An Implementation of Groebner Synthesis in *Theorema*

Adrian Craciun
RISC−Linz / IeAT Timisoara
acraciun@{risc.uni−linz.ac.at, ieat.ro}

INTAS meeting
Dec. 10 2006
Timisoara

Outline of the Talk

- **Implementation of the Lazy Thinking Synthesis Method In *Theorema***
  - ⟶ **Context**
  - ⟶ **Cascade**
  - ⟶ **Proof Analyzer**
  - ⟶ **Conjecture Generator**
- **Using the Lazy Thinking Implementation for Synthesis of a GB**
  - ⟶ **Problem**
  - ⟶ **Knowledge Base**
  - ⟶ **Algorithm Scheme**
  - ⟶ **Exploration using Lazy Thinking**
- **Conclusions, Future Work**

Lazy Thinking: Context

- **Computer Supported, Knowledge (Schemes) Based Exploration of Mathematical Theories (BB)**

⟶ Explore theories in exploration rounds (BB: "creativity spiral");
⟶ At each round add a new notion to the theory:
− by the application of a definition scheme;
− by solving a problem
.... and then
− explore the new notion (typical properties, by proposition schemes, interaction with known notions).

- **Solving Problems in an Algorithmic Fashion: Lazy Thinking**

Lazy Thinking: Informal Description

⟶ Start from a **formal (predicate logic) specification** of the problem:

$$\forall_{I[x]} P[x, A[x]]$$

⟶ Try out "*algorithm schemes*"

DO

⟶ attempt to **prove (automatically) the correctness theorem** for the algorithm scheme w.r.t to the given specification is started. This proof attempt **will fail** because nothing is known about the unspecified subalgorithms.

⟶ *analysis of the failing proof situation* + *conjecture generating algorithm*, and the proof gets over the failure

UNTIL the proof is completed (or give up).

⟶ RESULT: The algorithm defined using the scheme satisfies the specification, provided that there exists algorithms that satisfy the specifications generated by the conjecture generator.

To continue (and complete the synthesis):

⟶ Retrieve from the knowledge base algorithms that satisfy the specifications generated, or

⟶ Apply another round of lazy thinking.

Lazy Thinking: Implementation

To implement the method, one has to implement:

⟶ The Cascade Mechanism,

⟶ Failure Analysis of Proofs,

⟶ Conjecture Generator.

Theorema: Preliminaries

⟶ Proving in *Theorema*:

```
? Prove

Prove[f, using → kb, by → P, ProverOptions → {po}, transform-by → T,
  TransformerOptions → {to}, show-by → S, ShowOptions → {so}, opts]
  proves ‘f‘ w.r.t. the knowledge base ‘kb‘ using the prover ‘P‘.
The list ‘{po}‘ contains options for ‘P‘. In fact, ‘P[f,kb,po]‘ is called.

The resulting proof object is transformed (e.g. simplified) by the
  transformation function ‘T‘. The list ‘{to}‘ contains options for ‘T‘.
  In fact, ‘T[#,to]&‘ is called on the result returned by the prover ‘P‘.

Finally, the resulting proof object is displayed by ‘S‘.
  The list ‘{so}‘ contains options for ‘S‘. In fact, ‘S[#,so]&
  ‘ is called on the result returned by the transformation ‘T‘.

Default values for all options are provided.
```

⟶ Proof Object:

- ⟶ AND−OR(−IF) tree of proof situations;
- ⟶ proof situation: {goal, kb};
- ⟶ inference rules: transform proof situations
  - ⟶ modify the goal,
  - ⟶ modify the knowledge;

Lazy Thinking: Cascade (I)

? CascadeLT

```
The cascade prover implementing
  the Lazy Thinking Theory Exploration paradigm.
   CascadeLT[Prover_, ConjectureGenerator_, nS_,
  auxNS_, kConjectures_, nConjectures_]

  Arguments:
       - Prover_: the Theorema user prover employed in the exploration;
      - ConjectureGenerator_: analyses the failing proof situations
  and generates the conjectures so that proofs go through;
      - nS_ : the unknown notion in the Lazy
  Thinking exploration situation;
       - auxNS_ : a list containing the auxiliary
  notions in the Lazy Thinking exploration situation;
       - kConjectures_ : a list containing conjectures
  that involve only notions known in the exploration;
       - nKonjectures_ : a list containing conjectures that involve
  the auxiliary notions (requirements for the auxiliary notions);

 The Cascade prover is called in the Theorema Prove command.
  Prove[theorem_, using → kBase_, by → CascadeLT[...], ...]

 kBase contains the basic theory
  knowledge and the algorithm type knowledge.
```

```
Prove[Theorem["Groebner Bases specification: is Groebner Base"],
 using → Theory["pre GB1"],
  by → CascadeLT[BasicProver, GenerateConjectures,
    {}, {lc, df}, •asml[], •asml[]],
 ...]
```

