Le langage ssreflect

Georges Gonthier

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

Un peu d'histoire

- Ssreflect est issu du jeu de macros camlp4 de ma preuve du T4C
- Motivations: faciliter le développement exploratoire, la maintenance, le travail à deux fenêtres…
- … et l'utilisation de l'ordre supérieur
- Plusieurs versions avant 2005, quasi-stable depuis 2006

Declaratif vs. Impératif

Declaratif

- Assertions logiques
- Heuristiques aux feuilles
- “C’est lisible”

Impératif

- Déductions
- Directives partout
- “Ca marche”

Déclaratif pur (Mizar)

```
theorem
 (for x,y holds f.(x+y)+f.(x-y)=2*f.x*g.y) &
 (ex x st f.x<>0) & (for x holds abs(f.x)<=1)
 implies for x holds abs(g.x)<=1
 proof
  assume that
A1: for x,y holds f.(x+y)+f.(x-y)=2*f.x*g.y;
  given z such that
A2: f.z<>0;
  assume
A3: for x being Element of REAL holds abs(f.x)<=1;
  let y such that
A4: abs(g.y) > 1;
  set X = rng abs f, k = upper_bound X, D = abs(g.y);
A5: abs(g.y) > 0 by A4,XREAL_1:2;
A6: X is bounded_above
  proof
   take 1;
   let r be real number;
   assume r in X;
   then consider x being set such that
A7:   x in dom abs f and
A8:   (abs f).x = r by FUNCT_1:def 5;
   abs(f.x) = r by A7,A8,SEQ_1:def 10;
   hence r<=1 by A3,A7;
  end;
A9: for s being real number st s in X holds s<=k/D
  proof
   let s be real number;
   assume s in X;
   then consider x being set such that
B1:   x in dom abs f and
B2:   s = (abs f).x by FUNCT_1:def 5;
B3:   s = abs(f.x) by B1,B2,SEQ_1:def 10;
   reconsider x as Element of REAL by B1;
   set A = f.(x+y), B = f.(x-y), C = abs(f.x);
B4:   abs(A+B)<=abs(A)+abs(B) by COMPLEX1:142;
   abs(A) in X & abs(B) in X by Lm1;
   then abs(A)<=k & abs(B)<=k by A6,SEQ_4:def 4;
   then
B5:   abs(A)+abs(B)<=k+k by XREAL_1:9;
   abs(A+B) = abs(2*f.x*g.y) by A1
   .= abs(2*f.x)*D by COMPLEX1:151
   .= abs(2)*C*D by COMPLEX1:151
   .= 2*C*D by ABSVALUE:def 1;
   then 2*(C*D)<=2*k by B4,B5,XREAL_1:2;
   then C*D<=k by XREAL_1:70;
   then C*D/D<=k/D by A5,XREAL_1:74;
   hence thesis by A5,B3,XCMPLX_1:90;
  end;
  abs(f.z) in X by Lm1;
  then
B6: abs(f.z)<=k by A6,SEQ_4:def 4;
  0<abs(f.z) by A2,COMPLEX1:133;
  then 0<k by B6,XREAL_1:2;
  then k/D<k/1 by A4,XREAL_1:78;
  hence thesis by A9,SEQ_4:62;
 end;
```

Impératif pur (Hol-light)

```
let IMO = prove
 (`!f g. (!x y. f(x + y) + f(x - y) = &2 * f(x) * g(y)) /\
        ~(!x. f(x) = &0) /\
        (!x. abs(f(x)) <= &1)
        ==> !x. abs(g(x)) <= &1`,
  let LL = REAL_ARITH `&1 < k ==> &0 < k` in
  REPEAT STRIP_TAC THEN SPEC_TAC(`x:real`,`y:real`) THEN
  ABBREV_TAC `k = sup (IMAGE (\x. abs(f(x))) (:real))` THEN
  MP_TAC(SPEC `IMAGE (\x. abs(f(x))) (:real)` SUP) THEN
  ASM_SIMP_TAC[FORALL_IN_IMAGE; EXISTS_IN_IMAGE; IN_UNIV] THEN
  ANTS_TAC THENL [ASM SET_TAC[]; STRIP_TAC] THEN
  SIMP_TAC[GSYM REAL_NOT_LT; GSYM NOT_EXISTS_THM] THEN STRIP_TAC THEN
  FIRST_X_ASSUM(MP_TAC o SPEC `k / abs(g(y:real))`) THEN
  SIMP_TAC[NOT_IMP; NOT_FORALL_THM] THEN CONJ_TAC THENL
   [ASM_MESON_TAC[REAL_LE_RDIV_EQ; REAL_ABS_MUL; LL;
      REAL_ARITH `u + v = &2 * z /\ abs u <= k /\ abs v <= k ==> abs z <= k`];
    ASM_MESON_TAC[REAL_NOT_LE; REAL_LT_LDIV_EQ; REAL_LT_LMUL; REAL_MUL_RID; LL;
      REAL_ARITH `~(z = &0) /\ abs z <= k ==> &0 < k`]]);;
```

