Polynomials in Isabelle for the Working Mathematician

Wolfgang Schreiner, RISC Linz
Walther Neuper, TU Graz

May 16, 2014

Computer Theorem Proving is becoming indispensable for complicated or for tedious proofs, for instance for the proof of the Four Colour Theorem (Gonthier 2005) or for verification of safety-critical software components. Isabelle belongs to the provers closest to industrial use.

Nevertheless, the dominating majority of mathematicians still are working without provers, because proofs are even more demanding if done mechanically and this would slow down evolution of mathematics, effectively.

The present paper describes the proof-of-concept prototype for a polynomial package in Isabelle. The prototype is being developed in cooperation with the Isabelle team at TU Munich and with an expert at ETH Zurich. The description takes the point of view of a working mathematician, introduces Isabelle's concepts and technologies gently and gives an impression of strengths and limitations at the state-of-the-art in Computer Theorem Proving.

The planned polynomial package addresses an area, where the gap between working mathematicians and Computer Theorem Proving seems rather narrow and evidently is most urgent: Computer Algebra proves properties of algorithms by hand, still, while respective implementations in software are not verified at all.

Contents

1 Introduction **2**

2 An abstract polynomial for concise proofs **5**
2.1 Type definitions and invariants 6
2.2 Polynomial algebra over algebraic structures 8
2.3 Proofs about abstract algorithms 10

3 Specific representations and efficient code **11**
3.1 Distributive and recursive representation for specific algorithms 12
3.2 Automated generation of verified and efficient code 13
3.3 Readable input and output formats 15

4 Details: logical relations between executable representations **16**
4.1 Abstract representations types 16
4.2 Implementation types 17
4.3 Monomial orders 18

5 Conclusions for a working mathematician **18**

1 Introduction

Computer Algebra (CA) is related to computer software by existence, the relation between the academic discipline and the respective software is remarkable: Academic mathematicians make major efforts to *prove* mathematical properties of algorithms. But they do these proofs (almost) exclusively by hand and not by Computer Theorem Provers (TPs). And then, the state-of-the-art CA software products are derived from the proved properties *without verification*, that the proved properties still hold for respective implementations.

Altogether, academic CA together with CA software development are large-scale ventures with great impact on science, technology and engineering; from research and conception to development and optimisation a chain of human experts is involved; with ongoing evolution of CA the chain becomes longer, the links more numerous. This evolution increases the issue to

> *achieve a coherent conceptual framework and a gap-less chain of tools from proving properties of algorithms to generation of efficient code.*

The present introduction starts from the assumption, that recent releases of the theorem prover Isabelle [NPW02] [1] already provide both, a logical base for a framework covering theories and algorithms in CA, as well as core technologies for the tool chain, in particular automated code generation:

[1] http://isabelle.in.tum.de/

Mechanisation of mathematical theories TP started as an esoteric discipline with automated provers using various logical formalisms half a century ago. Today leading TPs like Isabelle are interactive proof assistants, include a variety of automated provers and wrap the respective formalisms uniformly. Since Formal Methods extended their scope to hybrid systems including physical phenomena, TPs extended discrete mathematics for computer science to various theories of mathematics. Isabelle's standard distribution includes theories[2] up to, for instance, multivariate analysis, Taylor series or Kurzweil-Henstock Gauge Integration. For additional theory developments there is an archive of formal proofs[3] .

Isabelle/Isar [Mak07] is a specification and proof language close to traditional mathematical notation and featuring human readable proofs[4]. Isabelle's front-end already incorporates many features expected in an IDE called PIDE[Wen12][5] here. It provides a metaphor of continuous proof checking of a versioned collection of theory sources, with instantaneous feedback in real-time and rich semantic markup for the formal text. In particular, the key-bindings are standard and need no adaption like good old emacs.

Isabelle is a generic framework for developing various kinds of formal logic[6]. For modelling program languages most appropriate is Higher-Order Logic (HOL); thus HOL is used for prototyping polynomials.

Abstract development of algorithms One part of the prototype's focus is investigation, how CA theories are applied in proofs of properties of respective algorithms. For for such proofs an algorithm needs to be described on the same level of abstraction as the mathematical theories. Isabelle provides a function package [Kra06][7] for defining functions in Isabelle/HOL.

The following is an abstract formulation of the Euclidean algorithm in the simplest form (neglecting primitive polynomials etc):

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

This algorithm is particularly abstract with respect to the type constraints in the first line: it works on type `'a::ring_div` which only requires division on rings, i.e. on `int` as well as on `'a mpoly::ring_div`, a univariate polynomial with some type of coefficients. Type polymorphism allows that `0` either denotes the integer or a polynomial; and `mod` must be available for integers as well as for polynomials.

Given the above algorithm, proving the post-condition would require a lemma on polynomials analogous to this one on integers (using a specific prover `metis` and several theorems like `add_commute` for the proof):

```
lemma gcd_add_mult_int: "gcd (m::int) (k * m + n) = gcd m n"
  by (metis gcd_commute_int gcd_red_int mod_mult_self1 add_commute)
```

[2] http://isabelle.in.tum.de/dist/library/HOL/
[3] http://afp.sourceforge.net/
[4] http://isabelle.in.tum.de/overview.html
[5] http://isabelle.in.tum.de/dist/doc/jedit.pdf
[6] See the list of theory libraries at http://isabelle.in.tum.de/documentation.html.
[7] http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/functions.pdf

The above lemma is already contained in the Isabelle distribution, the proof is optimised with respect to shortness and not to readability.

Optimisation by specific data representations CA systems use recursive polynomial representation for calculations in general, for instance for calculating the GCD using Euclid's algorithm.

