Mathematical Proofs on the computer

Yves Bertot
12 March

MAP INTERNATIONAL SPRING SCHOOL ON FORMALIZATION OF MATHEMATICS 2012

SOPHIA ANTIPOLIS, FRANCE / 12-16 MARCH

Inria informatics mathematics · ihp Institut Henri Poincaré · MICROSOFT RESEARCH INRIA JOINT CENTRE

Thanks!

- Inria, Microsoft Research-Inria joint centre, Institut Henri Poincaré
- Agnès Cortell, Nathalie Bellesso
- Guillaume Cano, Cyril Cohen, Maxime Dénés, Anders Mörtberg, Ioana Pașca
- Georges Gonthier, Thomas Hales, Julio Rubio, Bas Spitters, Vladimir Voevodsky
- Assia Mahboubi, Laurence Rideau, Pierre-Yves Strub, Enrico Tassi, Laurent Théry
- And all the participants: please do share your impressions!

A computer language for mathematics

- The calculus of constructions: a simple kernel based on dependent types
  - Algorithms
  - Proofs
- Extra layers to bridge the gap with mathematical practice
  - Notations
  - implicit arguments
  - coercions
  - canonical structures
  - Changing point of view
- Libraries of results
  - structuring principles
  - Searching approaches

Outline

1. Kernel
2. Layers
3. Library

The kernel

- Propositions as types, Programs as proofs
- A proof of "A implies B" is a tool to produce proofs of B
  - But it requires a proof of A as input
- Notation `f (g a)` instead of $f(g(a))$
- Notation `fun x : A => e`
  for the function that maps x of type A to e
- Example `fun x : A => x` is a proof of A -> A
  read this A *implies* B
- A simple notion of truth, a simple verification problem
- Beware that some concepts on computer are not totally faithful to reality, example of subtraction

Dependent types

- Families of types:
  - `list A`, lists of elements of type `A`
  - `prime n`, proofs that `n` is prime (if any)
  - `ordinal n`, numbers smaller than `n`
- More than one type for the possible outputs of one function
- Notation `forall x : T, B x`
- Useful for polymorphism: `nil : forall A, list A`
- Useful for logic: `forall x : nat, ~prime(4 * n)`

Dependent pairs

- Data with extra information
- Example `ordinal n` (will be noted `'I_n`)
  - Each element combines a number `p` and a proof of `p < n`
- Not exactly a subset of the type of natural numbers
- Used pervasively this week: qualified types
- Especially useful if the qualification is given by a boolean predicate
  - `eqtype`, `choicetype`, `monoid`, etc

Practical approaches

- The mathematical language uses a lot of ambiguity
- Notational conventions abound
- Polymorphism does not require explicit types
- Qualified data should be usable as unqualified data
- Information should be added to existing types as proof progresses

Notations

- Numbers are just a notation on top of a data-structure:
  3 = S (S (S O))
- S x is actually written x.+1
- a && b is a notation for andb a b
- operations are "dissymetric" :
  S (S (S x) and 3 + x are convertible, but not x + 3
- comparisons are computations:
  2 < 5 + x is convertible to true, but not 2 < x + 5

Implicit Arguments

- Coq can be configured so that arguments of functions are guessed when possible
- Functions with 5 arguments behave as if they had 2
- Convention used extensively in ssreflect
  Set Implicit Arguments. Unset Strict Implicit.
- About cat.
  cat : forall T : Type, seq T -> seq T -> seq T
  Argument T is implicit and maximally inserted
- Check cat [:: 1; 2].
  cat [:: 1; 2] : seq nat -> seq nat
- Also used for many theorems
  apply them directly to proofs of their first hypothesis

Coercions

- Data with added information does not belong in the same type
- For instance i : 'I_n contains both a natural number p and a proof that p < n
  Technically, it cannot be used as a natural number
- Coercions bridge the gaps
- Print Coercions. shows all coercions
- For instance nat_of_ord : forall n, 'I_n -> nat
- i + 5 is actually (nat_of_ord n i) + 5

Adding information to existing objects

- In human memory, everything has "connotations"
- Numbers: addition is commutative, has a neutral element...
- Number operations are created naked, structure is added later
- The mechanism is called a *canonical structure*
- For instance, for every associative op we have

```
t1 : forall x y z t, op (op x y) (op z t)=
   op x (op y (op z t))
```

- addition of numbers is associative, and so is concatenation of sequences
- Canonical structures provide a direct way to remember associativity and to apply `t1` to both operators

Changing points of view

- Changing points of view about objects is a natural process in mathematics
- In computer things are more rigid
- A systematic way to use equivalences or isomorphisms
- For instance coprimeP
  $(\exists u : nat * nat, u_1 n - u_2 m = 1) \Leftrightarrow (\texttt{coprime}\ n\ m)$
- apply/coprimeP will use the equivalence in the appropriate direction

The ssreflect library

- A large library (distributed version : 70kloc in 54 files)
- Mainly organized to support the odd order theorem
  - forays in algebra and linear algebra
  - advanced treatement of matrices and polynomials
- Loading files as needed
- `Require Import ssreflect ssrfun ssrbool ...`
- Theorem naming is systematic
  - Properties of associativity are denoted by an `A`
  - Properties of commutativity are denoted by a `C`
  - Properties of inversion are called `cancel` and denoted by a `K`

Maintenance discipline

- Big documents : big maintenance problem
- Proofs are linear script with an underlying tree structure
  - Making the tree-structure apparent: terminate branches with by
  - Indentation : 2*(n-1) when n subgoals are open
  - Always choose names for your hypotheses during proofs

Searching information in the library

- Use the graph to navigate files
  - Each node gives access to file outlines
  - File outlines contain documentation in the preamble
  - Defined symbols are clickable
- Within Coq, use the `Search` command

Searching

- Search *pattern*.
  Look for theorems whose conclusion matches the pattern
- Search $pattern_1$ $pattern_2$. Look for theorems whose conclusion matches $pattern_1$ and which contain $pattern_2$

```
Search (_ <= _ * _).
ltn_ceil  forall m d : nat, 0 < d -> m < (m %/ d).+1 * d
leq_pmull  forall m n : nat, 0 < n -> m <= n * m
leq_pmulr  forall m n : nat, 0 < n -> m <= m * n
...
Search (_ = _) (_ * _) (_ <= _).
eqn_pmul2l  forall .., 0 < m -> (m * n1 == m * n2) = (n1 == n2)
eqn_pmul2r  forall .., 0 < m -> (n1 * m == n2 * m) = (n1 == n2)
```