SOME COQ PROGRAM OF THE CONGRUENT RESOLUTION TECHNIQUE « PARTICULAR CUT ELIMINATION » OF THE ORIENTED EQUATIONS OF ADJUNCTION AS PRESENTED BY DOSEN

@1337777.NET

1. CONTENTS

Now [1] is successor from the earlier Solution Programme Memo [2], and presents some Coq program of the congruent resolution technique (« particular cut elimination ») of the oriented equations of adjunction as presented by Dosen [1']. Here some alternative (equivalent) terminology (formulation) is given such that each primitive rewrite equation is saturated (augmented) congruently such to command (« operate ») deep at any node (subterm), and this terminology contains some memory which memorize the links (« effects ») on the composition nodes. For example [6] the command F (G f2 o G f1) ~> F (G (f2 o f1)) , is some congruent (bellow F) rewriting of the functoriality of G, and memorizes that the outer composition is linked to the inner composition. In some sense, doing categorial programming may be comparable to doing the common « small-step operational semantics, Hoare memory logic » of C !?1

Congruence is indeed the critical angle of view of the common « small-step semantics » [4]. Also are there correlations between this categorial « congruent resolution » technique and the common « deep inference » technique ? And earlier in [2] , which proved « total/global cut elimination », although some non-saturated congruent « small-step semantics » was held, it is possible to formulate some « big-step semantics » inductive definition which mimicks/simulates the recursion contained the cut-elimination theorem : therefore small-step <-> particular and big-step <-> total/global. Also earlier in the semiassociativity completeness and confluence lemmas, the linking technique is easier to describe because the commands (for example, the associativity bracketing command) are in fact terms (arrow terms) of some easy inductive dependent type.

Also may some shallow embedding onto some logic as « dedukti modulo » [5] be some more sensible compu-logical context than the present deep embedding in Coq ?

SOLUTION PROGRAMME MEMO

Short : some Coq program of the termination technique of the oriented equations of adjunction as presented by Dosen [1']. For example, (counit f <o reflection (unit g)) ~> (f <o reflection g), which is the polymorphic (Yoneda) terminology of the common equations. And the confluence technique and links-model resolution technique for semiassociativity and associativity was already programmed and simultaneously-edited and timed-tutored in [2'] and [3']. These Coq programs use dependent types to encode source-target functions and use inductives to encode grammatical (free) generations of adjunctions.

The Solution Programme lacks to extend and mix these automation programming techniques into the enriched-lax monoidal topology (Tholen's TV) or the n-fold-lax categorial homotopy (Riehl's Kan-obsession) or the dissociativity technique (Dosen's boolean categories and linear logic) or the compositional automation technique (Chlipala's ?functorial? MirrorShard)... ultimately this may converge to the descent technique of Galois. The Solution Programme initiated by Gentzen lacks to comprehend things in different « terminologies » or « formulations » or « coordinates ». This angle of view was continued by Lambek into categories, then by Kelly, then by Dosen-Petric. Einstein's single deepest angle of view was about « coordinates », even before the « spin » or the « entropy » ...

2. CATEGORIAL PROGRAMMING

The flow/path of this text does not lack to exhaust all the possibilities/tree such as Dosen, therefore this presentation shall be smoother. The critical end is that there shall be some << logiciel >> where this categorial programming is done more computationally and more smoothly, and moreover such thing is certainly different than the present deep embedding into Coq.

Memo that the (covariant) Yoneda embedding is the function

```
  | ModCom : forall {UACom}, forall A3, UACom ⊢a A3 -> forall A1, A1 ⊢a UACom -> A1
⊢a A3
```

which says that put some arrow f2 to get the polymorphic higher-order map (f2 <o _) which accepts two arguments : first some object then some arrow. And the reverse is the identity grounding function that puts the polymorphic higher-order map α to get α(1) .

