GCD — a Case Study on Lucas-Interpretation

Walther Neuper
Institute for Software Technology
University of Technology
Graz, Austria
neuper@ist.tugraz.at

1 Status Quo of Prototyping Lucas-Interpretation

Lucas-Interpretation [5] combines computation and deduction such that a learner has free choice in interaction while solving problems in applied mathematics: a next step can be requested from the system *and/or* can be input with feedback from the system. Thus interactive support in stepwise problem solving comes close to traditional paper and pencil work. Next steps are computed by a program, while interpretation works stepwise like in a debugger and maintains an environment together with a logical context. The latter provides automated provers with data to check user input by establishing (or not establishing) deductions of input formulas from the context.

The prototype of Lucas-Interpretation in the ISAC project[1] raises several open research questions. One of them are the limits of "next-step-guidance": Which kinds of input guarantee the interpreter to resume execution? So far, there is one positive answer [2], lemma 7 on p.182.

Another open question is revealed in the proof of the above mentioned lemma, which involves reachability, not yet tackled in Isabelle [8]: How relates logical consistency of a calculation with the operational semantics of the respective program? Interest on clarification of theoretical foundations for Lucas-Interpretation is motivated by a case study [7]: this study revealed that ISAC's programming language is too complicated to hand over authoring to the public.

A promising way to make ISAC's programming language easier to use is to approach Isabelle's function package [4]. Most urgent for practical use is inclusion of rewriting (see example [5].p.92) into the function package; however, this inclusion will introduce a new class of termination proofs. Another kind of examples are engineering problems like [5].p.85; however, for these examples logical consistency is still unclear: clarification involves operational semantics of the programming language, which has not yet been tackled.

The present study investigates a class of examples, which is not affected by either difficulties, neither be rewriting and termination nor by engineering problems with involved semantics.

2 Case Study on Interaction in GCD Calculation

Calculation of the greatest common divisor (GCD) of polynomials is a typical example from computer algebra. These examples are characterised by lots of proved properties, see for instance for Isabelle's polynomials [2].

[1] http://www.sit.tugraz.at/isac
[2] http://isabelle.in.tum.de/dist/library/HOL/HOL-Library/Polynomial.html

Submitted to:
THedu'14

2.1 Make coefficient type closed under division in polynomial rings

In the present Isabelle distribution polynomial division requires coefficients to form a field. Mathematicians prefer rings to be closed under division for both for the polynomial ring as well as for the coefficient ring. This property of division with remainder carries over to the GCD.

In order to achieve this closure we introduce polynomial remainder sequences according to [9].p.84. This requires only little adaptions of existing code: `pdivmod_rel` is renamed to `prsmod_rel`, `div` is renamed to `prs` (polynomial remainder sequence) and all other identifiers are left as they are in the distribution, while respective code is slightly changed.

This is the abstract definition as relation using multiplication as weakly inverse operation ("?" is ∃, "%" is λ), `smult` is scalar multiplication:

```
definition prsmod_rel :: "'a::ring_div => 'a poly => 'a poly => 'a poly => 'a poly => bool"
  where "prsmod_rel a x y q r <->
    smult a x = q * y + r /\ (if y = 0 then q = 0 else r = 0 \/ degree r < degree y)"

definition prs :: "'a::ring_div poly => 'a poly => ('a  'a poly)" (infixl "prs" 70)
  where "x prs y = (THE (a, q). ?r. pdivmod_rel a x y q r)"

definition mod :: "'a::ring_div poly => 'a poly => 'a poly" (infixl "mod" 70)
  where "x mod y = (THE r. ?a q. prsmod_rel a x y q r)"
```

For these definitions the usual algebraic properties can be proved and finally a code lemma provides executability, which automatically transfers to `prs` and `mod`:

```
lemma prsmod_fold_coeffs [code]:
  "prsmod p q = (if q = 0 then ((0, 0), p)
    else fold_coeffs (%a ((m, s), r).
      let (m', n) = fact_quot (coeff (pCons a r) (degree q)) (coeff q (degree q))
      in ((if m' = 0 then m else m * m', pCons n (smult m' s)),
        smult m' (pCons (a * m) r) - smult n q)) p ((1, 0), 0))"
```

Now the Euclidean algorithm on $R[x]$ can be defined similar to ring R, just using the remainder function `mod`. Function `primitive` produces a polynomial of coefficients with GCD equal to one. Note, that `gcd` is used for different types: once for the coefficients (`gcd da db`), once for polynomials (`gcd a b`):

```
function gcd' :: "'a::ring_div poly => 'a poly => 'a poly"
  where "gcd' a b = (if b = 0 then a else gcd' b (a mod b))"

definition gcd :: "'a::{ring_div,Gcd,equal,ord,idom} poly => 'a poly => 'a poly"
  where "gcd a b =
    (let (da, a') = primitive a; (db, b') = primitive b; (_, g) = primitive (gcd' a' b')
    in smult (gcd da db) g)"
```

Traditional CA, for instance [9], prefers to compute polynomial GCD not simply by polynomial remainder sequences as shown above. The reason was, that coefficients get very large very quickly, see the example on p.3. Since there are arbitrary precision numbers in Isabelle (and SML), this is no problem anymore.