Impératif structuré

```
let LEMMA1 = prove
 (`(!x y. f(x + y) + f(x - y) = &2 * f(x) * g(y)) /\ (!x. abs(f x) <= &1)
   ==> !l x. abs(f x * (g y) pow l) <= &1`,
  DISCH_THEN(STRIP_ASSUME_TAC o GSYM) THEN INDUCT_TAC THEN
  ASM_SIMP_TAC[real_pow; REAL_MUL_RID] THEN GEN_TAC THEN MATCH_MP_TAC
   (REAL_ARITH `abs((&2 * a * b) * c) <= &2 ==> abs(a * b * c) <= &1`) THEN
  ASM_SIMP_TAC[] THEN FIRST_ASSUM(MP_TAC o SPEC `x + y`) THEN
  FIRST_ASSUM(MP_TAC o SPEC `x - y`) THEN REAL_ARITH_TAC);;

let LEMMA2 = prove
 (`~(x = &0) /\ &1 < abs(y) ==> ?n. &1 < abs(y pow n * x)`,
  SIMP_TAC[REAL_ABS_MUL; REAL_ABS_POW; GSYM REAL_LT_LDIV_EQ;
          GSYM REAL_ABS_NZ; REAL_ARCH_POW]);;

let IMO = prove
 (`!f g. (!x y. f(x + y) + f(x - y) = &2 * f(x) * g(y)) /\
        ~(!x. f(x) = &0) /\
        (!x. abs(f(x)) <= &1)
        ==> !x. abs(g(x)) <= &1`,
  MESON_TAC[LEMMA1; LEMMA2; REAL_NOT_LE; REAL_MUL_SYM]);;
```

Structuré (ssreflect)

```
Lemma IMO : forall f g, (forall x y : R, f (x + y) + f (x - y) == 2 * f x * g y) ->
  ~ (forall x, f x == 0) -> (forall x, `|f x| <= 1) ->forall y, `|g y| <= 1.
Proof.
move=> f g Efg Hf0 Hf1 y.
without loss Hgy: / `|g y| > 0.
   case: (leqr_total 1 `|g y|) => // Hy; apply; exact: ltr_leq_trans _ Hy.
pose k := sup {z | exists x, z == `|f x|}.
have{Hf1} Hfk: `|f _| <= k.
   by move=> x; apply: ubr_sup; [exists (1 : R) => _ [? ->] | exists x].
have{Hf0} Hk: k > 0.
   move=> Hk; elim: Hf0 => x; apply: absr_eq0.
   split; [exact: leqr_trans Hk | exact: leqr_0abs].
suffices: k <= k / `|g y| by rewrite leqr_pdivr // mulrC -leqr_pdivr // pdivrr.
suffices: `|f _| <= k / `|g y| => [Hf|x].
   by apply: leqr_sup_ub => [|_ [x ->]]; first by exists `|f y|; exists y.
suffices: `|f x * g y| <= k by rewrite absr_mul -leqr_pdivr.
have{Efg} Ek: `|2 * f x * g y| <= 2 * k.
   rewrite -Efg mul2r; apply: leqr_trans (absr_add _ _) _; exact: leqr_add.
by do [rewrite -mulrA absr_mul absr_eq ?leqr_pmul2l; auto] in Ek.
Qed.
```

Lisibilité et maintenance

Avec une interface interactive, il n’est pas nécessaire de pouvoir lire les scripts, il faut pouvoir s’y repérer, et les rectifier :

- Lisibilité du contrôle
  - – indentation & terminateurs
- Lisibilité des dépendances
  - – déclarations explicites, noms pertinents
- Orthogonalité
  - – tacticielles, options

Le langage de preuve de ssreflect

- Impératif mais contrôle et dépendances explicites

```
apply: lemma hypA _ => [n lt_0_n | all0].
- by case: n => [|n] // in lt_0_n *; apply: IH.
have{all0}: sum p = 0 by apply: sum_to_0.
```

- Plus de récriture

```
rewrite -!mulnA {2}[ _ * x]addnA.
```

- Plus la réflexion

```
move/andP: hypAB => [hypA hypB].
```

Bureaucratie

prouver P 0 n par récurrence sur n ≥ (i □ 0)

- LCF (Ltac)

  generalize 0, (leq0n n).

  elim n; clear n.

  intros i le_i_n.