Memo that the Yoneda lemma is some rearrangement (strengthening ?) of the arguments to some functor such to get the << opposite functor >> . For example for any functor F| (on objects F|0)

```
  | F : forall B1 B2, (B1 ⊢b B2) -> F|0 B1 ⊢a F|0 B2
```

and, assuming in Coq that there are coercions to Set and Functions, then the rearrangement of arguments gives

```
    Definition Yoneda_F : forall (B1 : CoModObjects), F|0 B1 -> forall (B2 :
CoModObjects), (B1 ⊢b B2) -> F|0 B2 :=
      (fun (B1 : CoModObjects) (x : F|0 B1) (B2 : CoModObjects) (g : B1 ⊢b B2) =>
         F| B1 B2 g x).
```

Now doing this Yoneda rearrangement on the (left) composition <o really do give the right composition o> . Therefore take the common presentation of categories and functors and natural transformations and adjunctions, where

*** the right composition o> is primitive together with the common equation : f1 o> f2 = f2 <o f1**

*** the counit is primitive but then uncommonly add the map (counit o> _) as primitive together with the common equation : counit = (counit o> _) 1 ... and similar for the unit.**

This presentation looks as

```
  Inductive convMod : forall A1 A2 : ModObjects, A1 ⊢a A2 -> A1 ⊢a A2 -> Prop :=
  |  ModCongCom : forall A2 A3, forall (f2 f2' : A2 ⊢a A3), f2 ≈a f2' -> forall A1,
forall (f1 f1' : A1 ⊢a A2), f1 ≈a f1' -> f2 <a f1 ≈a f2' <a f1'
  |  ModCongRefl1 : forall B1 B2, forall (g g' : B1 ⊢b B2), g ≈b g' -> F| g ≈a F|
g'
  |  ModCongCoUnit : forall A1 A2, forall (f f' : A1 ⊢a A2), f ≈a f' -> φ a> f ≈a φ
a> f'
  |  ModRefl : forall A1 A2 (f : A1 ⊢a A2), f ≈a f
  |  ModTrans : forall A1 A2, forall (uModTrans f : A1 ⊢a A2), uModTrans ≈a f ->
forall (f' : A1 ⊢a A2), f' ≈a uModTrans -> f' ≈a f
  |  ModSym : forall A1 A2,  forall (f' f : A1 ⊢a A2), f ≈a f' -> f' ≈a f
  |  ModCat1Right : forall A1 A2, forall f : A1 ⊢a A2,  f ≈a 1 a> f
  |  ModCat1Left : forall A1 A2, forall f : A1 ⊢a A2,  f ≈a 1 <a f
  |  ModCat2 : forall A3 A4 (f3 : A3 ⊢a A4), forall A2 (f2 : A2 ⊢a A3), forall A1
(f1 : A1 ⊢a A2),
                f1 a> (f3 <a f2) ≈a f3 <a (f1 a> f2)
  |  ModFun1Refl : forall B, 1 ≈a F| (@CoModIden B)
  |  ModFun2Refl : forall B2 B3 (g2 : B2 ⊢b B3) B1 (g1 : B1 ⊢b B2),
                   F| (g2 <b g1) ≈a F| g2 <a F| g1
  | ModNatCoUnit1 : forall A2 A3 (f2 : A2 ⊢a A3) A1 (f1 : A1 ⊢a A2),
                    φ a> (f2 <a f1) ≈a f2 <a φ a> f1
  | ModNatCoUnit2 : forall A2 A3 (f2 : A2 ⊢a A3) A1 (f1 : A1 ⊢a A2),
                    φ a> (f2 <a f1) ≈a (φ a> f2) <a F| G| f1
  | ModRectangle : forall  B1 B2 (g : B1 ⊢b B2) A2 (f : F|0 B2 ⊢a A2),
                    f <a F| g ≈a (φ a> f) <a F| (γ b> g)

  with convCoMod : forall B1 B2 : CoModObjects, B1 ⊢b B2 -> B1 ⊢b B2 -> Prop :=
  |  CoModCongCom : forall B2 B3, forall (g2 g2' : B2 ⊢b B3), g2 ≈b g2' -> forall
B1, forall (g1 g1' : B1 ⊢b B2), g1 ≈b g1' -> g2 <b g1 ≈b g2' <b g1'
  |  CoModCongRefl1 : forall A1 A2, forall (f f' : A1 ⊢a A2), f ≈a f' -> G| f ≈b G|
f'
  |  CoModCongCoUnit : forall B1 B2, forall (g g' : B1 ⊢b B2), g ≈b g' -> γ b> g ≈b
γ b> g'
  |  CoModRefl : forall B1 B2 (g : B1 ⊢b B2), g ≈b g
  |  CoModTrans : forall B1 B2, forall (uCoModTrans g : B1 ⊢b B2), uCoModTrans ≈b g
-> forall (g' : B1 ⊢b B2), g' ≈b uCoModTrans -> g' ≈b g
  |  CoModSym : forall B1 B2,  forall (g' g : B1 ⊢b B2), g ≈b g' -> g' ≈b g
  |  CoModCat1Right : forall B1 B2, forall g : B1 ⊢b B2,  g ≈b '1 b> g
  |  CoModCat1Left : forall B1 B2, forall g : B1 ⊢b B2,  g ≈b '1 <b g
  |  CoModCat2 : forall B3 B4 (g3 : B3 ⊢b B4), forall B2 (g2 : B2 ⊢b B3), forall B1
(g1 : B1 ⊢b B2),
                  g1 b> (g3 <b g2) ≈b g3 <b (g1 b> g2)
  |  CoModFun1Refl : forall A, '1 ≈b G| (@ModIden A)
  |  CoModFun2Refl : forall A2 A3 (g2 : A2 ⊢a A3) A1 (g1 : A1 ⊢a A2),
                     G| (g2 <a g1) ≈b G| g2 <b G| g1
  | CoModNatCoUnit1 : forall B2 B3 (g2 : B2 ⊢b B3) B1 (g1 : B1 ⊢b B2),
                      γ b> (g2 <b g1) ≈b  (γ b> g2) <b g1
  | CoModNatCoUnit2 : forall B2 B3 (g2 : B2 ⊢b B3) B1 (g1 : B1 ⊢b B2),
                      γ b> (g2 <b g1) ≈b  G| F| g2 <b (γ b> g1)
```

```
  | CoModRectangle : forall  A1 A2 (f : A1 ⊢a A2) B1 (g : B1 ⊢b G|0 A1),
                       G| f <b g ≈b G| (φ a> f) <b (γ b> g)
```