Specific algorithms require specific polynomial representations for efficiency reasons. For instance, the Gröbnerbases algorithm requires distributive representation; further optimisation of this algorithm involves specific monomial orders. Special polynomial packages, for instance Singular [DGPS12] or SAGE[8] meet these requirements and allow to select appropriate representations.

Isabelle provides a datatype package [BW99][9] for datatypes as usual in functional languages; these datatypes are perfect for modelling polynomials on different levels of abstractions; the package is being re-implemented presently. The prototype under consideration uses the new datatype package already. This is an example (which would have worked with the old package similarly):

```
datatype 'a raw_poly_rec = Coeff_raw 'a | Powers_raw "('a raw_poly_rec) list"
```

The above example shows an initial, "raw" version of a recursive polynomial representation, discussed more closely below.

In case verification is not the only requirement for CA algorithms, but also efficiency, Isabelle needs to meet efficiency requirement as well; these requirements are approached by Isabelle by automated code generation.

Mechanised generation of efficient code Functions defined by the function package as described above are appropriate for proving their properties, but evaluation (if possible at all) is very inefficient. This kind of evaluation within the prover would never achieve acceptable efficiency.

Isabelle's way to provide efficiency in computational speed or in large data-structures is automated code generation [HN10][10], where the proved properties of algorithms transfers to the generated code — this is the second part of the prototype's focus.

Present target languages of Isabelle's code generator are SML, OCaml, Haskell and Scala. The present introduction focuses generation to SML, because this language can easily executed within the Isabelle PIDE. This line in an Isabelle theory produces `SML` code from the function defined above as an example:

```
export_code gcd in SML module_name Gcd
```

The generated module `Gcd` includes all auxiliary functions required by `gcd`. If the type of the function is specified to `"int poly"`, the code for polynomials with integer coefficients is generated; the module then includes a library for arbitrary precision integers.

[8] http://www.sagemath.org/doc/reference/polynomial_rings/index.html
[9] http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/datatypes.pdf
[10] http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/codegen.pdf

The structure of this paper is as follows. §2 describes the prototype of an abstract polynomial, which serves proofs of mathematical properties of algorithms. The description is such, that also Isabelle's respective mechanisms are introduced in a gentle way. §3 shows how different polynomial representations can be used for defining specific algorithms in order to optimise certain kinds of efficiency; it also shows how the representations inherit properties proved for the abstract polynomial; §3.2 demonstrates automated generation of code from different kinds of algorithms, leading to efficient code for specific purposes. Finally §5 gives some conclusions about the present state of the prototype development and a preview to steps planned to achieve full usability for the working mathematician.

For readers interested in hands-on experience we recommend to download Isabelle from http://isabelle.in.tum.de and the theories of the prototype development from http://www.ist.tugraz.at/projects/isac/www/download/poly-proto-140506.tgz. The latter can also be cloned from the Mercurial repository by
`hg clone https://hg.risc.uni-linz.ac.at/wneuper/poly`.

2 An abstract polynomial for concise proofs

In Abstract Algebra polynomials are defined as mapping from the natural numbers to some ring. In the sequel we introduce the prototyped polynomials following a standard book on polynomial algorithms [Win96] which defines on p.17:

Definition 1 *An n-variate polynomial over the ring R is a mapping $p : \mathcal{N}_0^n \to R$, $(i_1, \ldots, i_n) \mapsto p_{1_i}, \ldots, p_{1_n}$ such that $p_{1_i}, \ldots, p_{1_n} = 0$ nearly everywhere.*

The prototype definition resembles this abstract approach. Instead of " $= 0$ *nearly everywhere*", Isabelle's type of `finite` set models the mapping, `'a => 'b::zero` denotes a mapping from some "type" (or "set") `'a` to type `'b::zero`, a type containing at least 0, `~=` denotes "not equal":

```
typedef ('a, 'b) poly_mapping =
  "{f :: 'a => 'b::zero. finite {x. f x ~= 0}}"
```

The above mapping, confined to finitely many images not equal zero, is nested in the definition for polynomial `'a mpoly` with some coefficient type `'a` below: the first `nat` is for a number designating a variable, the second `nat` designates the value of an exponent for this variable; the inner mapping is the first argument of the outer `poly_mapping`, where the second argument `'a::zero` concerns the coefficients of arbitrary type (confined to contain at least zero [11]), `UNIV` is the universal set constrained by type `poly_mapping` (please, regard the first two lines only; the other two lines will be addressed in §2.2):

```
typedef 'a mpoly =
  "UNIV :: ((nat, nat) poly_mapping, 'a::zero) poly_mapping set"
  morphisms mapping_of MPoly
  ..
```

[11] Enhancing the structure of polynomials towards a ring will impose further assumptions on the type of coefficients, as we shall see in §2.2 below.

Conceded, this two-step definition looks different from Def.1; in §3 and §4 we shall discuss further challenges for Isabelle's type system in modelling $\mathcal{N}_0^n$. However, to the prototype developers this definition seems most appropriate for further mechanisation of books like [Win96]. And we shall see, how these technicalities are covered in the sequel by notations, which can immediately be understood by mathematicians.

2.1 Type definitions and invariants

The keyword `typedef` used above abbreviates "type definition" and has formal foundations in HOL: In this logic, the definition of a new type like `'a mpoly` requires a proof, that the new type (seen as a set) is not empty. An empty type would make HOL inconsistent. So, Isabelle proves non-emptiness of type `'a mpoly` in the forth line of the definition above, indicated by `..` . In an interactive Isabelle session the cursor on the the three lines above `..` shows this proof obligation:

```
goal (1 subgoal):
  1. ? x. x : UNIV
Auto solve_direct: The current goal can be solved directly with
  Set.UNIV_witness: ? x. x : UNIV
```

For this proof obligation Isabelle finds a proof by one theorem already proved, by `? x. x : UNIV`, where `?` is ∃ and `:` is ∈. Thus Isabelle allows to shortcut the proof by `..` . "Safe reasoning in HOL is ensured by two very restricted mechanisms for extending the logic: one is the definition of new constants in terms of existing ones (for instance `finite`); the other is the introduction of new types by identifying non-empty subsets in existing types (for instance `'a mpoly`)" [KU11].

Traditional textbooks like [Win96] "forget" textual preliminaries like " $= 0$ *nearly everywhere*" in order to reach the required level of abstraction, for instance polynomial algebra using $+$, $\cdot$, etc. In analogy, TP "covers" logical details like `finite` or `poly_mapping` by abstraction mechanisms like `typedef`. For this purpose Isabelle provides convenient mechanisms supported by much automation. The general situation is as follows:

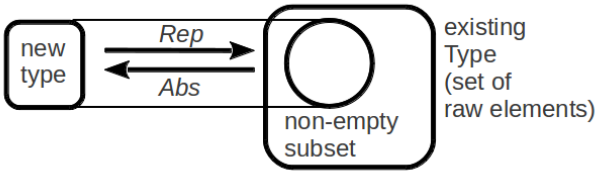

Figure 1: "Lifting" a new type from an existing one.

In Fig.2.1 a *new type*, for instance `'a mpoly`, is abstracted from an existing *raw type*, for instance `((nat, nat) poly_mapping, 'a::zero) poly_mapping`. The raw type must

be proved non-empty as shown above. Between the raw type and the new type two morphisms *Rep* and *Abs* are established; these morphisms can be named specifically, for instance by `mapping_of` and `MPoly` in `'a mpoly`. The respective naming is intuitive: The *Rep*resentation of an `'a mpoly` is created by `mapping_of`, and the *Abs*traction of the *raw type* is an `MPoly` (where the capital letters indicate a constructor according to Isabelle's coding standards).

The respective abstraction mechanism in Isabelle/HOL is called "lifting", a mechanism which can be generalised to so-called "quotient types" [Hom05, KU11]. The morphisms are used to automate plenty of tedious proving by the following setup:

```
setup_lifting (no_code) type_definition_mpoly
```

This kind of setup automatically generates a series of theorems behind the scenes (*MPoly_inverse*, *mapping_of_inverse*, *MPoly_inject*, *mapping_of_inject*, *MPoly_induct*, *mapping_of_induct*, *MPoly_cases*, *mapping_of_cases*, etc.) which in turn provide a high degree of automation.

As an example, let's see how addition is introduced in the current prototype. The two-step definition of `'a mpoly` via `poly_mapping` is the result of attempts with respective proofs; these were confusingly complicated with a one-step definition. Now the introduction of `plus` is the following, first attacking the first step on `poly_mapping`:

```
lift_definition plus_poly_mapping :: "('a::type, 'b::monoid_add) poly_mapping =>
    ('a, 'b) poly_mapping => ('a, 'b) poly_mapping"
  is "%f1 f2 k. f1 k + f2 k"
proof -
  fix f1 f2 :: "'a => 'b"
  assume "finite {k. f1 k ~= 0}"
    and "finite {k. f2 k ~= 0}"
  then have "finite ({k. f1 k ~= 0} union {k. f2 k ~= 0})" by auto
  moreover have "{x. f1 x + f2 x ~= 0} subset {k. f1 k ~= 0} union {k. f2 k ~= 0}"
    by auto
  ultimately show "finite {x. f1 x + f2 x ~= 0}" by (blast intro: finite_subset)
qed
```

This proof is read as follows: Given is a function `plus_poly_mapping` with two arguments of type `poly_mapping` returning the same type. The types of the two arguments of `poly_mapping` are constrained to `('a::type, 'b::monoid_add)`, i.e. the first argument is the most general type and the second is an additive monoid. And the function `is` defined by `%f1 f2 k.  f1 k + f2 k` (where `%` is λ), i.e. two functions applied to the same argument `k` (a set of naturals denoting variables mapped with their respective exponents) and the values `f1 k` and `f2 k` (the respective coefficients) added by `+`. The `+` can be used due to `'b::monoid_add`; the respective mechanism will be introduced in §2.2.

As soon as the proof obligation is clear, the proof itself is self-explanatory: one has to `show` that the the raw type of mappings meets the type constraint, i.e. that the set `{x.  f1 x + f2 x ~= 0}` is `finite`. The proof calls two automated provers, `auto` and

`blast`; in an interactive Isabelle session the proof states can be inspected by clicking the respective parts of the proof.

From `plus_poly_mapping` the addition for `'a mpoly` is a simple definition, where the proof involved can be done by one point at the end:

```
lift_definition plus_mpoly :: "'a mpoly => 'a mpoly => 'a mpoly"
  is "Groups.plus :: ((nat, nat) poly_mapping, 'a) poly_mapping => _" .
```

A general remark on the design of basic definitions: careful design is crucial for elegance of derived definitions and proofs. For instance, the addition as introduced above, had only to care about the interesting case, addition of equal monomials, while all other cases are automatically covered by the datastructure chosen for the basic definition — just set `union` (which is accompanied with lots of lemmas shortening further proofs), as seen in the proof of `lift_definition plus_poly_mapping` above.

And a final remark on principal benefits from `typedef`: this mechanism includes new types into Isabelle/HOL's type inference, it hides details distracting from a high-level view (e.g. `finite x.  f x ∼= 0` distracting from polynomial algebra) and at the same time enforces these details as invariants: One can write concrete mappings, but no `poly_mapping` or `'a mpoly` without proving at the same time, that the invariant holds.

2.2 Polynomial algebra over algebraic structures

A mathematician working with CA systems sees oneself bound to computation: polynomials, for instance, are written in such a system in order to compute sums, products, roots, etc. Now, polynomials defined in Isabelle are not executable per se, i.e. one cannot compute sums of polynomials — this seems evident from the above proof of `lift_definition plus_poly_mapping`.

Rather, `'a mpoly` is designed for concise proofs on a level as abstract as possible. So, the type variable `'a` allows to investigate properties of polynomials with respect to the type of their coefficients. As we have seen in `lift_definition plus_poly_mapping`, these can be constrained to abstract types like `monoid_add`. And what a mathematician wants to have proved first is, that polynomials form a ring.

Isabelle's mechanism for modelling abstract algebra are "axiomatic type classes" [NP95, HW07]. Some of such classes have been mentioned above already, `zero` and `monoid_add`. The implementation of classes relevant for polynomials is found in two theories called *Groups.thy* and *Rings.thy* [12]. This is a small selection of basic classes:

```
class zero =
  fixes zero :: 'a  ("0")

class plus =
```

[12] http://isabelle.in.tum.de/dist/library/HOL/Groups.html and http://isabelle.in.tum.de/dist/library/HOL/Rings.html

```
  fixes plus :: "'a => 'a => 'a"  (infixl "+" 65)

class semigroup_add = plus +
  assumes add_assoc: "(a + b) + c = a + (b + c)"

class monoid_add = zero + semigroup_add +
  assumes add_0_left: "0 + a = a"
    and add_0_right: "a + 0 = a"

class cancel_semigroup_add = semigroup_add +
  assumes add_left_imp_eq: "a + b = a + c ==> b = c"
  assumes add_right_imp_eq: "b + a = c + a ==> b = c"
```

The first two classes `zero` and `plus` introduce new syntax: `zero` can be written as `0` and `plus` can be written as `infix +` associating to the left. The other three classes, `semigroup_add`, `monoid_add` and `cancel_semigroup_add` add logical properties as expected.

The last class `cancel_semigroup_add` might appear unusual. Isabelle/HOL development introduces this class and many others as a means to group the many many lemmas. Already from the axioms of `monoid_add` follows the lemma $0 = x \leftrightarrow x = 0$. From `cancel_semigroup_add` follow $a + b = a + c \leftrightarrow b = c$, $b + a = c + a \leftrightarrow b = c$ and many others, already proved in the Isabelle/HOL distribution.

Now, how can polynomials take profit from algebraic notation with 0, +, etc as well as from the many many lemmas proved for the many classes? This is done by combining `lift_definition`, already introduced above, with `instantiation`. A first example is introduction of the zero polynomial, which is in two steps according to the two-step definition of `'a mpoly` as follows:

```
instantiation poly_mapping :: (type, zero) zero
begin
  lift_definition zero_poly_mapping :: "('a, 'b) poly_mapping" is "%k. 0"
    by simp
  instance ..
end

instantiation mpoly :: (zero) zero
begin
  lift_definition zero_mpoly :: "'a mpoly"
    is "0 :: ((nat, nat) poly_mapping, 'a) poly_mapping" .
  instance ..
end
```

From now on we can state that some polynomial is zero, i.e. `(p::'a::zero mpoly) = 0`. Due to `setup_lifting` the proofs above are done automatically. For plus the respective proof can not be done automatically, we know the proof from above p.7:

```
instantiation poly_mapping :: (type, monoid_add) monoid_add
```

```
begin
  lift_definition plus_poly_mapping :: "('a, 'b) poly_mapping =>
  ('a, 'b) poly_mapping => ('a, 'b) poly_mapping"
    is "%f1 f2 k. f1 k + f2 k"
  proof -
    fix f1 f2 :: "'a => 'b"
    assume "finite {k. f1 k ~= 0}"
      and "finite {k. f2 k ~= 0}"
    then have "finite ({k. f1 k ~= 0} union {k. f2 k ~= 0})" by auto
    moreover have "{x. f1 x + f2 x ~= 0} subset {k. f1 k ~= 0} union {k. f2 k ~= 0}"
      by auto
    ultimately show "finite {x. f1 x + f2 x ~= 0}" by (blast intro: finite_subset)
  qed
  instance
    by intro_classes (transfer, simp add: fun_eq_iff ac_simps)+
end

instantiation mpoly :: (monoid_add) monoid_add
begin
  lift_definition plus_mpoly :: "'a mpoly => 'a mpoly => 'a mpoly"
    is "Groups.plus :: ((nat, nat) poly_mapping, 'a) poly_mapping => _" .
  instance
    by intro_classes (transfer, simp add: fun_eq_iff add.assoc)+
end
```

Further proofs on this way are, as often, done almost automatically (explaining the details below like `(transfer, simp add: fun_eq_iff)+` is out of scope of this introduction):

```
instance poly_mapping :: (type, "{monoid_add, cancel_semigroup_add}") cancel_semigroup_add
  by intro_classes (transfer, simp add: fun_eq_iff)+

instance mpoly :: ("{monoid_add, cancel_semigroup_add}") cancel_semigroup_add
  by intro_classes (transfer, simp add: fun_eq_iff)+
```

Continuing this way leads to a series of mechanised proofs that polynomials form a ring; having proved this, makes available hundreds of lemmas for rings also for polynomial rings. And since Isabelle's rings are associated with traditional algebraic notation one then can write `a = q*b + r` also for polynomials.

2.3 Proofs about abstract algorithms

The previous section introduced Isabelle's mechanisms for implementing the ring of polynomials and for proving respective properties; these mechanisms also introduced the algebraic notation on rings.

