Expressive Typing and Abstract Theories in Nuprl and PVS

Paul Jackson
U. of Edinburgh

TPHOLs
26th August '96

NOTES:

- Assume some familiarity with HOL-like system, but not necessarily PVS or Nuprl.
- Issues orthogonal to constructivity. No need to know about constructive type theory or propositions-as-types encoding of logic.
- will try to include references to other systems where appropriate (e.g. Coq, IMPS, Mizar).

I: Expressive Typing

- Examples of types in Nuprl and PVS, but not in e.g. HOL.

- Description and evaluation of type-checking procedures in

  - Nuprl

  - PVS

Subtypes and Parametric Types

- Examples:

$$\begin{aligned}
\mathbb{N} &= \{i\!:\!\mathbb{Z}\,|\,i \geq 0\} \\
\{j..k\} &= \{i\!:\!\mathbb{Z}\,|\,j \leq i \leq k\} \\
\mathsf{Inj}(A,B) &= \{f\!:\!(A \to B)| \\
&\qquad \forall x,y : A.\ fx = fy \Rightarrow x = y\}
\end{aligned}$$

- Use for function domain types:

$$\mathsf{Array}(T,n) \;=\; \{i\!:\!\mathbb{N}\,|\,i < n\} \to T$$

- Provide information on function ranges (examples to come)

NOTES:

- subtyping for quantifiers is a notational convenience. For function domains is significant advance in expressiveness.

Dependent-Product Types

$$x\colon A \times B_x \qquad\qquad (\Sigma x\colon A.\ B_x)$$

$\langle a,\ b\rangle \ \in \ x\colon A \times B_x$

if $a \in A$ and $b \in B_a$.

Type of subtraction function on $\mathbb{N}$:

$(i\colon\mathbb{N} \times \{j\colon\mathbb{N} \mid j \leq i\}) \rightarrow \mathbb{N}$

Dependent-Function Types

$$x : A \to B_x \qquad\qquad (\Pi x : A.\ B_x)$$

$f \;\in\; x : A \to B_x$

if for all $a \in A$ we have $(f\ a) \in B_a$.

Type of *mod* function:

$$\mathbb{N} \to m : \{i : \mathbb{N} \mid i \neq 0\} \to \{i : \mathbb{N} \mid i < m\}$$

Types for Full Specifications

Type of square root function:

$$x : \{z : \mathbb{R} \mid z \geq 0\} \to \{y : \mathbb{R} \mid y \geq 0 \land y^2 = x\}$$

Type Universes as Types

- Permit definition of functions that take types as arguments and return types as results.

- Consider function τ for programming language semantics that maps elements of:

$$\begin{array}{rcl} \texttt{Datatype Typ} & = & \texttt{Int} \mid \texttt{Bool} \\ & & \mid \texttt{Fun of Typ} \times \texttt{Typ} \\ & & \mid \texttt{Prod of Typ} \times \texttt{Typ} \end{array}$$

  to corresponding types in theorem prover. τ needs universe type as range.

- Consider typing the C `printf` function.

- Very useful for defining classes algebraic of algebraic structures ...

NOTES:

- Up till now all types feature in both PVS and Nuprl.

- Only Nuprl has universe types.

Conditional Well-formedness

- Total types for usually-partial datatype destructors:

$$\begin{aligned} \texttt{hd} &\in \{x\!:\!T \text{ List} \,|\, x \neq \texttt{nil}\} \to T \\ \texttt{tl} &\in \{x\!:\!T \text{ List} \,|\, x \neq \texttt{nil}\} \to T \text{ List} \end{aligned}$$

- Problem Expression:

$$x \neq \texttt{nil} \wedge \texttt{hd}\ x = k$$

  Similar issue with

  - $P \Rightarrow Q$,
  - $P \vee Q$
  - $\texttt{if}\ P\ \texttt{then}\ t\ \texttt{else}\ f$

Conditional Well-formedness

- If-then-else Example:

$$\begin{aligned}\texttt{fib}(n : \texttt{nat}) &= \texttt{if } n < 2 \texttt{ then } 1 \texttt{ else} \\ &\quad \texttt{fib}(n-1) + \texttt{fib}(n-2)\end{aligned}$$

- Redundant predicates?

$$\texttt{int?}(x) = \texttt{israt}(x) \wedge \texttt{isint}(x)$$

- Pathological Liberalness?

$$\texttt{False} \ \wedge \ (\lambda x . x)$$

Type checking with expressive types

- Non-parameterized Subtypes: $\mathbb{N}$, $\mathbb{Z} \subseteq \mathbb{R}$ (IMPS, Mizar, Isabelle)

- Integer parameters:
  Consider n-element array f of type

  $$\text{Array}(T, n) \;=\; \{i : \mathbb{N} \,|\, i < n\} \rightarrow T$$

  and lookup $f\ e$ with e linear? e non-linear?