2.2 Examples for interaction on Euclid's algorithm

A primary educational aim of Lucas-Interpretation is to make algorithms accessible to learning by step-wise interaction. This aim is accomplished straight forward in rewriting, i.e. in algebraic simplification, in differentiation, etc. But this aim is not straight forward in computer algebra. Below we investigate two kinds of interaction, one which supports learning by observation primarily and another which supports learning by doing steps by writing expressions like in paper and pencil work.

Learning by observation of invariants Understanding division and the related generation of polynomial remainder sequences is supported by looking at the invariant (the fixpoint) of the recursion. Below is an example for $(1+2x+3x^2+4x^3+5x^4+6x^5)\ \mathit{prs}\ (7+8x+9x^2)$, which yields the final remainder $-3917-1972x$ in the sequence; the polynomials are represented by lists of integers[3]:

```
[1, 2, 3, 4, 5, 6] . 3    = [0, 0, 0, 2]         * [7, 8, 9] + [3, 6, 9, ~2, ~1, 0]
[1, 2, 3, 4, 5, 6] . 27   = [0, 0, ~1, 18]       * [7, 8, 9] + [27, 54, 88, ~10, 0, 0]
[1, 2, 3, 4, 5, 6] . 243  = [0, ~10, ~9, 162]    * [7, 8, 9] + [243, 556, 872, 0, 0, 0]
[1, 2, 3, 4, 5, 6] . 2187 = [872,~90,~81, 1458] * [7, 8, 9] + [~3917, ~1972, 0, 0, 0, 0]
```

And the Euclidean algorithm for the example given in [9].p.94, when shown the invariant at each recursion, leads to the result $\mathit{gcd}\ (-18-15x-20x^2+12x^3+20x^4-13x^5+2x^6)\ (8+28x+22x^2-11x^3-14x^4+x^5+2x^6)=\ldots$, where a closer look identifies the result as $-2-x+x^2$:

```
[~18, ~15, ~20, 12, 20, ~13, 2]  .  1      =  [1]           *  [8, 28, 22, ~11, ~14, 1, 2]   +  [~26, ~43, ~42, 23, 34, ~14]
[8, 28, 22, ~11, ~14, 1, 2]      .  98     =  [~41, ~14]    *  [~26, ~43, ~42, 23, 34, ~14]  +  [~282, 617, ~168, ~723, 344]
[~26, ~43, ~42, 23, 34, ~14]     .  59168  =  [787, ~2408]  *  [~282, 617, ~168, ~723, 344]  +  [~1316434, ~3708859, ~867104, 1525321]
[~282, 617, ~168, ~723, 344]     .  6783102487  =  [~2345549, 1529768]  *  [~1316434, ~3708859, ~867104, 1525321]  +  [~5000595353600, ~2500297676800, 2500297676800]
[~1316434, ~3708859, ~867104, 1525321]  *  7289497600  =  [1919, 4447]  *  [~5000595353600, ~2500297676800, 2500297676800]  +  []
```

For somebody, who is familiar with the two algorithms, the iterations of the invariants are informative. But a learner, who approaches the algorithm first time, needs some additional information.

In both examples the "=" might immediately motivate the learner to check the equality of the two sides — so appropriate tools for calculation are required; Isabelle's tool for such trials is given by *notepad begin ... end*; an analogous tool must be available in any frontend for Lucas-Interpretation. And there are other parts of the invariant, which are not immediately evident: In the first example the remainder's (at the right margin) degree decreases until it is smaller than the divisor's degree; in the second example the second summand's (at the right margin) degree decreases and the algorithm stops if it becomes zero — these informations are part of the logical context, so specific elements of the context must be accessible easily and probably specific hints might be useful additionally.

Learning by step-wise input Another promise of Lucas-Interpretation is support for activity based learning in stepwise problem solving close to traditional paper and pencil work supervised by a human tutor. Below we take `div` as a special case of `prs`.

There are courses, where division of polynomials is taught and exercised manually. One can also imagine, that at some occasion, for instance encountering partial fraction decomposition for some integral, a student wants to know in detail, how division works on polynomials. Or somebody wants to compare the division algorithm for polynomials with the division of integers. All these cases would expect steps as follows for a smaller example than those above, for instance $(4+3x+2x^2+x^3)\ \mathit{div}\ (2+x) = (3+x^2)-\frac{2}{2+x}$:

[3]The factors on the left-hand side are introduced in order to stay within the ring of integers.

```
    [4, 3, 2, 1] . 1
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [4, 3, 2, 1]
  - [0, 0, 2, 1] <--- [2, 1] . 1
    ----------------------------
    [4, 3, 0, 0]                        quot = [0] @ [1] @ (1 . [])
    [4, 3] . 1
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    [4, 3]
  - [6, 3] <--- [2, 1] . 3
    ----------------------------
    [~2, 0]                             quot = [] @ [3] @ (1 . [0, 1])
```

The traditional handwritten algorithm represents polynomials with the highest power first; here the order is reversed as usual in computer mathematics; so there are open questions for detailed design and for cooperation with educators in order to meet learners' requirements.