On this level of abstraction also algorithms can be defined as shown in the introduction on p.3. Which notions are further required for algorithms on this level of abstraction, this question is not yet closed. One basic notion is "coefficients"; so a function `coeffs` seems

useful, which extracts a respective set (see `primitive` below). The following notions already have been determined as indispensable, although they are closely related to representations: The notion "*main variable*" is related to *recursive representation* and the notion "*leading term*" is related to *distributive representation.* These notions are not yet implemented in the prototype, also the operations of the polynomial ring are not all implemented. But the following notion is easily implemented by use of Isabelle's existing knowledge:

```
definition primitive :: "'a::{ring_div,Gcd} mpoly => bool"
where "primitive p <--> Gcd (coeffs p) = 1"
```

The introduction above in §2.1 and in §2.2 used polynomial addition as an example; so the following simple example for an abstract algorithm uses addition, too:

```
definition double :: "'a::monoid_add mpoly => 'a mpoly"
where "double p = p + p"
```

The code above requires no hint on details of implementation or code generation, there is no distraction from design and development of an algorithm. If a developer wants to experiment with execution of an algorithm, some executable representation of a polynomial must be input, see §3 below. In case a "wrong" representation is chosen, execution might be inefficient but nevertheless operable.

We continue the example with the algorithm `double` and prove a simple theorem for demonstration purposes:

```
lemma double_not_primitive:
  assumes "q = double p"
  shows "neg primitive q"
proof
  assume "primitive q"
  then have *: "Gcd (coeffs q) = 1" by (simp add: primitive_def)
  from assms have "coeffs q = coeffs (double p)" by simp
  then have "coeffs q = coeffs (p + p)" by (simp add: double_def)
  then have "coeffs q = Groups.times 2 ‘ coeffs p" sorry
  then have "Gcd (coeffs q) = 2 * Gcd (coeffs p)" sorry
  then have "Gcd (coeffs q) ~= 1" sorry
  with * show False by simp
qed
```

This proof is not yet finished (which is indicated by `sorry`); but the proof shows, that `typedef 'a mpoly` abstracted away from mappings successfully and traditional notation can be used.

3 Specific representations and efficient code

As mentioned in the introduction on p.4, specific algorithms can enormously be optimised by using specific data representations. Another promise for efficient code is the affinity

to parallelisation within the functional paradigm; Isabelle has been parallelised already, and extension to parallelised algorithms seems straight forward; so far the prototype development under consideration did not yet tackle this issue.

The focus of the present proof-of-concept prototype was the question, how the known polynomial representations, the distributive and the representative, both in dense and sparse variants respectively, the distributive with specific monomial orders, can be modelled such, that

1. mathematical properties proved for abstract algorithms transfer to representations
2. representations can be selected in simple ways for code generation.

Point (1.) should be guaranteed to users of the envisaged polynomial package without further ado; details on how this guarantee is achieved are described in §4 below.

In the sequel we describe the current state of prototyping with respect to point (2.). The state can be called a successful proof-of-concept, but still shows bare bones of Isabelle's mechanisms. How these internals can be hidden by convenient notation will be described in §3.3.

3.1 Distributive and recursive representation for specific algorithms

Given an abstract definition of an algorithm one might want to do trials with execution, without bothering with efficiency. For execution of an algorithms an executable representations of polynomials are required for input.

Below we give four different kinds of representations, distributive dense, distributive sparse, recursive dense and recursive sparse for one and the same polynomial, $2 \cdot x + 3 \cdot z^4$:

```
definition p1 :: "int mpoly" --{* distributive dense *}
where "p1 = monom' (DISTR RLEX) (Nat_Mapping.single' DENSE 0 1) 2 +
            monom' (DISTR RLEX) (Nat_Mapping.single' DENSE 2 4) 3"

definition p2 :: "int mpoly" --{* distributive sparse *}
where "p2 = monom' (DISTR RLEX) (Nat_Mapping.single' SPARSE 0 1) 2 +
            monom' (DISTR RLEX) (Nat_Mapping.single' SPARSE 2 4) 3"

definition p3 :: "int mpoly" --{* recursive dense *}
where "p3 = monom' (REC (%_. DENSE)) (Nat_Mapping.single' DENSE 0 1) 2 +
            monom' (REC (%_. DENSE)) (Nat_Mapping.single' DENSE 2 4) 3"

definition p4 :: "int mpoly" --{* recursive sparse *}
where "p4 = monom' (REC (%_. SPARSE)) (Nat_Mapping.single' DENSE 0 1) 2 +
            monom' (REC (%_. SPARSE)) (Nat_Mapping.single' DENSE 2 4) 3"
```

The above representations still reveal internals of the rapid prototype; of course, these will be hidden from the user in the future, see §3.3 below. These four representations can, without any logical conflict, be input to the simple example algorithm from p.11:

```
value [code] "double p1" --{*result is distributive dense  *}
value [code] "double p2" --{*result is distributive sparse *}
```

```
value [code] "double p3" --{*result is recursive dense     *}
value [code] "double p4" --{*result is recursive sparse    *}
```

The command `value [code]` executes `double` with the respective polynomial representations as arguments; execution within Isabelle sufficiently efficient for rapid prototyping), the results are shown in an output window.

In case the developer decides, some part of an algorithm should work on a specific representation for some reason, this can be done with very little code. For instance, `double` can be determined to work on recursive representation as follows:

```
definition double :: "'a::monoid_add mpoly => 'a mpoly"
where "double p = rec_cast DENSE p + rec_cast DENSE p"
```

The cast function `rec_cast` performs conversion from distributive to recursive representation, in case the input is distributive (in case the input is already recursive, the cast is the identity function). Two constants `DENSE` and `SPARSE` determine respective details of representation.

3.2 Automated generation of verified and efficient code

In the introduction on p.4 has been mentioned, that development of algorithms is done within the logical framework of Isabelle/HOL. Then, for gaining efficiency, from the definitions within the logic automatically code is generated.

Isabelle's code generator [Haf12, HN10] is proved to preserve partial correctness: if evaluation of a term t in the target language (SML in our examples) terminates with value v, then $t = v$ is derivable from the respective definitions in Isabelle/HOL.

Code from an abstract algorithm is generated within one line:

```
export_code double in SML module_name Double
```

The above line exports code for executing the example function `double` to the target language `SML` wrapped into a module `Double` and lists it in the output window (if no `file` is specified). Inspection of the exported code is instructive, because code generation is nicely conservative and preserves readability of the original definitions:

```
fun double (A1_, A2_, A3_) p = plus_mpoly (A1_, A2_, A3_) p p;
```

So in the SML code the + in the original definition of `double` is replaced by `plus_mpoly`, the `lift_definition` given on p.9 above. The additional arguments `(A1_, A2_, A3_)` are best understood when comparing with the code generated from `double_int` shown on p.14 below[13]: `A1_` becomes `one_int`, `A2_` becomes `equal_int`, `A3_` becomes `ring_int` — all the logical properties are forwarded as arguments for appropriate implementation.

Module `Double` is filled with all code for checking logical properties, where type classes as introduced in §2.2 are managed by dictionaries [HN10]. However, without specifying the polynomial representation the code is not executable, so not further interesting here.

[13] Comparison of the modules `Double` and `Double_Int` is particularly interesting if done with a diff-tool.

Specific code for specific coefficient domains Given an algorithm with a signature containing type variables, these can easily be fixed. For instance, the example function `double` can be fixed to polynomials over `int`, `rat` and `complex` respectively:

```
definition double_int :: "int mpoly => int mpoly" --{* integers *}
where "double_int p = p + p"
export_code double_int in SML module_name Double_Int

definition double_rat :: "rat mpoly => rat mpoly" --{* rational numbers *}
where "double_rat p = p + p"
export_code double_rat in SML module_name Double_Rat

definition double_complex :: "complex mpoly => complex mpoly" --{* complex numbers *}
where "double_complex p = p + p"
export_code double_complex in SML module_name Double_Complex
```

The simplicity of the above procedure raises the question: what is easier to handle: (1) an all-embracing compiled system with lots of arguments, switches etc for detailed specification, or (2) a code generator which generates the specified code on the fly.

As already mentioned, generated code is readable such that also logical details can be investigated. For instance, from the above `definition double_int` the following code is generated:

```
fun double_int p = plus_mpoly (one_int, equal_int, ring_int) p p;
```

Above the final function `double_int` in the module `Double_Int` the following definitions can be found within the more than thousand lines:

```
type 'a ring =
  {ab_group_add_ring : 'a ab_group_add,
    semiring_0_cancel_ring : 'a semiring_0_cancel};

val ring_int =
  {ab_group_add_ring = ab_group_add_int,
    semiring_0_cancel_ring = semiring_0_cancel_int}
  : int ring;
```

This way all the logical facts are checked. And executability is provided by Isabelle/HOL's own integers:

```
datatype int = Zero_int | Pos of num | Neg of num;

fun plus_inta (Neg m) (Neg n) = Neg (plus_num m n)
  | plus_inta (Pos m) (Pos n) = Pos (plus_num m n)
  :
```

In recent Isabelle versions there are mechanism to invoke arbitrary precision integers available in some systems (like PolyML).

Algorithms bound to specific representations Some algorithms are particularly efficient on specific polynomial representations; the Gröbnerbases algorithm has been mentioned with respect to distributive representation.

So specific algorithms could be implemented for a specific representation in the following way:

```
definition double_int_distr :: "int poly_distr => int poly_distr"
where "double_int_distr p = p + p"
export_code double_int_distr in SML module_name Double_Int_Distr

definition double_int_rec :: "int poly_rec => int poly_rec"
where "double_int_rec p = p + p"
export_code double_int_rec in SML module_name Double_Int_Rec
```

The reader may note, that `'a poly_distr` and `'a poly_rec` are logically equivalent with the abstract type `'a poly`, see §4.1 below.

3.3 Readable input and output formats

Rapid prototyping during development of algorithms involves experimental execution; for that purpose convenient generation of test-data is required, in our case generation of polynomials. In the prototype executable polynomial representations are implemented as type classes `poly_distr` and `poly_rec`; these are accompanied with invariants (i.e. the set of coefficients unequal to zero is finite, etc). So `lift_definition` is required for functions generating the input polynomials; these enforce the respective invariants. Such functions are not yet implemented in the prototype, they will be named `Poly_Distr_Dense`, `Poly_Rec_Dense`, `Poly_Distr_Sparse` and `Poly_Rec_Sparse`.

Then execution within Isabelle/jEdit will look like as follows, if we provide `double_int` with the recursive representation of $2 \cdot x + 3 \cdot z^4$:

```
let p = Poly_Rec_Dense [:[:[:0, 0, 0, 0, 3:]:], [:[:2::int:]:]:]
in double_int p
```

For `Poly_Distr` also a monomial order must be specified, see §4.3 below. While in CA identifiers for variables usually are omitted, these might be desirable for other uses. So there might be a representation like the following, which is already available in the current Isabelle distribution:

```
lemma
  fixes x y z :: "'a::comm_ring_1"
  shows "2*x + 3*z^4 =
    poly (poly (poly [:[:[:0, 0, 0, 0, 3:]:], [:[:2:]:]:] [:[:x:]:]) [:y:]) z"
  by (simp add: algebra_simps power_numeral_even power_numeral_odd)
```

What has been mentioned here about readability of input likewise holds for output; both shall be considered in later phases of prototype development.

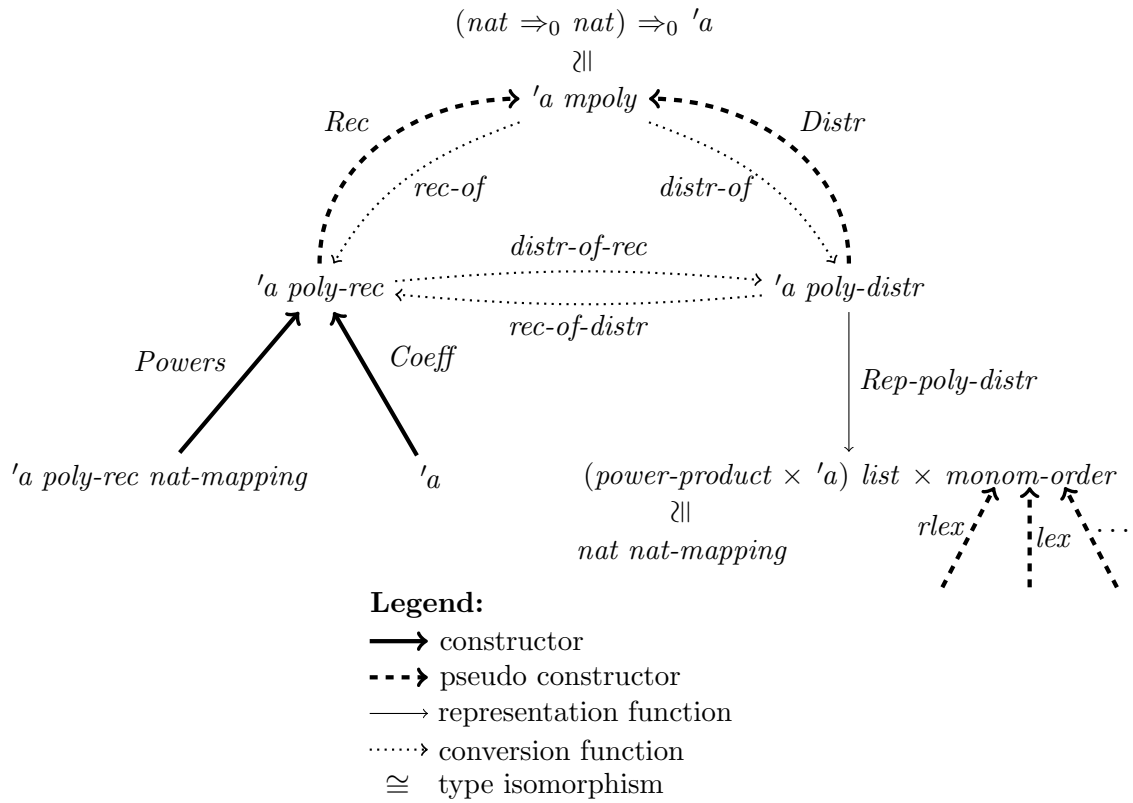

Figure 2: Overview of the two representations of polynomials ©Andreas Lochbihler

4 Details: logical relations between executable representations

.
.
.
GOON .
.
.
xxx

xxxxx

4.1 Abstract representations types

.
.
.
xxx ...
...
...

```
type_synonym monom = "nat nat_mapping"

type_synonym 'a poly_distr_raw = "(monom * 'a) list * monom_order"

typedef 'a poly_distr =
  "{(xs, cmp). c.dsorted_by (compare_vimage fst (monom_compare cmp)) xs /\
    0 ~: snd ` set xs} :: 'a :: zero poly_distr_raw set"

setup_lifting type_definition_poly_distr

lift_definition Distr :: "'a :: zero poly_distr => 'a mpoly"
  is "%p. Poly_Mapping.map_key Abs_nat_mapping
    (let (monom_coeffs, cmp) = Rep_poly_distr p in Abs_poly_mapping (
   %monom. case c.lookup (monom_compare cmp) monom monom_coeffs of None => 0 | Some v => v))" .

code_datatype Distr

...
...
...

datatype 'a raw_poly_rec = Coeff_raw 'a | Powers_raw "('a raw_poly_rec) list"

typedef 'a poly_rec = "{p :: 'a :: zero raw_poly_rec.
  no_trailing_zeros_raw_poly_rec p}"

setup_lifting (no_code) type_definition_poly_rec

function Rec_aux :: "nat => 'a :: {ring,one} poly_rec =>
  ((nat, nat) poly_mapping, 'a) poly_mapping"
where
  "Rec_aux n (Coeff x) = Poly_Mapping.single 0 x"
| "Rec_aux n (Powers ps) = Nat_Mapping.foldr (%e p r. r +
      Rec_aux (Suc n) p *
        Poly_Mapping.single (Poly_Mapping.single n e) 1) ps 0"

lift_definition Rec :: "'a :: {ring, one} poly_rec => 'a mpoly"
is "Rec_aux 0 :: 'a poly_rec => _" .

code_datatype Rec

...
...
```

4.2 Implementation types

.
.

