Developing Verified Polynomial Code with the Proof Assistant Isabelle

Wolfgang Schreiner[1] Walther Neuper[2]

[1]Wolfgang.Schreiner@risc.jku.at
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University (JKU), Linz, Austria

[2]wneuper@ist.tugraz.at
Institute for Software Technology
Graz University of Technology, Graz, Austria

Core Idea

A single formal (logical/implementation) framework for polynomial arithmetic.

- An abstract type:
  - The ring of multivariate polynomials.
  - $R[x_1, \ldots, x_n]$
- A mathematical representation type:
  - Functions from monomials (exponent sequences) to coefficients.
  - $\mathbb{N}^n \to R$
- Various executable representation types:
  1. Distributive versus recursive structures.
  2. Sparse versus dense (exponent/coefficient) sequences.

Develop theory, prove properties, formulate algorithms, and verify their correctness based on the mathematical representation; have automatically code generated for the executable representations.

Core Idea

abstract type — Map from monomials to coefficients

- Elegantly define basic operations
- Conveniently express algorithms, theorems, and proofs

↓

refinement —

- Theorems are preserved
- Representation values can instantiate variables of abstract type.

Every abstract algorithm is executable with every representation type.

representations	dense	sparse
recursive	✓	✓
distributive	✓	✓

→ generation → executable code

Key Developers

- Florian Haftmann
  Institute for Informatics, TU Munich
- Andreas Lochbihler
  Institute of Information Security, ETH Zurich

F. Haftmann, A. Lochbihler, W. Schreiner: Towards abstract and executable multivariate polynomials in Isabelle, Isabelle Workshop 2014, July 13, 2014, Vienna Summer of Logic, Vienna, Austria.

1. An Illustrative Example

2. Multivariate Polynomials

One-Zero Sequences

A sequence s of $n \geq 1$ ones followed by infinitely many zeros.

$$s = \underbrace{1, 1, 1}_{n=3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots$$

```
(* OneZeroSequences.thy *)
theory OneZeroSequences imports Main begin ... end
```

- The abstract type of such sequences:

```
typedef ozseq =
  -- "the type of one-zero sequences"
  "{s::natseq. ∃n::nat. isonezero s n}"
```

- The mathematical representation type:

```
type_synonym natseq =
  -- "an infinite sequence of natural numbers"
  "nat ⇒ nat"
```

- The subtype constraint:

```
definition isonezero :: "natseq ⇒ nat ⇒ bool" where
  -- "s can be decomposed into a prefix of ones and a suffix of zeros"
  "isonezero s n = (∀i::nat. if i ≤ n then s i = 1 else s i = 0)"
```

Proof Obligations

We have to show that the defined type is not empty.

```
typedef ozseq =
  -- "the type of one-zero sequences"
  "{s::natseq. ∃n::nat. isonezero s n}"

-- goal: ∃x. x ∈ {s. ∃n. isonezero s n}
by (metis mem_Collect_eq onezero)
```

```
definition onezeroseq :: "nat ⇒ natseq" where
  -- "an infinite sequence with (n+1) ones trailed by zeros"
  "onezeroseq n = (λi::nat. if i ≤ n then 1 else 0)"

lemma onezero:
  -- "the two notions above are consistent"
  "∀n::nat. isonezero (onezeroseq n) n"
by (metis isonezero_def onezeroseq_def)
```

Proofs of properties on the abstract level depend on lemmas proved on the representation level.

Some Low-Level Operations

```
setup_lifting type_definition_ozseq
-- "we will lift theorems on number sequences to one-zero sequences"

lift_definition ozseq :: "nat ⇒ ozseq" is
  -- "an one-zero sequence with given bound"
  "λn::nat. onezeroseq n"
by (metis onezero)

lift_definition ozbound :: "ozseq ⇒ nat" is
  -- "the bound of a one-zero sequence"
  "λs::natseq. onezerobound s"
.
```

```
definition onezerobound :: "natseq ⇒ nat" where
  -- "the bound between ones and zeros (if it exists)"
  "onezerobound s = (THE n::nat. isonezero s n)"
```

We lift operations from the representation level to the abstract level; we have to prove that the computed representation satisfies the subtype constraint.