- Non-uniqueness of Maximal Supertypes
  $\langle 5,\ \lambda i : \{0..5\}.i\rangle$ has maximal supertypes

  $$\mathbb{N} \times (\{0..5\} \rightarrow \{0..5\})$$

  and

  $$i : \mathbb{N} \times (\{0..i\} \rightarrow \{0..i\})$$

Type checking In Nuprl

- All by refinement-style proof.

- $$H_1, \ldots, H_n \vdash C$$

  means

  > "if hypotheses $H_1, \ldots, H_n$ are both well-formed and true, then conclusion C is also well-formed and true."

NOTES:

- Emphasize that *no* type-checking done outside of proof.

- Type-checking proofs are spread throughout the course of any proof; they aren't all done at start.

Nuprl rules generating type-checking subgoals

- Rules with a well-formedness premise:

$$\frac{\Gamma, A \vdash B \qquad \Gamma \vdash A \in \mathbb{P}}{\Gamma \vdash A \Rightarrow B}$$

$$\frac{\Gamma, x\!:\!T \vdash B \qquad \Gamma \vdash T \in \mathbb{U}}{\Gamma \vdash \forall x\!:\!T.\ B}$$

- Checking newly-introduced terms:

$$\frac{\Gamma,\ B_a \vdash C \qquad \Gamma \vdash a \in T}{\Gamma,\ \forall x\!:\!T.\ B_x \vdash C}$$

No checking necessary for cut:

$$\frac{\Gamma \vdash A \qquad \Gamma,\ A \vdash C}{\Gamma \vdash C}$$

Nuprl rules for doing type-checking

- Type Well-formedness:

$$\frac{\Gamma \vdash A \in \mathbb{U} \qquad\qquad \Gamma,\ x\!:\!A \vdash B \in \mathbb{U}}{\Gamma \vdash (x\!:\!A \rightarrow B) \in \mathbb{U}}$$

- Expression Well-formedness:

$$\frac{\Gamma \vdash a \in A \qquad\qquad \Gamma \vdash b \in B}{\Gamma \vdash \langle a,\ b\rangle \ \in\ A \times B}$$

$$\frac{\Gamma,\ x\!:\!A \vdash a \in B_x \qquad\qquad \Gamma,\ y\!:\!A \vdash B_y \in \mathbb{U}}{\Gamma \vdash (\lambda x.\ a)\ \in\ y\!:\!A \rightarrow B_y}$$

Checking function applications in Nuprl

Consider goal $\Gamma \vdash (f\ \ a) \in B$. Procedure is roughly:

1. Infer a type $x{:}A \rightarrow B'_x$ for f.

2. Now know that can probably prove

   $$(f\ \ a) \in B'_a$$

   Create subgoal

   $$\Gamma \vdash B'_a \subseteq B$$

3. Create subgoals

   $$\Gamma \vdash a \in A \qquad\qquad \Gamma \vdash f \in x{:}A \rightarrow B'_x$$

Notes on Nuprl procedure for proving applications

- Proof of $B'_a \subseteq B$ can involve reasoning about subtype predicates

- Alternate actions possible if $B'_a \subseteq B$ unprovable:
  - Alternative typings of f can be tried
  - B might be arithmetic subtype. If so, linear arithmetic decision procedure attempts proof of $(f\ a) \in B$.

Comments on automation of type-checking in Nuprl

- Linear arithmetic decision procedure essential when using arithmetic subtypes.

- Found it very useful to infer arithmetic properties of integer-valued functions. E.g. list length function.

- Performance often very poor. Caching and subsumption checking helpful.

Type checking in PVS
— based around type inference function τ

- On type, returns `TYPE` if type well-formed.
- On term, returns its type if term well-formed.
- τ also returns list of *Type Correctness Conditions* (TCCs) which need to be proven.
- TCCs appear as extra lemmas in PVS theories and as extra subgoals in proofs.
- Checking done whenever type, expressions and formulas are introduced, so all formulas in sequents are guaranteed well-formed.