Now Dosen says that keeping only 1 , F|0 , F| , '1 , G|0 , G| , (counit a> _) , (unit b> _) , as primitives then everything else can be eliminated (or atomized in case that the generating graph do have non-empty collection of generative arrows). This is immediate for a> , b> , counit , unit , but not immediate for <a and <b . Note that although the common naturality equation
    (conunit a> F G f) = f a> counit
is somehow decreasing, ultimately this Dosen saturation is really lacked.

Note that the solution for Kan extensions is not clear. Memo that any adjunction gives some (<< coreflective >>) left Kan extensions which satisfy some additional polymorphism condition. For example, putting F left adjoint to G, and for L , H variables for functors to Set, such adjunction produces some bijective map going down (for the common map of Kan extensions) but whose reverse map up now has more << form >> :

$$\frac{L \circ G \Rightarrow H \qquad \vee \qquad (\_ * F) \circ (L * unit)}{L \Rightarrow H \circ F \qquad \wedge \qquad (H * counit) \circ (\_ * G)}$$

which makes that the polymorphism equation
    Lan (L' o L) ≅ L o Lan L
hold , but such equation for Kan extensions does not hold in general unless L has some right adjoint.

The Coq text proceeds by defining or proving the following :
    (*) Nodes where self is some Com,
    (*) Nodes where self is some Refl,
    (*) Linking primitives,
    (*) Congruent linking,
    (*) Transitive linking,
    (*) redCongMod congruent reduction, parametrized by particular reductions,
    (*) reduceCongMod collects all the redCong_ instances of redCongMod,
    (*) reduceCongMultiMod do multi of reduceCongMod together with << embedded propositional extensionality >> (saturation of derivations modulo propositional equivalence of linking) as some constructor (non << axiomatic >>),
    (*) Some lemmas : extensionality of links transformations,
    (*) Some lemmas : Derived congruence,
    (*) Notations for categorial adjunctions programming with Coq existential variables and automation, for example :

```
    Lemma example1 : forall A, { f'' : _ & { ΔΔ' : _ | ΔΔ' ⊦ f'' *⌐a F| (γ| (G|
(@ModIden A) <b G| 1) <b G| 1) }} .
    Proof.
```

```
      intros. eexists. eexists.
      refine (INIT F| (γ| (G| (@ModIden A) <b G| 1) <b G| 1) ;;
fun2' at  ( (bottomF (leftComCoMod (bottomγ  (selfCoMod))))) ;;
fun1' at ( (bottomF_Refl (rightComCoMod_Refl selfCoMod_Refl))) ;;
cat1r at  ( (bottomF (leftComCoMod (bottomγ  (bottomG (selfMod)))))) ;;
INIT F| (γ| (G| 1) <b '1) ;;
fun1' at ((bottomF_Refl (leftComCoMod_Refl (bottomγ_Refl (selfCoMod_Refl))))) ;;
NOOP );
      shelveLinks;
      repeat (intuition eauto; econstructor).
      Show Proof.
    Qed.
```

(*) Derived congruences on the multi,
(*) Completeness lemma of new saturated-congruent reduce relation (for particular cut elimination) under the precedent reduce relation (for total cut elimination), note that this lemma is comparable to the << development lemma >> for semiassociativity, and is also comparable to the filtered/directed colimit technique for presentable objects in categories, which are reflections of free colimits in the polymorphic (Yoneda) representation ... memo that another name for filtered colimit is inductive colimit !
(*) Soundness lemma of new saturated-congruent reduce relation for the original conversion relation.
(*) And finally, the Coq proposition of the congruent resolution is :

```
    Fixpoint solveCongMod len {struct len} : forall A1 A2 (f : A1 ⊢a A2) (n :
nodesMod f) (H_gradeNodeTotalMod : gradeMod f + gradeNodeMod n <= len),
                                                { fSol : A1 ⊢a A2 & { Δ : nodesMod f
->  nodesMod fSol -> Prop |
                                                                        Δ ⊨ fSol *↜a
f /\ forall n', ~ Δ n n' }}
    with solveCongCoMod len {struct len} : forall B1 B2 (g : B1 ⊢b B2) (m :
nodesCoMod g) (H_gradeNodeTotalCoMod : gradeCoMod g + gradeNodeCoMod m <= len),
                                              { gSol : B1 ⊢b B2 & { Λ : nodesCoMod g
->  nodesCoMod gSol -> Prop |
                                                                      Λ ⊨ gSol *↜b
g /\ forall m', ~ Λ m m' }}.
```

3. RECURSION PROGRAMMING

Moving from purely sequential categorial programming, the question is whether the recursive programming techniques of nested memory may converge to categorial programming.

For example, to program the enumeration of combinations of selections from some data list, then 3 nested accumulators are held (in addition to the heights accumulator). This example is more technical than the common lexer which only hold 2 nested accumulators or more technical than the common recursive-descent parser whose main technique is backtracking.

```
Compute List.rev ( enumerate [ "a" ; "b" ; "c" ; "d" ; "e" ; "f" ; "g" ] [ 0 ; 1 ;
```

```
0 ] ).

(**
     = [["a"]; ["b"; "c"]; ["d"]]; [["a"]; ["b"; "c"]; ["e"]];
       [["a"]; ["b"; "c"]; ["f"]]; [["a"]; ["b"; "c"]; ["g"]];
       [["a"]; ["b"; "d"]; ["e"]]; [["a"]; ["b"; "d"]; ["f"]]; ...  **)

Definition enumerate {A :Set} (data : list A) (heights : list nat) : list (list
(list A)) :=
  let enumerate :=
      (fix  enumerate data heights selection combination enumeration {struct
data} :=
         match data with
           | [] =>
             match heights with
               | [] => ((List.rev combination) :: enumeration)
               | _ => enumeration
             end
           | x :: restData =>
             match heights with
               | [] => ((List.rev combination) :: enumeration)
               | 0 :: restHeights =>
                 enumerate restData heights selection combination
                       (enumerate restData restHeights [] ((List.rev (x ::
selection)) :: combination) enumeration)
               | S curHeight :: restHeights =>
                 enumerate restData heights selection combination
                       (enumerate restData (curHeight :: restHeights) (x ::
selection) combination enumeration)
             end
         end)
  in
  enumerate data heights [] [] [].
```

4. MACLANE ASSOCIATIVITY COHERENCE

5. DOSEN SEMI-ASSOCIATIVITY COHERENCE

REFERENCES

[1] 1337777.net, « https://github.com/1337777/dosen/blob/master/dosenSolution1.v »
[2] 1337777.net, « https://github.com/1337777/dosen/blob/master/dosenSolution.v »
[4] https://www.cis.upenn.edu/~bcpierce/sf/current/Smallstep.html
[5] Saillard, http://www.cri.ensmp.fr/people/saillard/Files/thesis.pdf
[1'] Dosen, «Cut Elimination in Categories» In: https://books.google.com/books?isbn=9401712077
[2'] 1337777.net, «MACLANE PENTAGON COHERENCE IS SOME RECURSIVE COMONADIC DESCENT» In: Studies in Logic, SYSU
[3'] 1337777.net, «1337777.info»