Another Operation

```
lift_definition ozjoin :: "ozseq ⇒ ozseq ⇒ ozseq" is
  -- "the union of two one-zero sequences"
  "λ(s1::natseq) (s2::natseq). maxseq s1 s2"
proof ...
```

```
definition maxseq :: "natseq ⇒ natseq ⇒ natseq" where
  -- "the maximum of two sequences"
  "maxseq s1 s2 = (λi::nat. max (s1 i) (s2 i))"

lemma boundmax:
  -- "the bound of such a sequence"
  "∀(s1::natseq) (s2::natseq) (n1::nat) (n2::nat).
     isonezero s1 n1 ∧ isonezero s2 n2 ⟶
       isonezero (maxseq s1 s2) (max n1 n2)"
proof ...
```

We have to prove that the result of *ozjoin* satisfies the subtype constraint.

Proof of Consistency of the Definition

```
lift_definition ozjoin :: "ozseq ⇒ ozseq ⇒ ozseq" is
  -- "the union of two one-zero sequences"
  "λ(s1::natseq) (s2::natseq). maxseq s1 s2"
-- goal: ⋀fun1 fun2. Ex (isonezero fun1) ⟹ Ex (isonezero fun2) ⟹
--                    Ex (isonezero (maxseq fun1 fun2))
proof -
  fix fun1 fun2
  assume 1: "Ex (isonezero fun1)"
  assume 2: "Ex (isonezero fun2)"
  show "Ex (isonezero (maxseq fun1 fun2))"
  proof -
    from 1 obtain n1 where
    3: "isonezero fun1 n1" by auto
    from 2 obtain n2 where
    4: "isonezero fun2 n2" by auto
    from 3 4 boundmax have "isonezero (maxseq fun1 fun2) (max n1 n2)"
      by metis
    from this show ?thesis by auto
  qed
qed
```

We reduce the proof to a core property of natural number sequences.

A High-Level Algorithm

```
definition ozubound :: "ozseq list ⇒ nat" where
  -- "the bound of the union of a list of one-zero sequences"
  "ozubound S = ozbound (ozunion S)"
```

```
definition ozunion :: "ozseq list ⇒ ozseq" where
  -- "the unions of a list of one-zero sequences"
  "ozunion S = foldr ozjoin S (ozseq 0)"
(*
fun ozunion :: "ozseq list ⇒ ozseq" where
  "ozunion [] = ozseq 0"
| "ozunion (s#r) = ozjoin s (ozunion r)"
*)

lemma ozubound:
  -- "the bound of a union is the maximum of all bounds"
  "∀S::ozseq list. ozbound (ozunion S) = foldr max (map ozbound S) 0"
proof ...
```

Correctness properties of the algorithm can be stated and verified.

Execution of the Algorithm

```
export_code ozunion in SML
-- "the high-level algorithm is (essentially) executable"

structure OneZeroSequences : sig
  type ozseq
  val ozunion : ozseq list -> ozseq
end = struct
...

fun ozjoin xb xc = Abs_ozseq (maxseq (rep_ozseq xb) (rep_ozseq xc));
fun ozunion s = List.foldr ozjoin s (ozseq Arith.Zero_nat);

end; (*struct OneZeroSequences*)

export_code ozunion in SML
```

The core of the algorithm is executable.

Low-Level Operations

```
definition ozubound :: "ozseq list ⇒ nat" where
  -- "the bound of the union of a list of one-zero sequences"
  "ozubound S = ozbound (ozunion S)"

export_code ozubound in SML

Wellsortedness error
(in code equation onezerobound ?s ≡ The (isonezero ?s),
with dependency "Pure.dummy_pattern" -> "ozbound" -> "onezerobound"):
Type nat not of sort enum
No type arity nat :: enum

export_code ozbound in SML

Wellsortedness error ...
```

But not all low-level operations (necessarily) are.

The Executable Representation Type

We can represent every one-zero sequence s just by a natural number:

$$s = \underbrace{1, 1, 1}_{n=3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots \quad \Leftarrow \quad s = OZrep(2)$$

```
definition OZrep :: "nat ⇒ ozseq" where
  -- "the mapping of the executable type into the abstract type"
  "OZrep n = ozseq n"
code_datatype OZrep

-- "the executable versions of the low-level operations"
lemma [code] : "ozseq n = OZrep n"
  by (metis OZrep_def)
lemma [code] : "ozbound (OZrep n) = n"
  by (metis OZrep_def ozseqbound)
lemma [code] : "ozjoin (OZrep n1) (OZrep n2) = OZrep (max n1 n2)"
  by (metis OZrep_def boundmax onezero ozjoin.rep_eq ozseq.abs_eq
        ozseq.rep_eq rep_ozseq_inverse sequnique)
```

We map the executable type into the abstract type and implement the low-level operations on this type; we have to prove that this implementation preserves the original properties.

