzen des Typsystems

- Programmiersprachen haben meist ein einfaches Typsystem. Ob ein Typfehler auftreten kann oder nicht, kann statisch (zur Compilezeit) geprüft werden

- *Typüberprüfung* bei PVS kann beliebig kompliziert werden
  - Grund: dependent types und subtypes

- Beispiel:

```
magicDate(year:nat): date =
        IF year mod 4 = 0
                THEN (year,2,29)
                ELSE (year,2,28)
```

  muss einen Typfehler erzeugen, weil nicht jedes durch 4 teilbare Jahr einen Schalttag hat.

Schwierigkeit der Typprüfung

- Viele Programmiersprachen haben ein recht simples Typsystem
  - keine Funktionstypen
  - keine prädikative Subtypen
  - keine abhängige Typen
- Je reichhaltiger ein Typsystem, desto schwieriger ist die Typüberprüfung
  - Beispiel:
    - `f(n:nat): RECURSIVE nat = IF n mod 2 = 0 THEN f(n div 2) ELSE f(3*n+1) ENDIF`
    - `Ist f : [nat -> nat]  ???`
- Schon prädikative Subtypen machen es unentscheidbar
  - Beispiel:
    - `p(n:nat) = n>2 AND EXISTS (x,y,z:nat): x^n+y^n=z^n`
    - Ist (p) leer ???