  ….

  intros n IHn i le_i_n.

```
push #0, #(leq0n n)

call elim, An, IH

pop i, le_i_n

…

IH: pop n, IHn, i, le_i_n
```

- ssreflect

  elim: n 0 (leq0n n) => [|n IHn] i le_i_n.

```
n, IHn, i, le_i_n := elim(n, 0, leq0n n).
```

elim : n 0 (leq0n n) => [|n IHn] i le_i_n

Contexte et pile

```
n : nat
IHn : forall (x : T) (y : T)(a : seq T),
      #|T| <= #|a| + n -> y \notin a ->
      reflect (dfs_path x y a) (y \in dfs n a x)
y : T
x : T
b : seq T
a : seq T
Hy : (y \in a) = false
Hn : #|T| <= #|a| + n
Ha : a \subset dfs n a x
Hdfs_y : (y \in dfs n a x) = false
============================
 (exists2 x' : T,
      x' \in x :: b & dfs_path x' y a) ->
 exists2 x' : T,
      x' \in b & dfs_path x' y (dfs n a x)
```

=> :

set

have have

Push – call – pop

- Tacticielle push

  : x {y}(f y) (g _) {3}z {Hz}
- Tactiques défectives

  move, case, elim, apply, exact
- Tacticielle pop

  => x -> {y} [|n s] //= *
- Tacticielle in

  in IHn *
- Vues, équations, familles

Induction généralisée

elim: {p}_.+1 {-2}p (ltnSn (size p)) => // n IHn p lepn.

ltnSn (size p) : size p < (size p).+1

- {-2}p généralise tous les p sauf (size p).+1

 p0, size p0 < ….+1 -> …[p p0]

- elim récurre sur _.+1 (i.e., ….+1), {p} supprime p
- Deux sous-buts

 p0, size p0 < 0 -> …, tué par // (car size … < 0 false)

- Continuation unique

n : nat, IHn : p, size p < n -> …, p : …, lepn : size p ≤ n

…

Terminateurs et indentation

- Tacticielle qui termine ou échoue

  by rewrite andbCA; apply: IHn.

- Début de ligne = permet de vérifier l'indentation
- Puces pour éviter l'indentation multiple
- Tacticielles de permutation

  apply IHn; last by exists x.

  case: s => [|x s]; last first.

Assertions / déclarations

- Orthogonalité

```
have p_gt0: p > 0.
have{IHx}: x < y by apply: IHx.
suffices p_div_n: p %| n.
without loss le_xy: x y Hx Hy / x <= y.
set x := _ + (2 * _)
pose fix sz t :=
  if t is Node t1 t1 then (sz t1 + sz t2).+1 else 0.
```

Récriture

- Principale source d’automatisation!
- Unification récriture/réduction + contrôle

```
rewrite 2!(IHn _ x) // {}/h –{3}[t _]/(g y) {x}[f _ x]/=
```

- Filtrage
  - – On recherche une instance exacte du terme de tête (ou une valeur canonique), conversion libre après (mais typée).
- Congruence (paramodulation)

```
congr (_ + _ * _).
```