Execution of the Algorithm

```
export_code ozubound in SML

structure OneZeroSequences : sig
  type ozseq
  val ozubound : ozseq list -> Arith.nat
end = struct

datatype ozseq = OZrep of Arith.nat;

fun ozseq n = OZrep n;
fun ozjoin (OZrep n1) (OZrep n2) = OZrep (Orderings.max Arith.ord_nat n1 n2);
fun ozbound (OZrep n) = n;
fun ozunion s = List.foldr ozjoin s (ozseq Arith.Zero_nat);
fun ozubound s = ozbound (ozunion s);

end; (*struct OneZeroSequences*)
```

We can execute the whole algorithm on objects of the executable type.

Execution of the Algorithm

```
-- "some one-zero sequences in executable representations"
definition p1 :: ozseq where "p1 = ozseq 3"
definition p2 :: ozseq where "p2 = ozseq 2"
definition p3 :: ozseq where "p3 = ozseq 5"
definition ps :: "ozseq list" where "ps = [p1,p2,p3]"

-- "all operations on these objects are executable"
value "ozbound p3"
value "ozunion ps"
value "ozubound ps"

"Suc (Suc (Suc (Suc (Suc 0)))))"
  :: "nat"
"OZrep (Suc (Suc (Suc (Suc (Suc 0))))))"
  :: "ozseq"
"Suc (Suc (Suc (Suc (Suc 0)))))"
  :: "nat"
```

We can execute the low-level operations and the high-level algorithm on objects of the executable type (also within Isabelle).

Types and Algebras

```
class semigroup =
  fixes plus :: "'a => 'a => 'a" (infixl "⊕" 70)
  assumes assoc: "(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)"

definition iplus :: "'a::semigroup ⇒ 'a::semigroup" where
  "iplus x = x ⊕ x"

instantiation ozseq::semigroup
begin
  definition "s1 ⊕ s2 = ozjoin s1 s2"
  instance proof
    fix a::ozseq and b::ozseq and c::ozseq
    from ozassoc show "(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)"
      by (metis plus_ozseq_def)
  qed
end

value "ozbound (p1 ⊕ (p2 ⊕ (iplus p3)))"

"Suc (Suc (Suc (Suc (Suc 0))))" :: "nat"
```

The structure (*ozseq*, *ojoin*) represents a semigroup; all functions applicable to semigroups can thus be applied to *ozseq*.

Algebra Axioms are Satisfied

```
lemma ozassoc:
  "∀ (s1::ozseq) (s2::ozseq) (s3::ozseq).
    ozjoin (ozjoin s1 s2) s3 = ozjoin s1 (ozjoin s2 s3)"
by (metis comp_def map_fun_def ozjoin.rep_eq ozjoin_def seqassoc)
```

```
lemma seqassoc:
  "∀ (s1::natseq) (s2::natseq) (s3::natseq).
   maxseq (maxseq s1 s2) s3 = maxseq s1 (maxseq s2 s3)"
proof (safe)
  fix s1 s2 s3
  show "maxseq (maxseq s1 s2) s3 = maxseq s1 (maxseq s2 s3)"
  proof (unfold maxseq_def)
    show "(λi. max (max (s1 i) (s2 i)) (s3 i)) =
          (λi. max (s1 i) (max (s2 i) (s3 i)))"
      by (metis max.commute max.left_commute)
  qed
qed
```

To prove that *ozjoin* is associative, we first prove this property on the corresponding operation on the representation type.