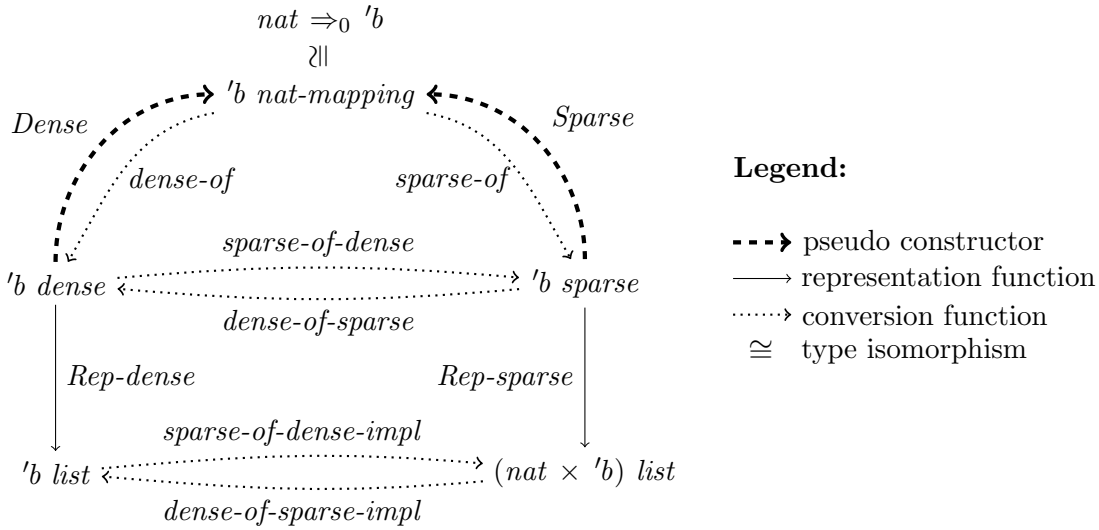

Figure 3: Overview of the two implementation types of representations ©Andreas Lochbihler

.
xxx
...
...

```
typedef 'a nat_mapping = "UNIV :: (nat, 'a) poly_mapping set" ..
```

...
...

4.3 Monomial orders

...
...

5 Conclusions for a working mathematician

This paper is an introduction to the state-of-the-art in Computer Theorem Proving (TP) from the point of view of a mathematicians working in the field of Computer Algebra.

The introduction proceeded alongside the discussion of a proof-of-concept prototype for a polynomial package. This prototype has been begun in a collaborative effort between TU Munich, ETH Zurich and RISC Linz with the question:

Can present Isabelle serve Computer Algebra (CA) in implementing and verifying algorithms such that resulting software is efficient enough to be taken serious in comparison with libraries like SAGE or Singular?

After two months of prototyping and discussing the answer to this question is: No principal obstacles are in sight but still lots of work is expected until positive answers might be given by benchmarks.

Polynomials, the focus of prototyping under consideration, are a central concept in CA and well settled from the side of mathematics. But from the side of TP there lots of interesting questions, which are being discussed in the Isabelle developer community [HLS14]:

- What definitions of polynomial (above called the "abstract" polynomial) are appropriate for concise proofs? Which lemmas are specifically required for proving properties of algorithms?
- How should the relations between abstract polynomial and representations look like in detail?
- How are representations selected for automated code generation?
- Can parallelisation of algorithms compete with mutable data in efficiency? (affinity to parallelisation is an advantage of the presented functional approach with the advent of multi-core systems; traditionally functional programs did not compete with imperative ones, because the latter use mutable data)
- What kinds of modularisation is required for generated code of larger developments?
- How can generated code be integrated into other systems such that reliability is not hampered by technicalities nor by concept?
- ...

And already the small prototyping team encountered lots of questions, which require further interdisciplinary cooperation, for instance:

- Which are the application areas of CA most urgently calling for verified software? And which of them can be served with least development efforts?
- Which monomial orders are specifically required in which algorithms?
- What is more practicable in which cases: compiled code with wide coverage and selection of appropriate functions (as usually done in CA systems) *or* generic code and ad-hoc generation of functions for specific applications (e.g. for a specific type of coefficients)?
- What are the topics, where mechanised polynomials might reach the frontiers of research most quickly?

Finally, RISC Linz and the Isabelle experts involved in the development of the proof-of-concept prototype presented in this paper, appear to be appropriate partners to initialise the challenging bootstrapping process for verified software in Computer Algebra.

References

[BW99] Stefan Berghofer and Markus Wenzel. Inductive datatypes in hol - lessons learned in formal-logic engineering. In *TPHOLs*, pages 19–36, 1999.

[DGPS12] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. SINGULAR 3-1-6 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de, 2012.

[Haf12] Florian Haftmann. *Code generation from Isabelle/HOL theories*. University of Technology, Munich, 2012. Contained in the Isabelle distribution.

[HLS14] Florian Haftmann, Andreas Lochbihler, and Wolfgang Schreiner. Towards abstract and executable multivariate polynomials in Isabelle. Discussion paper at the Isabelle Workshop, 2014.

[HN10] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer Berlin / Heidelberg, 2010.

[Hom05] Peter V. Homeier. A design structure for higher order quotients. In *In Proc. of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs), volume 3603 of LNCS*, pages 130–146, 2005.

[HW07] Florian Haftmann and Makarius Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, number 4502 in LNCS. TYPES 2006, Springer, 2007.

[Kra06] Alexander Krauss. Partial recursive functions in higher-order logic. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 589–603. Springer Verlag, 2006.

[KU11] Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proc. of the 26th ACM Symposium on Applied Computing (SAC'11)*, pages 1639–1644. ACM, 2011.

[Mak07] Wenzel Makarius. Isabelle/Isar — a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, volume 10 of *Studies in Logic, Grammar, and Rhetoric*. University of Bialystok, 2007.

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Wen12] Makarius Wenzel. Isabelle/jEdit — a Prover IDE within the PIDE framework. In J. Jeuring et al, editor, *Conference on Intelligent Computer Mathematics (CICM 2012)*, LNAI. Springer, 2012. to appear.

[Win96] Franz Winkler. *Polynomial algorithms in computer algebra*. Springer, 1996.