Lazy Thinking: Cascade (II)

```
CascadeLT[Prover_, ConjectureGenerator_, nS_, auxNS_,
   kConjectures_, nConjectures_][g_•lf, kb_•asml] := Module[{…},

  proof-object = Prover[g, kb, userBui, bui, properties, opts];
  If[ProofValue[proof-object] === "proved",
   Display[proof-object];
   Print["\n LAZY THINKING ::::: The proof is completed!!!!"];
   Return[{"proved", kb, nS, auxNS, kConjectures, nConjectures }],

   Display[proof-object];
   conjecture =
    ConjectureGenerator[proof-object, nS, auxNS, nConjectures]
  ];

  If[KnownSymbols[conjecture],
   AppendTo[kConjectures, conjecture];
  ];

  If[conjecture === "nothing",
   Print["\n LAZY THINKING ::::: Thinking did not pay off this
      time, although it usually does."]; Return[{"failed"}]];

  newKb = Append[kb, conjecture];
  newNConjectures = Append[nConjectures, conjecture];
 ];
Print["LAZY THINKING::::: The proof fails.
\n After analysing the failing proof, the following
    conjecture(s) is(are) added to the knowledge base: \n ",
  currentNewConjectures /. •asml → Sequence,
  "\n Now attempt the proof with the updated knowledge base. "];

CascadeLT[Prover, ConjectureGenerator, nS,
  auxNS, kConjectures, newNConjectures][g, newKb];
```

Lazy Thinking: Failure Analyzer

⟶ Works on the proof object structure (proof tree);
⟶ Not available to the user, but called by the ConjectureGenerator;

How it works:

⟶ The initial proof situation is {goal, initial_knowledgeBase}

⟶ Get the failing proof situation: {failing_goal (formula), failing_knowledgeBase (list of formulae) };

Remark:

− failing_goal will be a variable−free formula (by the time of failure of the proof),

− failing_knowledgeBase will contain the initial knowledge base (with which the proof was started), plus the temporary knowledge collected along the path from the initial proof situation to the failing one ;

⟶ Slect from the temporary knowledge, those formulae that contain no variables, i.e. ground_temp−Knowledge, a list of variable-free formulae;

⟶ Return {failing_goal, ground_tempKnowledge}.

… but not the end of the story…

Lazy Thinking: Conjecture Generation (I)

? GenerateConjectures

```
[proof object, desired symbol, list of auxiliary symbols, list of
  conjectures] generates conjectures in a Lazy Thinking exploration.
      - proof object typically corresponds to a failed proof attempt,
     - desired symbol denoted
  the function symbol that is being synthesized,
     - the list of auxiliary symbols is taken from
  the algorithm scheme (Lazy Thinking Cascade call).
```

How it works:

⟶ Calling FailureAnalyzer[proof object], yields {failing_goal, ground_tempKnowledge};

⟶ From ground_tempKnowledge filter out the formulae not connected to the goal: filtered_temp−Knowledge (list of ground formulae);

⟶ Construct the skeleton of the conjecture we want to generate:

conjunction of filtered_tempKnowledge ⇒ failing_goal

⟶ Generalize terms in the skeleton to obtain the conjecture, by employing generalization heuristics.

Remark. We have arbitrary but fixed constants in the skeleton, not just any terms!!!

Lazy Thinking: Conjecture Generation (II)

Generalization Strategy:

⟶ Make arbitrary but fixed constants variables:

- works in simple situations, usually not when algorithm schemes are involved;

⟶ When algorithm schemes are involved:

- first generalize terms of the form A[…, S1[…], …] to variables,
- whatever abf constants are left, generalize them to variables.
- works when simple recursive schemes are involved (e.g. divide-and-conquer):

$$\left(\bigwedge \begin{cases} \text{is-tuple}[X_0] \\ \neg\,\text{is-trivial-tuple}[X_0] \\ \text{is-tuple}[S[\text{ls}[X_0]]] \\ \text{is-tuple}[S[\text{rs}[X_0]]] \\ \text{ls}[X_0] \approx S[\text{ls}[X_0]] \\ \text{rs}[X_0] \approx S[\text{rs}[X_0]] \end{cases}\right) \Rightarrow X_0 \approx c[S[\text{ls}[X_0]], S[\text{rs}[X_0]]]$$

$$\underset{\substack{\text{is-tuple}[X,Y,Z] \\ \neg\text{is-trivial-tuple}[X]}}{\forall} \left(\left(\bigwedge \begin{cases} Y \approx \text{ls}[X] \\ Z \approx \text{rs}[X] \end{cases}\right) \Rightarrow X \approx c[Y, Z]\right)$$

⟶ Does this work for the Groebner synthesis too?