Une grande preuve…

```
Theorem Sylow's_theorem :
 [/\ forall P, [max P | p.-subgroup(G) P] = p.-Sylow(G) P,
     [transitive (G | 'JG) on 'Syl_p(G)],
     forall P, p.-Sylow(G) P -> #|'Syl_p(G)| = #|G : 'N_G(P)|
  &  prime p -> #|'Syl_p(G)| %% p = 1%N ].
Proof.
pose maxp A P := [max P | p.-subgroup(A) P]; pose S := [set P | maxp G P].
pose oG := orbit 'JG%act G.
have actS: [acts (G | 'JG) on S].
  apply/subsetP=> x Gx; rewrite inE; apply/subsetP=> P; rewrite 3!inE.
  exact: max_pgroupJ.
have S_pG: forall P, P \in S -> P \subset G /\ p.-group P.
  by move=> P; rewrite inE; case/maxgroupP; case/andP.
have SmaxN: forall P Q, Q \in S -> Q \subset 'N(P) -> maxp 'N_G(P) Q.
  move=> P Q; rewrite inE; case/maxgroupP; case/andP=> sQG pQ maxQ nPQ.
  apply/maxgroupP; rewrite /psubgroup subsetI sQG nPQ.
  by split=> // R; rewrite subsetI -andbA andbCA; case/andP=> _; exact: maxQ.
have nrmG: forall P, P \subset G -> P <| 'N_G(P).
  by move=> P sPG; rewrite /normal subsetIr subsetI sPG normG.
have sylS: forall P, P \in S -> p.-Sylow('N_G(P)) P.
  move=> P S_P; have [sPG  pP] := S_pG P S_P.
  by rewrite normal_max_pgroup_Hall ?nrmG //; apply: SmaxN; rewrite ?normG.
have{SmaxN} defCS: forall P, P \in S -> 'C_S(P | 'JG) = [set P].
  move=> P S_P; apply/setP=> Q; rewrite {1}in_setI {1}conjG_fix.
  apply/andP/set1P=> [[S_Q nQP]|->{Q}]; last by rewrite normG.
  apply: val_inj; symmetry; case: (S_pG Q) => //= sQG _.
  by apply: uniq_normal_Hall (SmaxN Q _ _ _) => //=; rewrite ?sylS ?nrmG.
have{defCS} oG_mod: {in S &, forall P Q, #|oG P| %% p = (Q \in oG P) %% p}.
  move=> P Q S_P S_Q; have [sQG pQ] := S_pG _ S_Q.
  have soP_S: oG P \subset S by rewrite acts_orbit.
  have: [acts (Q | 'JG) on oG P].
    apply/actsP=> x; move/(subsetP sQG)=> Gx R; apply: orbit_transr.
    exact: mem_imset.
  move/pgroup_fix_mod=> -> //; rewrite -{1}(setIidPl soP_S) -setIA defCS //.
  rewrite (cardsD1 Q) setDE -setIA setICr setI0 cards0 addn0.
  by rewrite inE set11 andbT.
have [P S_P]: exists P, P \in S.
  have: p.-subgroup(G) 1 by rewrite /psubgroup sub1G pgroup1.
  by case/(@maxgroup_exists _ (p.-subgroup(G))) => P; exists P; rewrite inE.
have trS: [transitive (G | 'JG) on S].
  apply/imsetP; exists P => //; apply/eqP.
  rewrite eqEsubset andbC acts_orbit // S_P; apply/subsetP=> Q S_Q.
  have:= S_P; rewrite inE; case/maxgroupP; case/andP=> _.
  case/pgroup_1Vpr=> [|[p_pr _ _] _].
    move/group_inj=> -> max1; move/andP: (S_pG _ S_Q) => sGQ.
    by rewrite (group_inj (max1 Q sGQ (sub1G Q))) orbit_refl.
  have:= oG_mod _ _ S_P S_P; rewrite (oG_mod _ Q) // orbit_refl.
  by case: {+}(Q \in _) => //; rewrite mod0n modn_small ?prime_gt1.
have oS1: prime p -> #|S| %% p = 1%N.
  move=> pr_p; rewrite -(atransP trS P S_P) (oG_mod P P) //.
  by rewrite orbit_refl modn_small ?prime_gt1.
have oSiN: forall Q, Q \in S -> #|S| = #|G : 'N_G(Q)|.
  by move=> Q S_Q; rewrite -(atransP trS Q S_Q) card_orbit conjG_astab1.
have sylP: p.-Sylow(G) P.
  rewrite pHallE; case: (S_pG P) => // -> /= pP.
  case p_pr: (prime p); last first.
    rewrite p_part lognE p_pr /=.
    by case/pgroup_1Vpr: pP p_pr => [-> _ | [-> //]]; rewrite cards1.
  rewrite -(LaGrangeI G 'N(P)) /= mulnC partn_mul ?cardG_gt0 // part_p'nat.
    by rewrite mul1n (card_Hall (sylS P S_P)).
  by rewrite p'natE // -indexgI -oSiN // /dvdn oS1.
have eqS: forall Q, maxp G Q = p.-Sylow(G) Q.
  move=> Q; apply/idP/idP=> [S_Q|]; last exact: Hall_max.
  have{S_Q} S_Q: Q \in S by rewrite inE.
  rewrite pHallE -(card_Hall sylP); case: (S_pG Q) => // -> _ /=.
  by case: (atransP2 trS S_P S_Q) => x _ ->; rewrite cardJg.
have ->: 'Syl_p(G) = S by apply/setP=> Q; rewrite 2!inE.
by split=> // Q sylQ; rewrite -oSiN ?inE ?eqS.
Qed.
```

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH