Types and Typechecking

Outline

- Paradoxes
- Hierarchy of Types
- Sets, Sorts & Types
- Typechecking
- Application:
  Correctness of Tabular Specifications
- Summary

Paradoxes

paradox - par·a·dox Etymology: From Greek paradoxon, from neuter of *paradoxos* contrary to expectation,

- a self-contradictory statement that at first seems true
- an argument that apparently derives self-contradictory conclusions by valid deduction from acceptable premises

Paradoxes result from self-referential statements. E.g.

Liar's paradox: The Cretan Epimenides said

> All Cretans are liars, and all statements made by Cretans are lies.

Russel's paradox

Bertrand Russel showed that naive set theory was inconsistent with the following paradox:

Let $\mathbf{P}$ be the set of all sets that do not contain themselves as an element.

$$\mathbf{P} = \{\mathbf{Q} \in sets | \mathbf{Q} \notin \mathbf{Q}\}$$

e.g. $\emptyset \in \mathbf{P}$ and $\{1, 2\} \in \mathbf{P}$ and $\{1, 2, \{1, 2\}\} \in \mathbf{P}$

Question: Is $\mathbf{P} \in \mathbf{P}$?

But by def. of $\mathbf{P} \in \mathbf{P} \leftrightarrow \mathbf{P} \notin \mathbf{P}$

i.e. $\mathbf{P} \in \mathbf{P} \leftrightarrow \neg(\mathbf{P} \in \mathbf{P})$

By defining $\mathbf{P}$ we have created a contradition!

Conclusion: Naive set theory is inconsistent. We must eliminate such self-referential definitions to make set theory consistent.

Type Theory:

Russel created the theory of types, a new set theory that eliminated contradictions by construction.

How? Define a hierarchy of types (all possible sets). Any well defined set can only have elements from lower set levels.

Therefore $\mathbf{P} \in \mathbf{P}$ is always false! A set cannot contain itself since it can only contain elements from levels lower than itself.

Self-reference prohibited by preventing a type α from containing elements of type $\{\alpha\}$

Hierarchy of Types:

The universe $\mathbf{U}$ is composed of individuals (In-duh-viduals)

1. Lowest level - individuals: e.g. integer 2, "Bob"
   These are things that are not sets.
2. Next level - sets of individuals: which are of type $\mathbf{U} \rightarrow \{T, F\}$
   E.g. set of integers $\mathbb{Z}$, set of students, etc.
3. Higher levels - Let α and β be types from previous levels. Then $\alpha \rightarrow \beta$ is a type. Also $\alpha \rightarrow \{T, F\}$ is a type.

E.g. The set of class lists:

$$C : (\mathbf{U} \rightarrow \{T, F\}) \rightarrow \{T, F\}$$

A function $f : \alpha \rightarrow \beta$ has type or *signature* $\alpha \rightarrow \beta$

A function's *return type* is the range type (e.g. β for f above).

Sets, Sorts & Types

For our purposes, a *type* is just a set.

Type $\alpha \rightarrow \beta$ denotes the set of all (total) functions from α to β.

E.g. In PVS [[real, nzreal] -> real] is the set of all functions from real $\times$ non-zero reals to reals.

```
/:[[real, nzreal] -> real]
```

is an instance of type [[real, nzreal] -> real]

Some of the more algebraic treatments of logic refer to sorts instead of types. A *sort* is just a non-empty type.

Typechecking

Typed programming languages can check for easily decidable properties:

- use of undefined terms
- adding a boolean to an integer
- security violations (java)

These are properties that can be check mechanically. A language is *type safe* if programs exhibiting these properties will be rejected during typechecking (often during compilation).

PVS also automatically identifies these problems in specification files when they are *typechecked*.

Typechecking in PVS

More general typechecking is needed to make sure that formulas are *well typed* (i.e. never result in undefined terms).

Predicate subtypes with typechecking can be used to check for:

- division by zero
- out of bound array references
- more complicated properties (e.g. invariant properties of a database system)

Many properties are not effectively decidable (i.e. no general algorithm exists to check them). But we may still be able to *prove* them!

The use of *predicate subtypes* allows PVS to automatically generate the proof obligations (TCCs - Type Correctness Conditions) to guarantee formulas are well typed.

Predicate Subtypes

In our setting types can be thought of as sets. Thus a type α is a *subtype* of type β if the defining set of α is a subset of the defining set of β.

Predicate subtypes provide a tightly bound characterization by associating a predicate (property) with a subtype. In PVS, $\mathbb{N}$ is a predicate subtype of $\mathbb{Z}$.

```
nat: NONEMPTY_TYPE = {i:int | i >= 0} CONTAINING 0
```

The predicate is $i \geq 0$.

In the definition of type nzreal, real is the type that will be subtyped and $x \neq 0$ is the predicate defining the subtype.

For any $P : \alpha \rightarrow \{T, F\}$, a predicate defined on type α, P defines a subtype, denoted (P):

$$(P) = \{a \in \alpha | Pa\}$$

PVS Example

In PVS you can define a predicate:

$$even?: \mathbb{Z} \rightarrow \{T, F\}$$

Then use it to define predicate subtype of even integers:

```
even?(i:int):bool =
   EXISTS (j:int): i= 2 * j

even: TYPE = (even?)

f(i:int):even = 2 * i
```

Here $f: \mathbb{Z} \rightarrow even$

Interpreted and Uninterpreted Types

Interpreted types such as bool, real etc. provide standard mathematical interpretations.

Uninterpreted types:

- Abstract implementation details
- Allow parametrized types (e.g. sets) that are like C++ templates in LEDA

Example:

```
class:TYPE
mark:TYPE
transcript:TYPE = set[[class,mark]]
```

Prelude defines operators and properties of all types of sets using parametrized theory:

```
sets [T: TYPE]: THEORY
 BEGIN
  set: TYPE = [T -> bool]
  . . .
END sets
```

Empty Sets and Types

Extra care must be taken when dealing with possibly empty sets (types). Consider PVS declaration:

```
T:TYPE
const:T
```

declares a constant of type T. Results in following unprovable TCC:

```
% Existence TCC generated . . . for c: T
  % unfinished
c_TCC1: OBLIGATION (EXISTS (x: T): TRUE);
```

What's wrong? By definition $c \in T$ but if $T = \emptyset$ then we have a contradiction.

This can be fixed by making declaration:

```
T:NONEMPTY_TYPE
c:T
```

Proving quantified versions for empty and nonempty uninterpreted types.

Dependent types

What? parametrized families of types that can be used to

i) more accurately specify range of function

ii) restrict domain of (subsequent) arguments

Why use dependent types?

- the more specific you can be about a function's return value the easier it is to prove formulas utilizing it are "well typed" (contain no undefined terms for all possible variable values)
- restricting domain of function arguments w.r.t. current value of previous arguments is only way to make some "functions" total.

How? Make types depend on previous arguments

Dependent Types in Function Range

Ex. 1st version of abs(x)

```
abs(m:real): nonneg_real
  = IF m < 0 THEN -m ELSE m ENDIF
```

A better version

```
abs(m:real): {n: nonneg_real | n >= m}
  = IF m < 0 THEN -m ELSE m ENDIF
```

Note: For abs(x), the range type is dependent on the argument m, providing information in the type that is usually provided through separate lemmas.

```
h(x:real):nonneg_real=sqrt(abs(x)-x)
```

1st version generates more TCCs for h.

Dependent Types in Function Domain

Ex. Consider $\sqrt{x-y}$

```
  % Dependent Types Example
  sqrt: [nonneg_real -> nonneg_real]

f(x,y:real):nonneg_real=sqrt(x-y)
g(x:real,y:{y:real|x>=y}):nonneg_real=sqrt(x-y)
```

To see the Type Correctness Conditions generated use the PVS "show-tccs" command:

```
% Subtype TCC generated for x - y
  % unfinished
f_TCC1: OBLIGATION
  (FORALL (x: real, y: real): x - y >= 0);

% Subtype TCC generated for x - y
  % completed
g_TCC1: OBLIGATION
  (FORALL (x: real, y: {y: real | x >= y}):
x - y >= 0);
```

Type Information in PVS

```
g_TCC1 :

  |-------
{1} (FORALL (x: real, y: {y: real | x >= y}): x - y >= 0)

Rerunning step: (SKOLEM!)
Skolemizing,
this simplifies to:
g_TCC1 :

  |-------
{1}    x!1 - y!1 >= 0

Rerunning step: (TYPEPRED "y!1")
Adding type constraints for  y!1,
this simplifies to:
g_TCC1 :

{-1}    x!1 >= y!1
  |-------
[1]    x!1 - y!1 >= 0

Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures
Q.E.D.
```

(SKOLEM!) followed by (TYPEPRED "t") implemented by (SKOLEM-TYPEPRED).

Undefined Terms in PVS

Note: In PVS everything must be defined before its first use. E.g. If g were redefined as:

```
g(y:{y:real|x>=y},x:real):nonneg_real=sqrt(x-y)
```

PVS would produce the typecheck error:

```
    Expecting an expression
    No resolution for x
```

When defining a function

$$f(x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n) : t_r$$

t_j, the type of x_j, may only depend on the values of x_i's where $1 \le i < j$

The return type of the function, t_r, may depend upon any or all of the argument values.

PVS Command (REPLACE . . .)

Rule I part (b) Substitution of Equals is implemented by the PVS (REPLACE . . .) command.

$$
\begin{array}{r|c}
-1 & \phi_1 \\
-2 & \phi_2 \\
\vdots & \vdots \\
-n & t_L = t_R \\
\hline
1 & \psi_1 \\
2 & \psi_2 \\
\vdots & \vdots
\end{array}
\quad \overset{\text{(REPLACE -n * LR)}}{\Longrightarrow} \quad
\begin{array}{r|c}
-1 & \phi_1[t_R|t_L] \\
-2 & \phi_2[t_R|t_L] \\
\vdots & \\
-n & t_L = t_R \\
\hline
1 & \psi_1[t_R|t_L] \\
2 & \psi_2[t_R|t_L] \\
\vdots &
\end{array}
$$

$$
\begin{array}{r|c}
-1 & \phi_1 \\
-2 & \phi_2 \\
\vdots & \vdots \\
-n & t_L = t_R \\
\hline
1 & \psi_1 \\
2 & \psi_2 \\
\vdots & \vdots
\end{array}
\quad \overset{\text{(REPLACE -n * RL)}}{\Longrightarrow} \quad
\begin{array}{r|c}
-1 & \phi_1[t_L|t_R] \\
-2 & \phi_2[t_L|t_R] \\
\vdots & \\
-n & t_L = t_R \\
\hline
1 & \psi_1[t_L|t_R] \\
2 & \psi_2[t_L|t_R] \\
\vdots &
\end{array}
$$

Variations of (REPLACE . . .) command let you replace selected instances of equal terms.

PVS Commands (EXPAND "t")

Rule I(a): $(\forall x)x = x$ and all its variations are built into PVS

```
  x,y: VAR real
  f(x,y):real = x+y
  g(x,y):real = x+y
  Ia: THEOREM f(y,1)=g(y,1)

  |-------
{1}    (FORALL (y: real): f(y, 1) = g(y, 1))

Rule? (skolem! )

  |-------
{1}    f(y!1, 1) = g(y!1, 1)

Rule? (expand "f")
Expanding the definition of f,
```

```
  |-------
{1}    (1 + y!1 = g(y!1, 1))

Rule? (expand "g")
Expanding the definition of g,

  |-------
{1}    TRUE

which is trivially true.
Q.E.D.
```

Alternatively use (EXPAND* t_1 t_2 $\ldots t_n$) :

```
Ia :

  |-------
{1}    (FORALL (y: real): f(y, 1) = g(y, 1))

Rule? (expand* "f" "g")
Expanding the definition(s) of (f g),
Q.E.D.
```

PVS Commands (LIFT-IF)

```
P4 :

  |-------
{1}    FORALL (x: real):
IF x >= 0 THEN sqrt(x) ELSE sqrt(-x) ENDIF = sqrt(abs(x))

Rule? (skolem! )

  |-------
{1}    IF x!1 >= 0 THEN sqrt(x!1)
       ELSE sqrt(-x!1) ENDIF = sqrt(abs(x!1))

Rule? (lift-if )
Lifting IF-conditions to the top level,
this simplifies to:

  |-------
{1}    IF x!1 >= 0 THEN sqrt(x!1) = sqrt(abs(x!1))
      ELSE sqrt(-x!1) = sqrt(abs(x!1))
      ENDIF

Rule? (expand "abs")
P4 :

  |-------
{1}    TRUE

which is trivially true.
Q.E.D.
```

Tabular Specifications of Functions

A function $f : T_1 \times \ldots \times T_m \to T_r$ may have a tabular representation:

$$f(x_1, \ldots, x_m) = \begin{array}{|c|c|c|c|} \hline c_1 & c_2 & \ldots & c_n \\ \hline e_1 & e_2 & \ldots & e_n \\ \hline \end{array}$$

Here each c_i is a boolean expression (term) and e_i is a term of type T_r. When c_i is true f returns e_i.

The following are sufficient conditions for the table to properly define a (total) function:

Disjoint: $i \neq j \to (c_i \wedge c_j \leftrightarrow \bot)$

Complete: $(c_1 \vee c_2 \vee \ldots \vee c_n) \leftrightarrow \top$

Why? Why are they not necessary?

Example:

$$sign(x) = \begin{array}{|c|c|c|} \hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & 1 \\ \hline \end{array}$$

PVS COND Construct

COND
 c_1 -> e_1,
 c_2 -> e_2,
 ...
 c_n -> e_n
ENDCOND

PVS treats this the same as:

IF c_1 THEN e_1
ELSIF c_2 THEN e_2
 ...
ELSIF c_{n-1} THEN e_{n-1}
ELSE e_n

Therefore to prove properties involving COND statements can use (LIFT-IF) with (SPLIT) or (BDDSIMP). (GRIND) can also handle CONDs. (Why?)

Typechecking COND Statements

```
signs: TYPE = { x: int | x >= -1 & x <= 1}

  sign_cond(x): signs =
    COND
      x<0 -> -1,
      x=0 -> 0,
      x>0 -> 1
    ENDCOND
```

COND causes PVS to generate Disjointness and Completeness TCCs (proof obligations).

```
% Disjointness TCC generated for
 % COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  % unfinished
sign_cond_TCC3: OBLIGATION
      (FORALL (x: int):
 NOT (x < 0 AND x = 0)
 AND NOT (x < 0 AND x > 0)
 AND NOT (x = 0 AND x > 0));
```

```
% Coverage TCC generated for
% COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  % unfinished
sign_cond_TCC4: OBLIGATION
(FORALL (x: int): x < 0 OR x = 0 OR x > 0);
```

PVS Table Construct

Equivalent notation that is translated into PVS COND construct

```
  sign_htable(x): signs = TABLE
                 %-------------------%
                 |[ x<0 | x=0 | x>0 ]|
                 %-------------------%
                 |  -1  |  0  |  1  ||
                 %-------------------%
 ENDTABLE
```

2 dimensional version is nested CONDs

Example: 2A04 Lab 2

lab2 theory (intolerant version) OK

lab2b theory is lab2 with tolerance - 90+ cases of overlap

lab2d theory (somewhat improved) - gives unprovable sequent

```
func_TCC11.15.1 :

[-1]    (a!1 + b!1 < c!1)
[-2]    e(a!1, b!1)
[-3]    e(b!1, c!1)
[-4]    e(a!1, c!1)
  |-------
[1]    e(a!1, 0) & e(b!1, 0) & e(c!1, 0)

Rule?
```

Theorem CE in lab2d verifies existence of counter example.

lab2e final version w/tolerance - works!