The left column is what pupils are taught to do by hand (except the additional factors $\cdot 1$ and $\cdot 3$ generalising to polynomial remainder sequences). So, at this level of Lucas-Interpretation (and not on the pedagogical level of user guidance) for all terms like $[4,3,2,1]$, $\cdot 1$, $[0,0,2,1]$ etc both should be possible: input by the learner or output by the system. The right columns is up to debate; as given above it just reflects the details of constructing the quotient.

Step-wise input to Euclid's algorithm on polynomials seems not reasonable, see the example on p.3. If such input is indeed requested, then it seems feasible only together with a notepad for auxiliary calculations.

3 Conclusions for Further Development

The findings from the case study are the following:

(a) Invariants, in computer science known for representing the essence of algorithms, turn out helpful for understanding mathematical algorithms, too. Instantiation of an invariant with the values at each iteration of an algorithm gives a sequence of steps. This sequence is instructive for learners asking how a result is calculated step by step from the input. The findings originate from polynomial division and GCD and seem to transfer to a large class of algorithms in computer algebra.

(b) Polynomial division done by hand involves steps far off the code in a streamlined program in a functional language. Also this finding carries over to the few algorithms in computer algebra, where calculation by hand is exercised traditionally.

(c) Both of the findings, (a) and (b) suggest to make a kind of notepad available for investigations and trials during study of invariants as well as during stepwise calculation.

These findings relate to the present state in the prototyped Lucas-Interpretation:

ad (a) A tactic INVARIANT can be included in ISAC's programming language. This tactic takes a term containing the respective variables as arguments and returns no value; rather, Lucas-Interpretation identifies INVARIANT as a breakpoint and handles interaction as a side-effect: it hands over control to the learner for the purpose of investigation.

ad (b) The steps of a traditional calculation are unrelated to the code of an algorithm; so it seems best to relate these steps to some single location in the program, probably a subprogram. Such a subprogram then is free to implement any pedagogically motivated steps; this architectural solution seems to respect the interface to ISAC's user guide, which is implemented in Java and resides in the front-end.

ad (c) A notepad will be a special kind of "worksheet" in ISAC. Such a notepad does not start with a formal specification for a subsequent calculation. Rather, it might inherit the present logical context from the worksheet the trials come from.

The findings and their relation to the present state suggest these steps in future ISAC development:

1. Extend ISAC's programming language with a new tactic INVARIANT and implement it as a breakpoint for Lucas-Interpretation.
2. Extend ISAC's programming language with function definitions as close as possible to Isabelle's function package. Evaluate these functions in the Lucas-Interpreter reusing machinery of Isabelle's code generator [3].
3. Narrow existing programs in ISAC gradually to the syntax of the function package; parallel to this "standardization" of the programming language adapt the code generator's machinery to cooperate with the Lucas-Interpreter.

These steps shall increase the usability of ISAC's programming language and thus promote spreading of a new generation of educational math software [6].

References

[1] (eduTPS 2013): *Theorem-Prover based Systems for Education.* eJMT, the Electronic Journal of Mathematics & Technology. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".

[2] Gabriella Daróczy & Walther Neuper (2013): *Error-Patterns within "Next-Step-Guidance" in TP-based Educational Systems.* [1], pp. 175–194. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".

[3] Florian Haftmann (2013): *Code generation from Isabelle/HOL theories.* Theorem Proving Group at TUM, Munich. Available at `http://isabelle.in.tum.de/dist/Isabelle2013/doc/codegen.pdf`. Part of the Isabelle distribution.

[4] Alexander Krauss (2013): *Defining Recursive Functions in Isabelle/HOL.* Theorem Proving Group TUM, Munich. Available at `http://isabelle.in.tum.de/dist/Isabelle2013/doc/functions.pdf`. Part of the Isabelle distribution.

[5] Walther Neuper (2012): *Automated Generation of User Guidance by Combining Computation and Deduction.* Electronic Proceedings in Theoretical Computer Science 79, Open Publishing Association, pp. 82–101, doi:10.4204/EPTCS.79.5.

[6] Walther Neuper (2013): *On the Emergence of TP-based Educational Math Assistants.* [1], pp. 110–129. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".

[7] Jan Ročnik (2013): *Trials with TP-based Programming for Interactive Course Material.* [1], pp. 91–109. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".

[8] G. Rosu, A. Stefanescu, S. Ciobacá & B.M. Moore (2013): *One-Path Reachability Logic*. In: *Logic in Computer Science (LICS)*, pp. 358–367, doi:10.1109/LICS.2013.42.

[9] Franz Winkler (1996): *Polynomial algorithyms in computer algebra*. Springer.