Auxiliary functions on PVS types

- μ: finds maximal types
- π: finds predicate part of a type

For any type T:

$$T \;\equiv\; \{x : \mu(T) \mid \pi(T)(x)\}$$

An example:

$$T \;\doteq\; \mathbb{N} \to (i : \mathbb{N} \times \{j : \mathbb{Z} \mid j \leq i\})$$

then

$$\begin{array}{rcl} \mu(T) & = & \mathbb{N} \to (\mathbb{Z} \times \mathbb{Z}) \\ \pi(T) & = & \lambda f : (\mathbb{N} \to (\mathbb{Z} \times \mathbb{Z})).\ \forall x : \mathbb{N}. \\ & & \quad \pi_1(f\ x) \geq 0 \ \wedge \\ & & \quad \pi_2(f\ x) \leq \pi_1(f\ x) \end{array}$$

Definition of PVS type inference function τ

$$
\begin{array}{rcl}
\tau(\Gamma)(\langle a_1,\ a_2\rangle) & = & \tau(\Gamma)(a_1) \times \tau(\Gamma)(a_2) \\
\\
\tau(\Gamma)(\lambda x\,{:}\,A.\ a) & = & x\,{:}\,A \to B\ \textit{where} \\
 & & \quad \tau(\Gamma)(A) = \mathtt{TYPE} \wedge \\
 & & \quad B = \tau(\Gamma, x : \mathtt{VAR}\ A)(a) \\
\\
\tau(\Gamma)(f\ a) & = & B_a,\ \textit{where} \\
 & & \quad \tau(\Gamma)(f) = x\,{:}\,A \to B_x, \\
 & & \quad \tau(\Gamma)(a) = A', \\
 & & \quad \mu(A), \mu(A') \\
 & & \qquad \text{Compatible at } a \\
 & & \quad \Gamma \vdash \pi(A)(a)
\end{array}
$$

Compatibility testing also creates proof obligations.

Comments on type-checking in PVS

- Maintains seperation of type system and expression language.

- Higher performance than Nuprl, especially when not dealing with theories that generate many TCCs.

- Better, faster decision procedures to help out with solving TCCs. E.g. Shostak's integrated congruence-closure, linear arithmetic procedure. This also handles some basic non-linear arithmetic.

Property lemmas (judgements) in PVS

Given

```
0 : real

expt : [real,nat->real]

max : [m:real,n:real->
         {p: real | p >= m AND p >= n}]
```

the user supplies property lemmas such as:

```
0 HAS_TYPE nat

expt HAS_TYPE [rational, nat -> rational]
expt HAS_TYPE [posint, nat -> posint]

max HAS_TYPE [i:int,j:int ->
                {k: int | i<=k AND j<=k}]
max HAS_TYPE [i:nat,j:nat ->
                {k: nat | i<=k AND j<=k}]

posrat SUBTYPE_OF nzrat
```

Other typing-related issues in both PVS and Nuprl

- Argument synthesis

- Coercions

- Contravariant function subtyping

Argument synthesis

- In PVS can write

```
map f a
```

- PVS infers type parameters `S` and `T` from types of `f` and `a`

```
map[S,T] f a
```

- Something similar happens in Nuprl and many other systems

Coercions and function domain subtyping

- In

$$\sum_{i=a}^{b} f_i$$

ideally have $f \in \{a..b\} \to T$

- But then

$$\sum_{i=a}^{b} f_i = \sum_{i=a}^{c-1} f_i + \sum_{i=c}^{b} f_i$$

requires additional typings

$$f \in \{a..c-1\} \to T, \qquad f \in \{c..b\} \to T$$

Evaluation of expressive typing

- Specifications significantly more accurate and concise
- Higher level of reasoning
- Performance a concern
- Need fast powerful
  - linear (+ non-linear?) arithmetic
  - congruence reasoning
  - property inference
  - proof obligation subsumption
- If used with care, large developments very feasible

II: Abstract Theories

- Examples, Uses

- PVS

- Nuprl

- Issues

Introduction to abstract theories

Informally, an abstract theory consists of

- types T
- operators (possibly nullary) F over the types in T.
- predicates that the operators F can be assumed to satisfy

An abstract theory is instantiated when instances are provided for the types and operators that satisfy the predicates

Examples of abstract theories

A *monoid* is a tuple $\langle M, \circ, e\rangle$ where

- M is a type,
- $\circ$ is a binary operator of type $C^2 \to C$ and e is a distinguished element of M,
- $\circ$ is associative and e is a left and right identity for $\circ$.