1. An Illustrative Example

2. Multivariate Polynomials

Multivariate Polynomials

- Definition (Winkler, 1996):

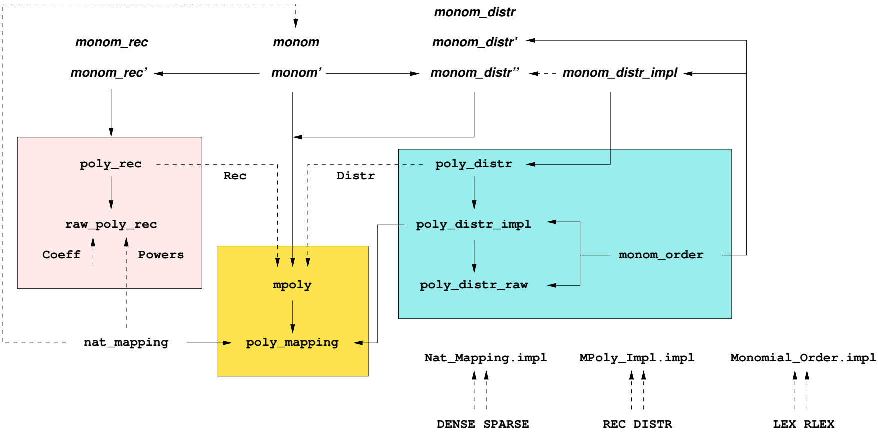
  An n-variate polynomial over the ring R is a mapping $p : \mathbb{N}_0^n \to R, (i_1, \ldots, i_n) \mapsto p_{i_1,\ldots,i_n}$, such that $p_{i_1,\ldots,i_n} = 0$ nearly everywhere. p is written as $\sum p_{i_1,\ldots,i_n} x_1^{i_1} \cdots x_n^{i_n}$ where the formal summation ranges over all tuples $(i_1, \ldots, i_n)$ on which p does not vanish. The set of all n-variate polynomials over R form a ring $R[x_1, \ldots, x_n]$.

- Polynomial addition:

$$\sum_{i \in \mathbb{N}_0^n} p_i \overline{x}^i + \sum_{i \in \mathbb{N}_0^n} q_i \overline{x}^i = \sum_{i \in \mathbb{N}_0^n} (p_i + q_i) \overline{x}^i$$

Elegant formulation of polynomial operations (simple generalization of the univariate case).

The Big Picture



Abstract Type and Mathematical Representation Type

- The constructor of the abstract type:

```
typedef 'a mpoly =
  "UNIV :: ((nat ⇒0 nat) ⇒0 'a::zero) set"
```

```
class zero = (* Groups.thy *)
  fixes zero :: 'a  ("0")
```

- The constructor of the mathematical representation type:

```
typedef ('a, 'b) poly_mapping =
  "{f :: 'a ⇒ 'b::zero. finite {x. f x ≠ 0}}"
type_notation poly_mapping ("(_ ⇒0 /_)" [1, 0] 0)
```

- The finiteness constraint on the constructor:

```
inductive finite :: "'a set => bool" (* Finite_Set.thy *)
  where
    emptyI: "finite {}"
  | insertI: "finite A ==> finite (insert a A)"
```

Both the variable vector and the coefficient mapping are represented by infinite sequences that are almost everywhere zero (the variable number *n* is implicitly determined by the coefficient mapping).

Polynomial Operations on Abstract Type

```
setup_lifting (no_code) type_definition_mpoly

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

instantiation mpoly :: (monoid_add) monoid_add
begin
lift_definition plus_mpoly :: "'a mpoly ⇒ 'a mpoly ⇒ 'a mpoly"
  is "Groups.plus :: ((nat ⇒₀ nat) ⇒₀ 'a) ⇒ _" .
instance by intro_classes (transfer, simp add: fun_eq_iff add.assoc)+
end

instance mpoly :: (comm_monoid_add) comm_monoid_add
  by intro_classes (transfer, simp add: fun_eq_iff ac_simps)+
```

Define monoid $(0, +)$ on the abstract type by lifting the corresponding operations from the mathematical representation type.

Polynomial Operations on Representation Type

```
setup_lifting (no_code) type_definition_poly_mapping

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

instantiation poly_mapping :: (type, monoid_add) monoid_add
begin
lift_definition plus_poly_mapping :: "('a⇒₀'b) ⇒ ('a⇒₀'b) ⇒ ('a⇒₀'b)"
  is "λf1 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} ∪ {k. f2 k ≠ 0})" by auto
  moreover have "{x. f1 x + f2 x ≠ 0} ⊆ {k. f1 k ≠ 0}∪{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

instance poly_mapping :: (type, comm_monoid_add) comm_monoid_add
  by intro_classes (transfer, simp add: fun_eq_iff ac_simps)+
```