- The Problem of Groebner Bases
- Building-up the Knowledge Base
- Algorithm Scheme Critical−Pair/Completion: Preprocessed
- Applying Lazy Thinking: First Round of Exploration
- Conjecture Generation: Revisited

Conjecture Generation: Groebner Bases Synthesis (I)

FailureAnalyser[$TmaProofObject]

```
{{•lf[23, trd[rd[p_0, g_0], CPC[F_0]] = trd[rd[p_0, g_1], CPC[F_0]],
   •finfo[]], •asml[•lf[12.1, g_1 ∈ CPC[F_0], •finfo[]],
   •lf[12.2, lp[g_1] | p_0, •finfo[]],
   •lf[12.3, f2_0 = rd[p_0, g_1], •finfo[]],
   •lf[13.1, trd[rd[lc[g_0, g_1], g_0], CPC[F_0]] =
     trd[rd[lc[g_0, g_1], g_1], CPC[F_0]], •finfo[]],
   •lf[2.1, is-Noetherian[⟶_{CPC[F_0]}], •finfo[]],
   •lf[4.1, is-pp[p_0], •finfo[]], •lf[4.2, p_0 ⟶_{CPC[F_0]} f1_0, •finfo[]],
   •lf[4.3, p_0 ⟶_{CPC[F_0]} f2_0, •finfo[]],
   •lf[8.1, g_0 ∈ CPC[F_0], •finfo[]], •lf[8.2, lp[g_0] | p_0, •finfo[]],
   •lf[8.3, f1_0 = rd[p_0, g_0], •finfo[]]]}}
```

GenerateConjectures[$TmaProofObject, {}, {lc, df}, {}]

```
•lma[conjecture$296, •range[], True, •flist[
  •lf[conjecture$296.1, ∀_{F4,g13,g14,p6} ((lp[g13] | p6) ⋀ (lp[g14] | p6) ⋀
       is-Noetherian[⟶_{CPC[F4]}] ⋀ is-pp[p6] ⋀ g13 ∈ CPC[F4] ⋀
       g14 ∈ CPC[F4] ⋀ (trd[rd[lc[g13, g14], g13], CPC[F4]] =
         trd[rd[lc[g13, g14], g14], CPC[F4]]) ⇒
      (trd[rd[p6, g13], CPC[F4]] = trd[rd[p6, g14], CPC[F4]]))]]]
```

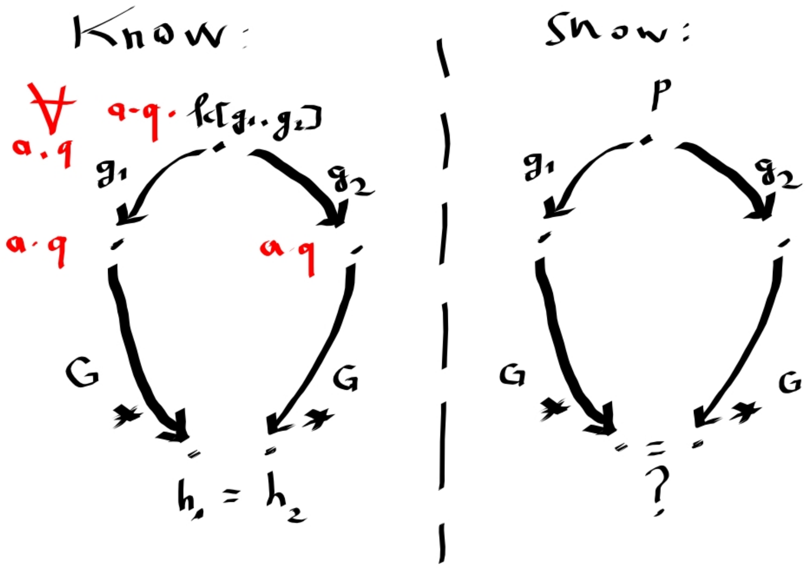
```
•lma["conjecture$296", •range[], True, •flist[
  •lf["conjecture$296.1", ∀_{F4,g13,g14,p6} ((lp[g13] | p6) ⋀ (lp[g14] | p6) ⋀
       is-Noetherian[⟶_{CPC[F4]}] ⋀ is-pp[p6] ⋀ g13 ∈ CPC[F4] ⋀
       g14 ∈ CPC[F4] ⋀ (trd[rd[lc[g13, g14], g13], CPC[F4]] =
         trd[rd[lc[g13, g14], g14], CPC[F4]]) ⇒
      (trd[rd[p6, g13], CPC[F4]] = trd[rd[p6, g14], CPC[F4]]))]]]
```

Conjecture Generation: Groebner Bases Synthesis (II)

But what does this mean?

