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:**

\[
\begin{align*}
\mathbb{N} &= \{i : \mathbb{Z} \mid i \geq 0\} \\
\{j..k\} &= \{i \in \mathbb{Z} \mid j \leq i \leq k\} \\
\text{Inj}(A, B) &= \{f : (A \to B) \mid \forall x, y : A. \, fx = fy \Rightarrow x = y\}
\end{align*}
\]

- **Use for function domain types:**

\[
\text{Array}(T, n) = \{i : \mathbb{N} \mid i < n\} \to T
\]

- **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

\[ x : A \times B_x \quad (\Sigma x : A. \ B_x) \]

\[ \langle a, b \rangle \in x : A \times B_x \]

if \( a \in A \) and \( b \in B_a \).

Type of subtraction function on \( \mathbb{N} \):

\[ (i : \mathbb{N} \times \{j : \mathbb{N} \mid j \leq i\}) \rightarrow \mathbb{N} \]
Dependent-Function Types

\[ x : A \rightarrow B_x \]  \quad (\Pi x : A. \ B_x)

\[ f \in x : A \rightarrow B_x \]

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

Type of \textit{mod} function:

\[ \mathbb{N} \rightarrow m : \{i : \mathbb{N} \mid i \neq 0\} \rightarrow \{i : \mathbb{N} \mid i < m\} \]
Types for Full Specifications

Type of square root function:

\[ x : \{ z : \mathbb{R} \mid z \geq 0 \} \rightarrow \{ y : \mathbb{R} \mid y \geq 0 \land y^2 = x \} \]
Type Universes as Types

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

- Consider function $\tau$ for programming language semantics that maps elements of:

  $$
  \text{Datatype Typ} = \text{Int} | \text{Bool} | \text{Fun of Typ} \times \text{Typ} | \text{Prod of Typ} \times \text{Typ}
  $$

  to corresponding types in theorem prover. $\tau$ 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:

\[
\begin{align*}
\text{hd} & \in \{ x : T \text{ List} \mid x \neq \text{nil} \} \rightarrow T \\
\text{tl} & \in \{ x : T \text{ List} \mid x \neq \text{nil} \} \rightarrow T \text{ List}
\end{align*}
\]

- Problem Expression:

\[
x \neq \text{nil} \land \text{hd} \ x = k
\]

Similar issue with

- \( P \Rightarrow Q \),

- \( P \lor Q \)

- if \( P \) then \( t \) else \( f \)
Conditional Well-formedness

- If-then-else Example:

\[
\text{fib}(n : \text{nat}) = \begin{cases} 
1 & \text{if } n < 2 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise}
\end{cases}
\]

- Redundant predicates?

\[
\text{int?}(x) = \text{israt}(x) \land \text{isint}(x)
\]

- Pathological Liberalness?

\[
\text{False} \land (\lambda x. x)
\]
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

\[
\text{Array}(T, n) = \{ i : \mathbb{N} \mid i < n \} \rightarrow T
\]

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 sub-goals

• Rules with a well-formedness premise:

\[
\frac{\Gamma, A \vdash B \quad \Gamma \vdash A \in \mathbb{P}}{\Gamma \vdash A \Rightarrow B}
\]

\[
\frac{\Gamma, x : T \vdash B \quad \Gamma \vdash T \in \mathbb{U}}{\Gamma \vdash \forall x : T. B}
\]

• Checking newly-introduced terms:

\[
\frac{\Gamma, B_a \vdash C \quad \Gamma \vdash a \in T}{\Gamma, \forall x : T. B_x \vdash C}
\]

No checking necessary for cut:

\[
\frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}
\]
Nuprl rules for doing type-checking

- Type Well-formedness:

\[
\Gamma \vdash A \in \mathbb{U} \quad \Gamma, \ x : A \vdash B \in \mathbb{U} \\
\Gamma \vdash (x : A \rightarrow B) \in \mathbb{U}
\]

- Expression Well-formedness:

\[
\Gamma \vdash a \in A \quad \Gamma \vdash b \in B \\
\Gamma \vdash \langle a, b \rangle \in A \times B
\]

\[
\Gamma, \ x : A \vdash a \in B_x \quad \Gamma, \ y : A \vdash B_y \in \mathbb{U} \\
\Gamma \vdash (\lambda x. \ a) \in y : A \rightarrow B_y
\]
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

   $$\Gamma \vdash a \in A \quad \Gamma \vdash f \in x:A \rightarrow B'_x$$
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 $(f \ a) \in B$. 

14
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 \( \tau \)

- On type, returns TYPE if type well-formed.
- On term, returns its type if term well-formed.
- \( \tau \) 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

- $\mu$: finds maximal types

- $\pi$: finds predicate part of a type

For any type $T$:

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

An example:

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

then

$$\mu(T) = \mathbb{N} \to (\mathbb{Z} \times \mathbb{Z})$$
$$\pi(T) = \lambda f : (\mathbb{N} \to (\mathbb{Z} \times \mathbb{Z})). \forall x : \mathbb{N}.
\begin{align*}
\pi_1(f \ x) & \geq 0 \ \land \\
\pi_2(f \ x) & \leq \pi_1(f \ x)
\end{align*}$$
Definition of PVS type inference function $\tau$

$$\tau(\Gamma)(\langle a_1, a_2 \rangle) = \tau(\Gamma)(a_1) \times \tau(\Gamma)(a_2)$$

$$\tau(\Gamma)(\lambda x : A. \ a) = x : A \to B \ where$$

$$\tau(\Gamma)(A) = \text{TYPE} \land$$

$$B = \tau(\Gamma, x : \text{VAR } A)(a)$$

$$\tau(\Gamma)(f \ a) = B_a, \ where$$

$$\tau(\Gamma)(f) = x : A \to B_x,$$

$$\tau(\Gamma)(a) = A',$$

$$\mu(A), \mu(A')$$

Compatible at $a$

$$\Gamma \vdash \pi(A)(a)$$

Compatibility testing also creates proof obligations.
Comments on type-checking in PVS

- Maintains separation 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

  \[
  \text{map } f \ a
  \]

- PVS infers type parameters \( S \) and \( T \) from types of \( f \) and \( a \)

  \[
  \text{map} \left[ S, T \right] \ f \ a
  \]

- Something similar happens in Nuprl and many other systems
Coercions and function domain subtyping

- In

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

ideally have \( f \in \{a..b\} \rightarrow T \)

- But then

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

requires additional typings

\( f \in \{a..c-1\} \rightarrow T, \quad f \in \{c..b\} \rightarrow T \)
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 $C^2 \rightarrow C$ 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

Semigroup : \( \langle \mathbb{R}, \text{min} \rangle \)
Monoid : \( \langle T \text{ List}, \text{append, nil} \rangle \)
AbelianMonoid : \( \langle \mathbb{B}, \land, \top \rangle \)
\( \langle \mathbb{N}, \text{max}, 0 \rangle \)
\( \langle T \text{ Set}, \cup, \emptyset \rangle \)
Group : \( \langle T \text{ Bij}, \circ, \text{id, inv} \rangle \)
Field : \( \langle \mathbb{R}, +, -, 0, \times, 1 \rangle \)
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

\[ \vdash \sum_{i \in A} x_i + \sum_{i \in B} x_i = \sum_{i \in A \cup B} x_i \]

- 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

\[
i, j : \text{VAR int}
\]
\[
f : \text{VAR [int->T]}
\]
\[
% f(i) \circ \ldots \circ f(j)
\]
\[
\text{itop}(i, j)(f) : \text{RECURSIVE T =}
\]
\[
\text{IF } i > j \text{ THEN } e
\]
\[
\text{ELSE } f(i) \circ \text{itop}(i+1, j)(f) \text{ ENDIF}
\]
\[
\text{MEASURE LAMBDA (i, j)(f) : max(1+j-i, 0)}
\]
\[
\text{itop_unroll_hi : LEMMA}
\]
\[
i \leq j \text{ IMPLIES}
\]
\[
\text{itop}(i, j)(f) = \text{itop}(i, j-1)(f) \circ f(j)
\]
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:

\[ \text{MonSig} \equiv T : \mathbb{U} \times \text{op} : (T \to T \to T) \times T \]

\[ |m| \equiv m.1 \]
\[ *m \equiv m.2.1 \]
\[ \text{em} \equiv m.2.2 \]

\[ \text{Assoc}(T; \text{op}) \equiv \]
\[ \forall x, y, z : T. \ x \ \text{op} \ (y \ \text{op} \ z) = (x \ \text{op} \ y) \ \text{op} \ z \]

\[ \text{Ident}(T; \text{op}; \text{id}) \equiv \]
\[ \forall x : T. \ x \ \text{op} \ \text{id} = x \ \land \ \text{id} \ \text{op} \ x = x \]

\[ \text{Mon} \equiv \{ \ m : \text{MonSig} \mid \text{Assoc}(|m|; *m) \land \text{Ident}(|m|; *m; \text{em}) \} \]

Note essential use of type universe \( \mathbb{U} \).
Instances of monoids in Nuprl

\(<\mathbb{Z}, +> = \langle\mathbb{Z}, \lambda x, y. x + y, 0\rangle\)
\(\vdash \langle\mathbb{Z}, +\rangle \in \text{Mon}\)

\(r\downarrow\text{rmn} = \langle|r|, \ast r, 1r\rangle\)
\(\vdash \forall r: \text{Rng. } r\downarrow\text{rmn} \in \text{Mon}\)
Example abstract theorem in Nuprl

\[ \vdash \forall g:\text{Mon}. \ \forall a, b : \mathbb{Z}. \]
\[ a \leq b \]
\[ \Rightarrow (\forall E:\{a..b\} \rightarrow |g|. \ \forall k : \mathbb{Z}. \]
\[ \Pi g \ a \leq j < b. \ E[j] \]
\[ = \Pi g \ a + k \leq j < b + k. \ E[j - k]) \]

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

\[ m, n, p : \text{VAR Mon} \]
\[ x, y : \text{VAR T} \]

\[ \text{HomTy}(m, n) : \text{TYPE} = [(c(m)) \rightarrow (c(n))] \]

\[ \text{hom?}(m, n)(f : \text{HomTy}(m, n)) : \text{bool} = \]
\[ \quad (\text{FORALL } (x, y : (c(m))) : f(\text{op}(m)(x, y)) \]
\[ 	\quad = \text{op}(n)(f(x), f(y))) \]
\[ \quad \text{AND } f(\text{id}(m)) = \text{id}(n) \]

\[ \text{Hom}(m, n) : \text{TYPE} = (\text{hom?}(m, n)) \]

\[ \text{hom_comp} : \text{LEMMA} \]
\[ \quad \text{FORALL } (f : \text{Hom}(m, n)), (g : \text{Hom}(n, p)) : \]
\[ \quad \text{hom?}(m, p)(g \circ f) \]
Automatically instantiating abstract theories

Consider using the abstract theorem:

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

to rewrite

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

where $+$ $\in \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$. 
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 \( \text{Mon} \), 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

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

  Need to know that

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

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

- Defining $S \subseteq^* T$ easiest when $S, T$ have named fields (Mizar, IMPS, Axiom).
\[ \mathbb{C}^* \text{ with named fields (structural subtyping)} \]

<table>
<thead>
<tr>
<th></th>
<th>AbGroup</th>
<th>Mon</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fields</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( C : \mathbb{U} )</td>
<td>( C : \mathbb{U} )</td>
</tr>
<tr>
<td></td>
<td>( op : C^2 \rightarrow C )</td>
<td>( op : C^2 \rightarrow C )</td>
</tr>
<tr>
<td></td>
<td>( inv : C \rightarrow C )</td>
<td>( id : C )</td>
</tr>
<tr>
<td></td>
<td>( id : C )</td>
<td>( id : C )</td>
</tr>
<tr>
<td><strong>properties</strong></td>
<td>( \text{Assoc}(C, op) )</td>
<td>( \text{Assoc}(C, op) )</td>
</tr>
<tr>
<td></td>
<td>( \text{Ident}(C, op, id) )</td>
<td>( \text{Ident}(C, op, id) )</td>
</tr>
<tr>
<td></td>
<td>( \text{Inv}(C, op, id, inv) )</td>
<td>( \text{Inv}(C, op, id, inv) )</td>
</tr>
<tr>
<td></td>
<td>( \text{Comm}(C, op) )</td>
<td>( \text{Comm}(C, op) )</td>
</tr>
</tbody>
</table>
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