Overview of the Type System of PVS

Lecture Interactive Proof Tools

Summer Term 2003

Hans de Nivelle, Patrick Maier

Types in Programming Languages

- provide structure
  - provide specification which can be checked (type checking),
  - provide specification which documents,
- are naturally associated with functions,
- ensure certain correctness properties (e. g., strong normalization in typed lambda calculus)

Types in Logic

- Many simple logics (e. g., propositional logics) are untyped. Even if there are functions (e. g., in first-order logic) the logic can be untyped (yet there exist typed versions of FOL).
- Many higher-order logics are typed, for much the same reasons than programming languages.
- Specification provided on the type level is unneccessary because it is expressible in the (untyped) logic itself. But type specification is convenient (especially when it is automatically checkable).
- In *type theory* there is a close link between types and logic. (However, PVS is not based on type theory and does not exploit this.)

Types in PVS

In PVS, a **type** is a set of values.

Questions:

- Which values contains a given type?
- How can one construct these values?
- How can one construct the types?
- What purpose serve the types?

Constructing Types from Types

Basic builtin types:

- `bool`: two-element set of truth values `TRUE`, `FALSE`
- `nat`: countable set of natural numbers $0, 1, 2, \ldots$
- `int`: countable set of integers $\ldots, -2, -1, 0, 1, 2, \ldots$
- `rat`: countable set of rational numbers, e. g., $1, 0.5, \frac{1}{3}, \ldots$
- `real`: uncountable set of real numbers, e. g., $1, 0.33, \frac{1}{\sqrt{3}}, \pi, \ldots$

Constructing Types from Types

Tuples:

- type: $\texttt{[T}_1, \ldots, \texttt{T}_n\texttt{]}$ $\quad (n \geq 1)$
- meaning: cartesian product $\texttt{T}_1 \times \cdots \times \texttt{T}_n$
- constructor: `( )`
- destructors: $`1, \ldots, `n$
- examples:

```
[int] = int
(7, TRUE) : [int, bool]
(7, TRUE)`1 = 7
(7, TRUE)`2 = TRUE
```

Constructing Types from Types

Records: tuples with named components.

- type: $\texttt{[\#a}_1 : \texttt{T}_1, \ldots, \texttt{a}_n : \texttt{T}_n\texttt{\#]}$ $\quad (n \geq 1)$
- meaning: cartesian product $\texttt{T}_1 \times \cdots \times \texttt{T}_n$
- constructor: `(# #)`
- destructors: $\texttt{a}_1, \ldots, \texttt{a}_n$
- examples:

```
(# left := 7, right := TRUE #) : [# left: int, right: bool #]
(# left := 7, right := TRUE #) = (# right := TRUE, left := 7 #)
left((# left := 7, right := TRUE #)) = 7
right((# left := 7, right := TRUE #)) = TRUE
```

Constructing Types from Types

Functions:

- type: `[`$\mathtt{T}_1, \ldots, \mathtt{T}_n$`->T]` $\quad (n \geq 1)$
- meaning: function space $\mathtt{T}^{\mathtt{T}_1 \times \cdots \times \mathtt{T}_n}$
- `[`$\mathtt{T}_1, \ldots, \mathtt{T}_n$`->T]` $=$ `[[`$\mathtt{T}_1, \ldots, \mathtt{T}_n$`]->T]`
- constructors: `LAMBDA`, `RECURSIVE`
- destructor: function application
- examples:

```
(LAMBDA x: x+1) : [int -> int]
(LAMBDA x: x+1)(7) = 7 + 1
```

Constructing Types from Types

Predicates: functions of type `[T -> bool]`

- examples:

```
pos(i: int): bool = i > 0

even(n: nat): RECURSIVE bool =
  IF n = 0 THEN TRUE
    ELSIF n = 1 THEN FALSE
      ELSE even(n-2)
  ENDIF
  MEASURE n

prime(n: nat): bool
AXIOM prime(n) = FORALL m: divides(m,n) IMPLIES m = 1 OR m = n
```

Constructing Types from Types

Predicates subtypes:

- type: `{x: T | p(x)}` or `(p)` where `p: [T -> bool]`
- meaning: subset $\{x \in \mathtt{T} \mid \mathtt{p}(x)\}$ of `T`
- Predicate subtypes may be empty.
- Type checking is undecidable.
- Terms may have multiple types.
- examples:

  `2: (pos)`

  `2: (even)`

  `2: (prime)`

Constructing Types from Types

Dependent types: generalize tuple and function types.

- *Types*: set of all types
- dependent pair: $[x : A, B(x)]$ where $B : A \longrightarrow \mathit{Types}$
- meaning: dependent product $\{\langle x, y\rangle \in A \times \bigcup \mathit{Types} \mid y \in B(x)\}$
- dependent function: $[x : A \longrightarrow B(x)]$ where $B : A \longrightarrow \mathit{Types}$
- meaning: dependent function space $\{f \in (\bigcup \mathit{Types})^A \mid \forall x \in A : f(x) \in B(x)\}$
- In PVS: For all x, $B(x)$ must be a predicate subtype of some common supertype.