Other examples are linear orders and stacks.

Example instances of abstract theories

$$
\begin{aligned}
\text{Semigroup}: &\quad \langle \mathbb{R},\ \mathtt{min} \rangle \\
\text{Monoid}: &\quad \langle T\ \mathtt{List},\ \mathtt{append},\ \mathtt{nil} \rangle \\
\text{AbelianMonoid}: &\quad \langle \mathbb{B},\ \wedge,\ \top \rangle \\
&\quad \langle \mathbb{N},\ \mathtt{max}\ , 0 \rangle \\
&\quad \langle T\ \mathtt{Set},\ \cup,\ \emptyset \rangle \\
\text{Group}: &\quad \langle T\ \text{Bij},\ \circ,\ \text{id},\ \text{inv} \rangle \\
\text{Field}: &\quad \langle \mathbb{R}, +, -, 0, \times, 1 \rangle
\end{aligned}
$$

Example theorems over abstract theories

Theorems about iteration:

- on semigroup / monoid

$$\vdash \sum_{i=j}^{k} x_i \ = \ x_j + \sum_{i=j+1}^{k} x_i$$

- on abelian monoid

$$\vdash \sum_{i \in A} x_i + \sum_{i \in B} x_i \ = \ \sum_{i \in A \uplus B} x_i$$

- on ring

$$a \times \sum_{i=j}^{k} x_i \ = \ \sum_{i=j}^{k} a \times x_i$$

Uses of abstract theories

- General theorem-proving support (view as enriched polymorphism)

- Program specification and refinement

- Mathematics (Algebra, Analysis, Topology, Category Theory)

An abstract theory as a PVS theory

```
monoids1[T : TYPE, o:[T,T->T], e:T] : THEORY
 BEGIN
  ASSUMING
   x,y,z : VAR T
   assoc  : ASSUMPTION  (x o y) o z =
                           x o (y o z)
   lident : ASSUMPTION  e o x = x
   rident : ASSUMPTION  x o e = x
  ENDASSUMING

  ...
 END monoids1
```

A development in PVS monoids theory

```
i,j : VAR int
f : VAR [int->T]

% f(i) o ... o f(j)

itop(i,j)(f): RECURSIVE T =
  IF i > j THEN e
  ELSE f(i) o itop(i+1,j)(f) ENDIF

   MEASURE  LAMBDA (i,j)(f) : max(1+j-i,0)


itop_unroll_hi : LEMMA
  i <= j IMPLIES
    itop(i,j)(f) = itop(i,j-1)(f) o f(j)
```

Importing and instantiating PVS theories

```
monoids2 : THEORY
 BEGIN

  intplusmon : THEORY = monoids1[int,+,0]

  i,j: VAR int
  f : VAR int->int
  sum(i,j)(f) = intplusmon.itop(i,j)(f)

  n: VAR nat
  sum_squares : LEMMA
     6 * sum(0,n)(LAMBDA (i): i * i) =
       n * (n+1) * (2 * n + 1)

 END monoids2
```

Abstract theories in Nuprl

All instances of a theory are collected into a type:

```
MonSig == T:𝕌 x op:(T→T→T) x T

|m| == m.1
*m == m.2.1
em == m.2.2

Assoc(T;op) ==
  ∀x,y,z:T. x op (y op z) = (x op y) op z

Ident(T;op;id) ==
  ∀x:T. x op id = x ∧ id op x = x

Mon == { m:MonSig | Assoc(|m|;*m)
                  ∧ Ident(|m|;*m;em) }
```

Note essential use of type universe $\mathbb{U}$.

Instances of monoids in Nuprl

```
<ℤ,+> ==  <ℤ, λx,y.x + y, 0>
⊢ <ℤ,+> ∈ Mon

r↓xmn ==  <|r|, *r, 1r>
⊢ ∀r:Rng. r↓xmn ∈ Mon
```

Example abstract theorem in Nuprl

$$
\begin{array}{l}
\vdash \forall \texttt{g:Mon.}\ \forall \texttt{a,b:}\mathbb{Z}. \\
\qquad \texttt{a} \leq \texttt{b} \\
\qquad \Rightarrow (\forall \texttt{E:}\{\texttt{a..b}^{-}\} \rightarrow \texttt{|g|.}\ \forall \texttt{k:}\mathbb{Z}. \\
\qquad\qquad\quad \mathbf{\Pi}\texttt{g a} \leq \texttt{j < b. E[j]} \\
\qquad\qquad\quad \texttt{=}\ \mathbf{\Pi}\texttt{g a + k} \leq \texttt{j < b + k. E[j - k])}
\end{array}
$$

$$\prod_{j=a}^{b-1} E_j = \prod_{j=a+k}^{b+k-1} E_{j-k}$$

When should algebraic classes be types?

If classes are not types

- Quantification over classes always outermost $\forall$
- Fixed finite number of class instances

If classes are types

- Arbitrary quantification and families of instances OK
- Can define reason about functions and operations on algebraic structures. E.g. free constructions, refinement mappings

NOTES:

- *Algebraic class* $\doteq$ collection of instances of an abstract theory
- *not types* approach OK for much theorem-proving support
- Type universes complicate type theory. Get non-canonical type expressions
- IMPS, EHDM, OBJ provide special support for refinement mappings without use of classes. However support not as flexible as when have classes
- classes essential for maths

Algebraic classes in PVS

```
monoids9[T : TYPE] : THEORY
 BEGIN
  MonTy : TYPE =
         [#
           c : set[T],
           op:[(c),(c)->(c)],
           id:(c)
         #]

  Mon?(m : MonTy) : bool =
    associative?(op(m))
    AND left_identity(op(m))(id(m))
    AND right_identity(op(m))(id(m))

  Mon : TYPE = (Mon?)

 ...
 END monoids9
```

NOTES:

- Similar to approach Elsa Gunter tried in HOL

- However, get function domains right in PVS

PVS development using monoid class type

```
m,n,p : VAR Mon
x,y : VAR T

HomTy(m,n) : TYPE = [(c(m)) -> (c(n))]

hom?(m,n)(f : HomTy(m,n)) : bool =
  (FORALL (x,y:(c(m))) : f(op(m)(x,y))
                          = op(n)(f(x),f(y)))
  AND f(id(m)) = id(n)

Hom(m,n) : TYPE = (hom?(m,n))

hom_comp : LEMMA
   FORALL (f:Hom(m,n)),(g:Hom(n,p)) :
     hom?(m,p)(g o f)
```

Automatically instantiating abstract theories

Consider using the abstract theorem:

$$\forall m\,{:}\,\texttt{Mon}.\ \forall x, y, z, w\,{:}\,|m|.\\ \quad (x \circ_m y) \circ_m (z \circ_m w) = x \circ_m (y \circ_m z) \circ_m w$$

to rewrite

$$(1 + 2) + (3 + 4)$$

where $+ \in \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$.

A simple matching function could yield bindings

$$x \mapsto 1,\ y \mapsto 2,\ z \mapsto 3,\ w \mapsto 4$$

$$\circ_m \mapsto +$$

Type matching could give $|m| \mapsto \mathbb{Z}$, yielding the binding

$$m \mapsto \langle \mathbb{Z},\ +,\ u \rangle$$

for unknown u.
Knowing m must have type `Mon`, consultation of a maths database could give the full binding

$$m \mapsto \langle \mathbb{Z},\ +,\ 0 \rangle$$

Issues in automatic instantiation

- database still needed to justify typing for m, even if no unknowns.

- database might only have entry

  $\langle \mathbb{Z},\ +,\ 0,\ - \rangle \in \texttt{AbGroup}$

  Need to know that

  $\texttt{AbGroup} \subseteq^* \texttt{Mon}$

- Automation of inference with $\subseteq^*$ important

- Defining $S \subseteq^* T$ easiest when S, T have named fields (Mizar, IMPS, Axiom).

$\subseteq^*$ with named fields (structural subtyping)

	AbGroup	**Mon**
fields	$C : \mathbb{U}$ $op : C^2 \to C$ $inv : C \to C$ $id : C$	$C : \mathbb{U}$ $op : C^2 \to C$ $id : C$
properties	$Assoc(C, op)$ $Ident(C, op, id)$ $Inv(C, op, id, inv)$ $Comm(C, op)$	$Assoc(C, op)$ $Ident(C, op, id)$

Key issues in abstract theories

- theory interpretations
  - special support / automation needed
  - structure subtyping a first step
- Algebraic classes as types or theories?
  - For mathematics
  - For hardware/software verification
  - For program specification refinement