Conjecture Generation: New Strategy (I)

How does the Groebner proof work?



Lazy Thinking: Failure Analisys and Conjecture Generation (Revisited)

Failure Analysis

⟶ The initial proof situation is {goal, initial_knowledgeBase}

⟶ Get the failing proof situation: {failing_goal (formula), failing_knowledgeBase (list of formulae) };

Remark:

− failing_goal will be a variable−free formula (by the time of failure of the proof),

− failing_knowledgeBase will contain the initial knowledge base (with which the proof was started), plus the temporary knowledge collected along the path from the initial proof situation to the failing one ;

⟶ Slect from the temporary knowledge, those formulae that contain no variables, i.e. ground_temp−Knowledge, a list of variable-free formulae

AND

universally quantified formulae that will not be used for knowledge rewriting (exclude ∀(…⇒ …))

⟶ Return {failing_goal, tempKnowledge}.

Conjecture Generation:

⟶ Calling FailureAnalyzer[proof object], yields {failing_goal, tempKnowledge};

⟶ From tempKnowledge filter out the formulae not connected to the goal and those universally quantified that do not match the goal:

filtered_tempKnowledge ;

⟶ Construct the skeleton of the conjecture we want to generate:

conjunction of filtered_tempKnowledge ⇒ existential_formula

⟶ Generalize terms in the skeleton to obtain the conjecture, by employing generalization heuristics.

Conjecture Generation: New Strategy (II)

```
FailureAnalyser[$TmaProofObject]
```

```
{{•lf[23, trd[rd[p0, g0], CPC[F0]] = trd[rd[p0, g1], CPC[F0]],
   •finfo[]], •asml[•lf[12.1, g1 ∈ CPC[F0], •finfo[]],
   •lf[12.2, lp[g1] | p0, •finfo[]],
   •lf[12.3, f20 = rd[p0, g1], •finfo[]],
   •lf[13.1, trd[rd[lc[g0, g1], g0], CPC[F0]] =
     trd[rd[lc[g0, g1], g1], CPC[F0]], •finfo[]],
   •lf[16, ∀_{a,q} (trd[rd[a * q * lc[g0, g1], g0], CPC[F0]] =
       trd[rd[a * q * lc[g0, g1], g1], CPC[F0]]), •finfo[]],
   •lf[17, ∀_{a,q} (a * q * trd[rd[lc[g0, g1], g0], CPC[F0]] =
       a * q * trd[rd[lc[g0, g1], g1], CPC[F0]]), •finfo[]],
   •lf[2.1, is-Noetherian[⟶CPC[F0]], •finfo[]],
   •lf[24, ∀_{a,q} (a * q * rd[p0, g1] = a * q * rd[p0, g1]), •finfo[]],
   •lf[25, ∀_{a,q} (a * q * f20 = a * q * rd[p0, g1]), •finfo[]],
   •lf[4.1, is-pp[p0], •finfo[]], •lf[4.2, p0 ⟶CPC[F0] f10, •finfo[]],
   •lf[4.3, p0 ⟶CPC[F0] f20, •finfo[]],
   •lf[8.1, g0 ∈ CPC[F0], •finfo[]], •lf[8.2, lp[g0] | p0, •finfo[]],
   •lf[8.3, f10 = rd[p0, g0], •finfo[]],
   •lf[9, ∀_{a,q} (a * q * f10 = a * q * rd[p0, g0]), •finfo[]]]}}
```

```
GenerateConjectures[$TmaProofObject, {}, {lc, df}, {}]
```

```
•lma["conjecture$295", •range[],
 True, •flist[•lf["conjecture$295.1",
    ∀_{g11,g12,p5} ((lp[g11] | p5) ∧ (lp[g12] | p5) ∧ is-pp[p5] ⇒
       ∃_{a,q} (p5 = a * q * lc[g11, g12]))]]]
```

Conclusions, Future Work

⟶ New conjecture generation strategy to deal with the Groebner bases synthesis;

⟶ KNOWLEDGE (ALGORITHM) SCHEMES

⟶ Theory exploration (lazy thinking + schemes → invention)

⟶ ? efficiency

⟶