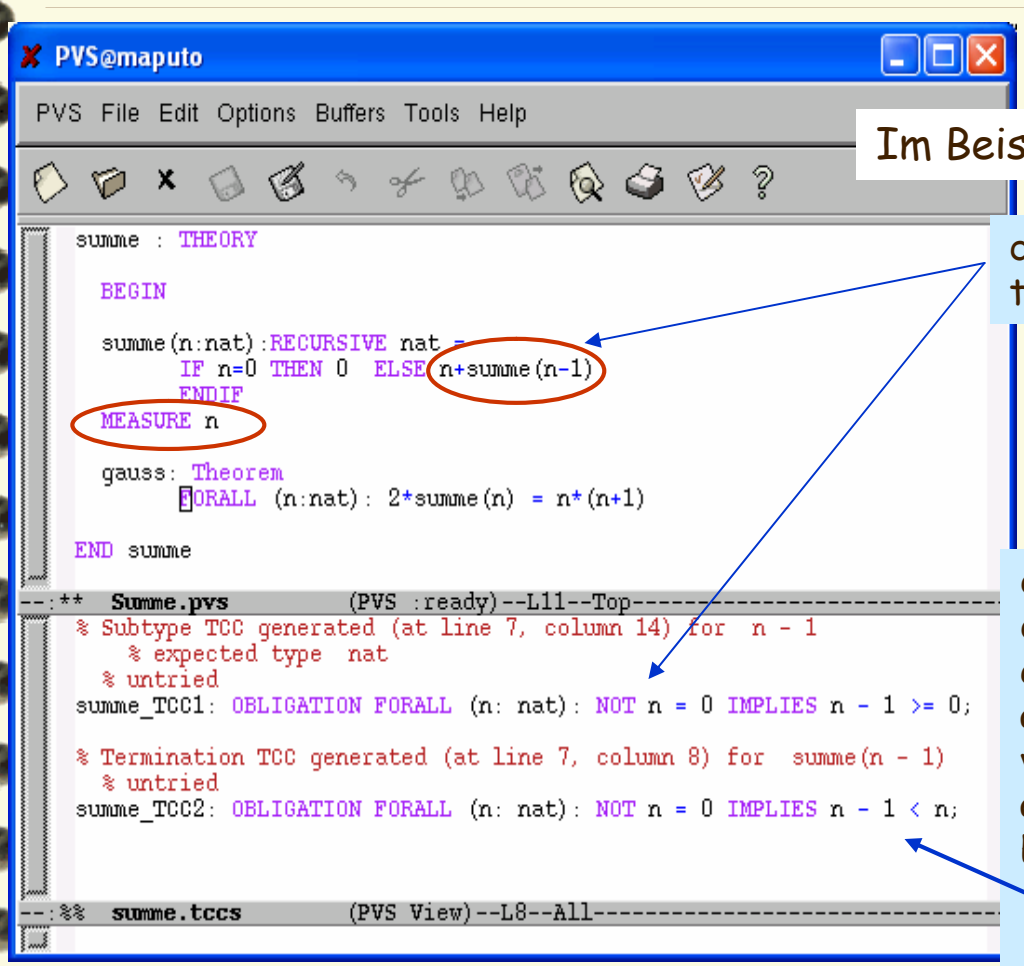
TCC – Type Correctness Conditions

- Typüberprüfung in PVS ist nicht entscheidbar
- PVS versucht einen Typcheck soweit möglich zu erledigen,
  - es können aber offene Fragen bleiben
  - diese werden als *type correctness conditions* (TCC) bezeichnet
  - sie müssen vom PVS-Benutzer mit Hilfe des Beweisers gezeigt werden
- Der PVS-Befehl

  M-x tc

  initiiert eine Typüberprüfung und versucht die TCCs zu beweisen.
  - Gelegentlich bleiben einige übrig
  - Erst wenn diese auch bewiesen sind, ist die Spezifikation o.k.

Entstehung von TCCs



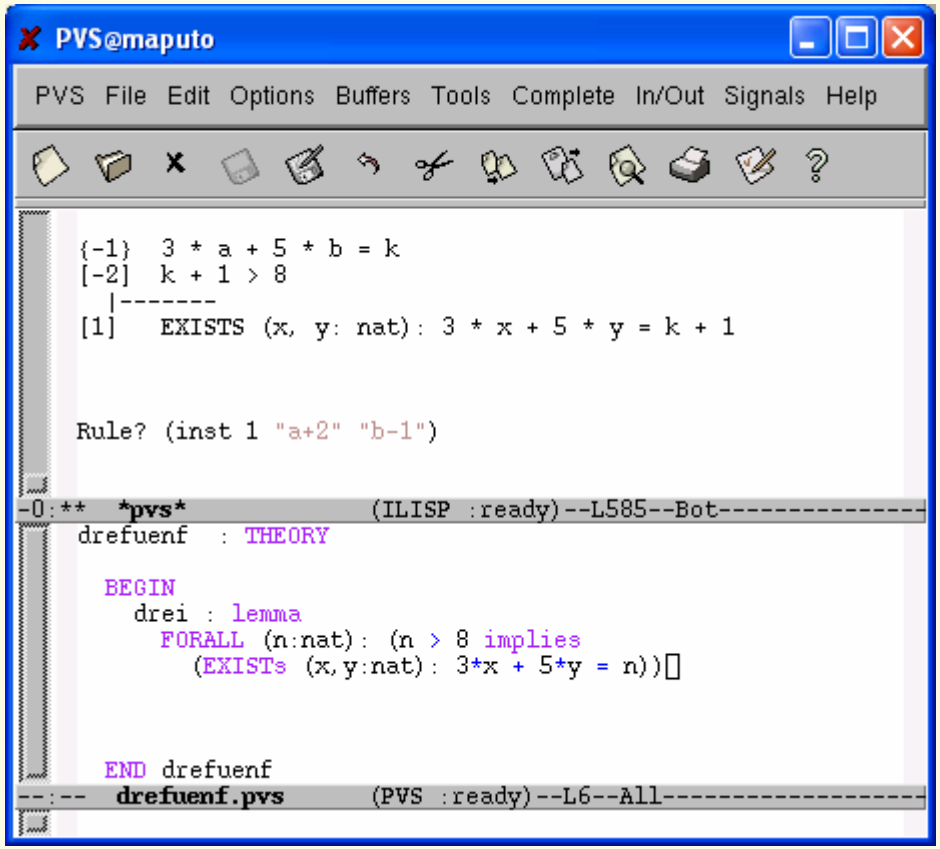
Im Beispiel muss sichergestellt werden,

dass das Argument von summe tatsächlich eine nat. Zahl ist:

dass summe eine totale Funktion ist, d.h., immer terminiert. Dazu dient ein Maß in Form eines arithmetischen Ausdrucks t, der von den Parametern der Funktion abhängt. Hier: t(n) = n.
Es muss überprüft werden, dass
- t(n) immer ≥ 0
- t wird bei jedem Aufruf echt kleiner.

Entstehung von TCCs

- TCCs können auch während eines Beweises entstehen
  - bei einer Instantiierung einer gebundenen Variablen durch einen Ausdruck. Es muss überprüft werden, dass der Ausdruck den richtigen Typ hat.



In diesem Falle entsteht die zusätzliche Obligation: