Computation and reflection in Coq and HOL

John Harrison

Intel Corporation, visiting Katholieke Universiteit Nijmegen

20th August 2003

What is reflection?

Stepping back from the straightforward use of a formal system and reasoning *about* it (‘reflecting upon it’).

- Exploiting the syntax to prove general/meta theorems
- Asserting consistency or soundness

Similar concept in programming where programs can examine and modify their own syntax [Brian Cantwell Smith 1984].

The ‘reflection principle’ in ZF set theory is rather different.

Logical power of reflection

The process of reflection may involve:

- Adding new theorems that were not provable before
- Adding a new, but conservative, rule to the logic
- Making no extension of the logic

[Giunchiglia and Smail 1989] use ‘reflection principles’ to refer to logic-strengthening rules only, but use ‘reflection’ to describe the process in all cases.

Reflection principles

The classic reflection principle is an assertion of consistency.

More generally, we can assert the ‘local reflection schema’:

$$\vdash Pr(\ulcorner \phi \urcorner) \Rightarrow \phi$$

By Gödel’s Second Incompleteness Theorem, neither is provable in the original system, so this makes the system stronger.

The addition of reflection principles can then be iterated transfinitely [Turing 1939], [Feferman 1962], [Franzén 2002].

A conservative reflection rule

Consider instead the following reflection rule:

$$\frac{\vdash Pr(\ulcorner \phi \urcorner)}{\vdash \phi}$$

Assuming that the original logic is Σ-sound, this is a conservative extension.

On the other hand, it does make some proofs much shorter.

Whether it makes any *interesting* ones shorter is another matter.

Total internal reflection

We can exploit the syntax-semantics connection without making any extensions to the logical system.

This has been done quite often in HOL, but not considered as ‘reflection’.

It’s usually called ’using proforma theorems’.

However, as I understand it, this is essentially what is called ‘reflection’ in Coq.

Calculation and proof in Coq

In Coq and other constructive type theories:

- There is a distinct notion of ‘definitional’ equality.
- One can consider this as formalizing the notion of a ‘calculation’, as distinct from a proof
- Coq’s primitive core therefore includes a special reduction engine
- This may make reduction much more efficient than conventional equality reasoning.
- On the other hand, it makes the logical core more complicated.

Calculation and proof in HOL

In HOL and similar classical provers, however:

- There is no such formal distinction, and only one kind of equality.
- Calculations are implemented (via programmability) in terms of inferences.
- Thus, calculations are just a particular case of proofs.
- The logical core is simpler both conceptually and practically
- The specific reduction strategy is precisely controllable.
- However, reduction is somewhat less efficient.

Since HOL does not save proofs, the issue of proof construction during the calculation is not as significant as it would be in Coq.

Reflection in Coq and HOL

I suspect that proforma theorems in HOL and reflection in Coq are essentially the same:

$$\begin{tikzcd}
f(x) & \ulcorner f(x) \urcorner \arrow[l, "\text{Syntax to semantics}"'] \\
x \arrow[u, "f"] \arrow[r, "\text{Semantics to syntax}"'] & \ulcorner x \urcorner \arrow[u, "\text{Syntactic transform}"']
\end{tikzcd}$$

Both achieve a transformation $x \mapsto f(x)$ using a syntactic transform.

The difference

The main difference is that:

- In HOL, the syntactic transformations are done by ordinary inference
- In Coq, they are done by the special reduction engine

Some things like ring normalization are done directly in HOL, but using reflection in Coq.

The main motivation in Coq must be to avoid creating proofs (?)

Nevertheless, even in HOL the technique is often very useful.

Ring normalization results

HOL's normalizer and the Coq "`Rational`" tactic seem to have comparable performance on tests from Hilbert-Waring problem [Nathanson 1996], though Coq's "`Ring`" is much worse.

Problem	HOL normalizer	Coq "`Rational`"
`LAGRANGE_4`	0.3	2.7
`LAGRANGE_8`	2.0	47.0
`LIOUVILLE`	0.6	1.4
`FLECK`	9.9	17.0
`HURWITZ`	124.6	389.0
`SCHUR`	268.5	1428.0

For comparison, SCHUR takes 0.06 seconds in PARI/GP.

Presburger arithmetic

Presburger arithmetic is the decidable theory of linear arithmetic over $\mathbb{Z}$ (or equally well $\mathbb{N}$).

A nice example of reflection in HOL is the quantifier elimination step in Cooper's algorithm for this theory.

Using a proforma theorem, we can prove the main transformation once and for all, then apply it quite efficiently in particular cases.

The rest of my talk will show how this works.

The syntax of formulas

We first define a HOL type for the relevant class of formulas:

```
let cform_INDUCT,cform_RECURSION = define_type
  "cform = Lt int
         | Gt int
         | Eq int
         | Ne int
         | Divides int int
         | Ndivides int int
         | And cform cform
         | Or cform cform
         | Nox bool";;
```

That is, quantifier-free first order formulas in the language of arithmetic, assumed in NNF.

The semantics of formulas

The meaning of these formulas is now defined recursively:

```
let interp = new_recursive_definition cform_RECURSION
  ‘(interp x (Lt e) = x + e < &0) /\
   (interp x (Gt e) = x + e > &0) /\
   (interp x (Eq e) = (x + e = &0)) /\
   (interp x (Ne e) = ~(x + e = &0)) /\
   (interp x (Divides c e) = c divides (x + e)) /\
   (interp x (Ndivides c e) = ~(c divides (x + e))) /\
   (interp x (And p q) = interp x p /\ interp x q) /\
   (interp x (Or p q) = interp x p \/ interp x q) /\
   (interp x (Nox P) = P)‘;;
```

Syntactic transformations

We can now define the syntactic transformation defined by Cooper, e.g. the ‘minus infinity’ version $\psi_{-\infty}$ of a formula ψ:

```
let minusinf = new_recursive_definition cform_RECURSION
 ‘(minusinf (Lt e) = Nox T) /\
  (minusinf (Gt e) = Nox F) /\
  (minusinf (Eq e) = Nox F) /\
  (minusinf (Ne e) = Nox T) /\
  (minusinf (Divides c e) = Divides c e) /\
  (minusinf (Ndivides c e) = Ndivides c e) /\
  (minusinf (And p q) = And (minusinf p) (minusinf q)) /\
  (minusinf (Or p q) = Or (minusinf p) (minusinf q)) /\
  (minusinf (Nox P) = Nox P)‘;;
```

Other syntactic notions

And similarly the ‘B-set’ of a formula:

```
let Bset = new_recursive_definition cform_RECURSION
 ‘(Bset (Lt e) = {}) /\
  (Bset (Gt e) = {(--e)}) /\
  (Bset (Eq e) = {(--(e + &1))}) /\
  (Bset (Ne e) = {(--e)}) /\
  (Bset (Divides c e) = {}) /\
  (Bset (Ndivides c e) = {}) /\
  (Bset (And p q) = (Bset p) UNION (Bset q)) /\
  (Bset (Or p q) = (Bset p) UNION (Bset q)) /\
  (Bset (Nox P) = {})‘;;
```

Auxiliary concept

We also define a predicate asserting that the moduli of any congruence appearing in a formula divide some integer d:

```
let alldivide = new_recursive_definition cform_RECURSION
 `(alldivide d (Lt e) = T) /\
  (alldivide d (Gt e) = T) /\
  (alldivide d (Eq e) = T) /\
  (alldivide d (Ne e) = T) /\
  (alldivide d (Divides c e) = c divides d) /\
  (alldivide d (Ndivides c e) = c divides d) /\
  (alldivide d (And p q) = alldivide d p /\ alldivide d q
  (alldivide d (Or p q) = alldivide d p /\ alldivide d q)
  (alldivide d (Nox P) = T)`;;
```

Cooper’s main result

We can now state, and prove by induction over formulas, the main transformation in Cooper’s algorithm:

```
|- !p d. alldivide d p /\ &0 < d
         ==> ((?x. interp x p) =
                 ?j. &1 <= j /\ j <= d /\
                     (interp j (minusinf p) \/
                      ?b. b IN Bset p /\ interp (b + j) p))
```

This is a direct formulation of the main result in Cooper’s paper.

Applying the theorem

To apply the theorem in a particular case, we start with the innermost quantifier and:

- Put the body into a canonical form
- Map it into the interpretation of a formula by rewriting ‘backwards’ with the definition of `interp`
- Apply Cooper’s main transformation to get a new formula
- Map back to the semantics, by rewriting ‘forwards’ with the definition of `interp`

Using the usual programmability, all this is automated, and reasonably efficient.

The overall quantifier elimination procedure just iterates this process inside-out.

Examples

```
#COOPER_CONV `!x. a < &3 * x ==> b < &3 * x`;;
it : thm =
 |- (!x. a < &3 * x ==> b < &3 * x) =
    ((~(&3 divides a + &1) \/ &0 < a + -- &1 * b + &1) /\
     (~(&3 divides a + &2) \/ &0 < a + -- &1 * b + &2)) /\
    (~(&3 divides a + &3) \/ &0 < a + -- &1 * b + &3)
#COOPER_CONV `!x. ~(&2 divides x) /\ &3 divides (x - &1) =
                  &12 divides (x - &1) \/ &12 divides (x - &7)`;;
it : thm =
 |- (!x. ~(&2 divides x) /\ &3 divides x - &1 =
         &12 divides x - &1 \/ &12 divides x - &7) = T
#COOPER_CONV `!x. x >= &8 ==> ?u v. u >= &0 /\ v >= &0 /\
                                    (x = &3 * u + &5 * v)`;;
it : thm =
 |- (!x. x >= &8
         ==> (?u v. u >= &0 /\ v >= &0 /\ (x = &3 * u + &5 * v))) = T
```

Conclusions

There is an established method in the HOL community for using ‘proforma theorems’ to implement inference patterns once and for all.

- It seems very similar to the use of reflection in Coq
- It is not as critical (because we don’t worry about proof terms) but can still be useful

Another further step is to extract the syntactic object as a program and run it externally.

This could really improve efficiency more dramatically, but has its own drawbacks.