Computer Algebra implemented in Isabelle's Function Package under Lucas-Interpretation — a Case Study

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

1 Introduction

The relation of this paper to "Theorem-Proving (TP) components for educational software" deserves explanation: TP technology is designed for mechanised justification of formalised facts — so educational software gains a prerequisite for being a "transparent system" [12] which explains itself.

Computer Algebra (CA), however, is *not* designed for justification (and thus leaves full responsibility for interpreting results with the user, which over-strains students frequently), and CA is *not* designed for "transparency"[1]. So integration of CA into TP promises to improve justification and "transparency" of CA, provides requisites for step-wise "Lucas-Interpretation" [11], another advantage for interactive educational software.

Integration of CA in TP also serves verification of CA, which is of great importance for extending the scope of Formal Methods, i.e. independent of educational goals. The importance is reflected by several kinds of integration of computational power of CA with deductive power of TP, for instance "the skeptics approach" [5] which takes CA as an "oracle" and verifies results in TP — an approach not feasible in the case study under consideration.

This case study was motivated by a master thesis at RISC Linz, which implemented a CA algorithm for the greatest common divisor of multivariate polynomials [10] in SML. The SML implementation comprises about 90 functions with about 700 lines of code. Subsequent trials with transferring the implementation to Isabelle's function package were surprisingly successful and lead to this report; at the time of writing this text, the first step of transfer was done for univariate polynomials. i.e. for 41 functions of the 90 mentioned above.

2 Programming with Isabelle's Function Package (FP)

The initial hands-on experience with Iabelle/Isar's function package (FP)[7, 6] is pleasant, in particular if following the tutorial [8]. The new prover IDE Isabelle/jEdit has a look-and-feel programmers are familiar with from IDEs like Eclipse.

Translation from SML to the FP is almost one-to-one; auxiliary functions as required for CA are available in Isabelle to the same extent as in SML: *dvd*, *gcd* and others are not present in the latter,

[1]Maple [9] is sometimes said to be "transparent", but following down the traces usually ends at the in-transparent kernel; the same is with WolframAlpha

Submitted to:
THedu'13

but in Isabelle/HOL. The first challenge was to identify, which SML integers represent coefficients of polynomials, and which represent primes. The latter were represented by HOL's *nat* — an instructive exercise in precise programming.

The translation took care to use the simplest Isabelle tool possible: *definition* for non-recursive functions (which well might comprise *fold* or *map*), *primrec* for primitive recursion and *fun* otherwise. The latter resolves all proof obligations automatically and creates a suite of theorems: customized induction rules **.induct* and case rules **.cases* required later for proofs in §4 and §5. The simplification rules **.simps*, generated as well, allow immediate testing of functions by *value* — most important for programmers' approaches.

Only if implementation by *fun* does not succeed immediately, one expands the definition to *function* and resolves the proof obligations explicitly calling *by pat_completeness auto* and postpones the termination proof by *termination sorry*. Table.1 gives a survey on all functions implemented so far:

Isabelle's tool	tool's features	number of occurrences
`definition`	no recursion	21
`primrec`	primitive recursion	1
`fun`	automated proofs	10
`function`	interactive proof	9
total of function definitions		41

Table 1: Kinds of function definitions for the univariate gcd.

Defining functions in the FP is *almost* trivial but *not* in *all* cases of 41 above, which seem typical for CA. The first function which really challenged the programmer was the following one, which decides divisibility (*dvd*) for univariate polynomials (*up*, actually lists of integers, i.e. of exponents). The identifier for the function is declared as *infix* with priority *70* in line 01, the function is implemented by two patterns in line 02 and in line 03:

```
01  function dvd_up :: "unipoly ⇒ unipoly ⇒ bool" (infixl "%|%" 70) where
02     "[d] %|% [p] = (|d|  ≤  |p| ∧ p mod d = 0)"
03  |  "ds %|% ps =
04     (let
05        ds = drop_lc0 ds; ps = drop_lc0 ps;
06        d000 = (List.replicate (List.length ps - List.length ds) 0) @ ds;
07        quot = (lcoeff ps) div2 (lcoeff d000);
08        rest = drop_lc0 (ps minus_up (d000 mult_ups quot))
09     in
10        if rest = [] then True
11           else if quot ≠ 0 ∧ List.length ds ≤ List.length rest then ds %|% rest else False)"
12  by pat_completeness auto
13  termination sorry
```

In this function the FP had problems with *by pat_completeness auto* which required a couple of interventions by the (very supportive) Isabelle developer team. The respective experiences are worth to be communicated, but out of scope of this abstract. The successful outcome looks as follows:

```
01  function (sequential) dvd_up :: "unipoly ⇒ unipoly ⇒ bool" (infixl "%|%" 70) where
```

```
:
    in
      rest = [] ∨ (quot ≠ 0 ∧ List.length ds ≤ List.length rest ∧ ds %|% rest))
  by pat_completeness auto
  termination sorry
```

3 Using Isabelle's Code Generator

For the working programmer automated code generation for relevant target languages is a major motivation. And indeed motivating, following the tutorial [4] and executing

```
export_code "gcd_up" (* gcd for *u*nivariate *p*olynomials *) in SML
module_name GCD_Univariate file "~/codegen/gcd_univariate.ML"
```

immediately led to successfully running SML code: just evaluate the two lines in an auxiliary theory importing *GCD_Poly_FP.thy*, which holds *gcd_up* and the preceding definitions. Inspection of the automatically generated code was the next pleasant surprise: the original SML code (i.e. the model for translation into Isabelle's FP) and the generated code are almost identical.[2].