Constructing Types from Types

Dependent types example:

Informal specification: `take` takes a list `xs` and a natural number `n` and returns the prefix of length `n` of `xs`.

- No type dependencies:
  - `take: [list[int], nat -> list[int]]`
- Dependencies between arguments:
  - `take(xs: list[int], n: {i: nat | i <= length(xs)}): list[int]`
  - `take: [xs: list[int], {i: nat | i <= length(xs)} -> list[int]]`
- Dependencies between arguments and result:
  - ```
    take: [xs: list[int], n: {i: nat | i <= length(xs)}
        -> {zs: list[int] | length(zs) = n}]
    ```
    ```

Constructing Types from Nothing

- So far: Construct types from basic types via the type constructors (dependent) product, (dependent) function space and predicate subtype.
- Now: Synthesize an **inductive type** from no other type.

Inductive Types

Examples:

- One element type (constructor `one`, recognizer `one?`)

```
unit: DATATYPE
BEGIN
  one: one?
END unit
```

- Two element type (constructors `tt`, `ff`, recognizers `tt?`, `ff?`)

```
mybool: DATATYPE
BEGIN
  tt: tt?
  ff: ff?
END mybool
```

- General enumeration types (n constructors, n recognizers)

Inductive Types

Examples:

- Natural numbers (constructors **`zero`**, **`succ`**, destructor **`pred`**, recognizers **`zero?`**, **`succ?`**)

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

- meaning: countable set of distinct **constructor terms** `zero`, `succ(zero)`, `succ(succ(zero))`, . . .
- PVS generates file `mynat_adt.pvs` containing axioms defining
  - interaction between constructors, destructors and recognizers,
  - the subterm relation and generic measure functions

Inductive Types

Examples:

- Lists of booleans (constructors `null`, `cons`, destructors `car`, `cdr`, recognizers `emptylist?`, `nonemptylist?`)

```
boollist: DATATYPE
BEGIN
  null: emptylist?
  cons(car: bool, cdr: boollist): nonemptylist?
END boollist
```

- meaning: countable set of distinct constructor terms `null`, `cons(FALSE,null)`, `cons(TRUE,null)`, `cons(FALSE,cons(FALSE,null))`, `cons(FALSE,cons(TRUE,null))`, `cons(TRUE,cons(FALSE,null))`, `cons(TRUE,cons(TRUE,null))`, ...

Inductive Types

Examples:

- Terms for integers (constructors `zero`, `succ`, `pred`)

  ```
  myint_term: DATATYPE
  BEGIN
    zero : zero?
    succ(predofsucc: myint_term): succ?
    pred(succofpred: myint_term): pred?
  END myint_term
  ```

- meaning of `myint_term`: set of constructor terms `zero`, `succ(zero)`, `pred(zero)`, `succ(succ(zero))`, `succ(pred(zero))`, `pred(succ(zero))`, `pred(pred(zero))`, ...

Inductive Types

Example continued:

- Identify semantically equivalent terms through axioms

```
myint: THEORY
BEGIN
  IMPORTING myint_term
  myint: TYPE = myint_term
  pred_succ: AXIOM FORALL (x: myint): pred(succ(x)) = x
  succ_pred: AXIOM FORALL (x: myint): succ(pred(x)) = x
END myint
```

- meaning of `myint`: set of equivalence classes of constructor terms

Structural Recursion over Inductive Types

Examples:

- Recursion over an enumeration type

```
not(b: mybool): mybool =
  CASES b OF
    tt: ff,
    ff: tt
  ENDCASES
```

- Recursion over an infinite inductive type

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

Declaring Uninterpreted Types

Uninterpreted types are essentially **free type variables**. They can stand for any type which is expressible in PVS.

- declaration of uninterpreted type $\mathtt{T}_1$:

  $\mathtt{T}_1$`: TYPE`

- nonempty uninterpreted types $\mathtt{T}_2$ and $\mathtt{T}_3$:

  $\mathtt{T}_2$`: NONEMPTY_TYPE`

  $\mathtt{T}_3$`: TYPE+`

- uninterpreted subtypes $\mathtt{T}_4$ and $\mathtt{T}_5$ of $\mathtt{T}_1$:

  $\mathtt{T}_4$`: TYPE FROM `$\mathtt{T}_1$

  $\mathtt{T}_4$`: NONEMPTY_TYPE FROM `$\mathtt{T}_1$

References

1. PVS Language Reference. Chapter 4 and 8.
2. Sam Owre, Natarajan Shankar. The Formal Semantics of PVS. Technical Report CSL-97-2R. 1997.