Further investigations on the efficiency of the generated code are planned, in particular with respect to the indefinite precision integers and naturals in comparison with the original SML code.

4 Proving Termination

Admittedly, generating code from Isabelle definitions with *termination sorry* (as done in §2 and §3 above) is a bad practice in terms of software verification in HOL: On termination, even when stated with *sorry*, i.e. omitting a proof, the theorems mentioned in §2 are created and seduce to inappropriate use. However, as already mentioned, this case study started from a programmers perspective which considers automated generation of code a major success — while the generated code is not less trustworthy than the original SML code.

The obligation to prove termination, as imposed by the FP, marks the transition from naive programming to software engineering. Isabelle's FP supports this transition with much automation: Table.1 shows that from 41 functions only 9 require an explicit termination proof:

Termination proof	number of occurrences
not necessary (`definition, primrec`)	22
done automatically by FP (`fun`)	10
by `lexicographic order`	1
by `relation measure`	2
by `size_change`	0
did *not yet* succeed	6
total of function definitions	41

Table 2: Success of explicit termination proofs.

[2]The final paper will show some examples.

The ration between automated proof and interactive proof of termination seems typical for CA: recursion is not structural with respect to some datatype, rather depends on specific mathematical knowledge. In this case study the crucial function for proving termination is

function next_prime_not_dvd :: "nat ⇒ nat ⇒ nat"

which determines a prime with a certain property greater than a given number. Isabelle2013 lacks respective knowledge, however, the Isabelle development already started to fill this gap[3]. So we look forward to have completed all termination proofs at the end of the case study.

5 Verifying CA Algorithms in Isabelle

This task has not begun at the time of writing this extended abstract. For sake of efficiency in development and of coherence of Isabelle's knowledge the first step of this task will be to adopt Isabelle's multivariate Polynomials[4] — from the side of Isabelle development respective tools are already under construction[5] for supporting this task.

Transferring the *gcd* algorithm completely to Isabelle's FP and verifying it seems essential: there seems no way to produce a certificate for the property of being the *greatest* common divisor. Thus, in order to get a computable *gcd* into the Isabelle distribution, verification of some algorithm seems necessary.

6 Making CA transparent by Lucas-Interpretation

Lucas-Interpretation [11] is a TP-based technology, which maintains both, an environment (for computation) and a logical context (for deduction) and thus allows step-wise execution of algorithms, where a student is free at any step to investigate the underlying knowledge, to suggest a next step (where derivability is proved automatically from the context) or to ask the system for a next step.

While the benefits for education seem evident, the benefits for implementing CA are not. On completion the case study is expected to produce detailed requirements for debugging (with inspection of environment and context) and experiences, how Lucas-Interpretation might support verification of the underlying CA algorithm.

7 Conclusion

The steps already done in this case study suggest, that Isabelle's function package is ready for implementation of comprehensive algorithms in Computer Algebra. Furthermore, the experiences with "programming in Isabelle" are promising such, that integrative support for "Domain Engineering" (as an offspring of Formal Methods) [1, 2, 3] might come into sight within some years.

About advantages of Lucas-Interpretation for the implementation of CA nothing can be said at the time of writing this extended abstract; results are expected on completion of the case study for the final publication in autumn 2013.

[3]See the development branch at `http://isabelle.in.tum.de/reports/Isabelle/rev/7f864f2219a9`.
[4]`http://isabelle.in.tum.de/dist/library/HOL/HOL-Library/Polynomial.html`
[5]See the development branch at `http://isabelle.in.tum.de/reports/Isabelle/rev/3cc46b8cca5e`.

References

[1] Dines Bjørner (2006): *Software Engineering*. *Texts in Theoretical Computer Science* 3, Springer, Berlin, Heidelberg.

[2] Dines Bjørner (2009): *Domain Engineering. Technology Management, Research and Engineering*. *COE Research Monograph Series* 4, JAIST Press, Nomi, Japan.

[3] Dines Bjørner (2013): *Domain Science and Engineering as a Foundation for Computation for Humanity*, chapter 7, pp. 159–177. Francis & Taylor. In: J.Zander & P.J.Mosterman (eds.), Computational Analysis, Synthesis, and Design of Dynamic Systems.

[4] 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.

[5] John Harrison & Laurent Thèry (1998): *A sceptic's approach to combining HOL and Maple*. *J. of Automated Reasoning* 21, pp. 279–294.

[6] Alexander Krauss (2006): *Partial recursive functions in higher-order logic*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning (IJCAR 2006)*, *Lecture Notes in Artificial Intelligence* 4130, Springer Verlag, pp. 589–603, doi:10.1007/11814771_48.

[7] Alexander Krauss (2010): *Partial and Nested Recursive Function Definitions in Higher-order Logic*. *J. Autom. Reasoning* 44(4), pp. 303–336, doi:10.1007/s10817-009-9157-2.

[8] 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.

[9] Douglas Meade (2009): *Getting Started with Maple, 3rd ed.* Wiley.

[10] Diana Meindl (2013): *Implementation of an Algorithm Computing the Greatest Common Divisor for Multivariate Polynomials*. Master's thesis, Reserach Institute for Symbolic Computation (RISC) Linz.

[11] 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.

[12] Walther Neuper (2013): *On the Emergence of TP-based Educational Math Assistants*. 7, pp. 110–129. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".