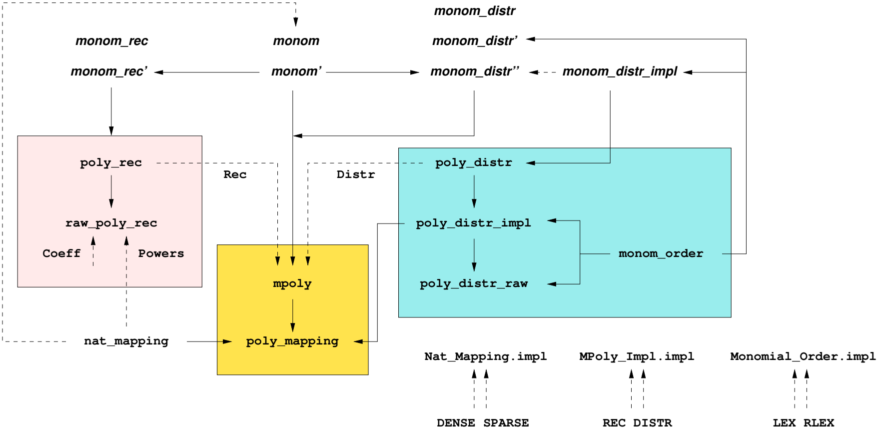
Define monoid $(0, +)$ on the mathematical representation type.

Algorithms on Abstract Type

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

lift_definition coeffs :: "'a::zero mpoly ⇒ 'a set"
is "Poly_Mapping.range :: ((nat ⇒0 nat) ⇒0 'a) ⇒ _" .

definition primitive :: "'a::{ring_div, Gcd} mpoly ⇒ bool"
where "primitive p ⟷ Gcd (coeffs p) = 1"

lemma double_not_primitive:
  fixes p q :: "int mpoly"
  assumes "q = double p"
  shows "¬primitive q"
proof
  assume "primitive q"
  moreover from this have "Gcd (coeffs q) = 1" by auto
  moreover from assms coeffs_plus_same[of "p"] have
    "Gcd (coeffs q) = 2 * Gcd(coeffs p)" by auto
  ultimately show False by auto
qed
```

Define algorithm on the abstract type and verify some property.

The Big Picture



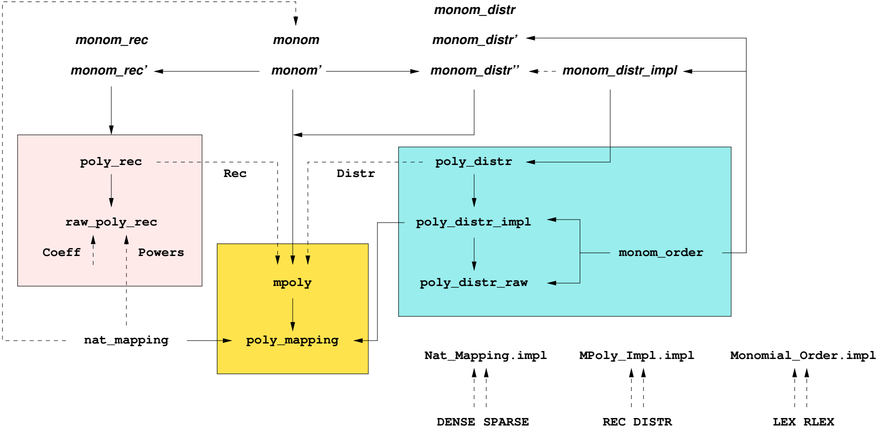
Executable Polynomial Representations

Our goal is to derive two executable representations of multivariate polynomials.

```
typedef 'a poly_rec = ...
typedef 'a poly_distr = ...

lift_definition Rec :: "'a poly_rec ⇒ 'a mpoly"
is ...

lift_definition Distr :: "'a poly_distr ⇒ 'a mpoly"
is ...

code_datatype Distr Rec
```

Two corresponding type constructors.

The Big Picture



Infinite Sequences

```
typedef 'a nat_mapping = "UNIV :: (nat ⇒₀ 'a) set" ..
setup_lifting (no_code) type_definition_nat_mapping
type_notation nat_mapping ("(ℕ ⇒₀ /_)" [0] 0)

lift_definition Dense :: "'a::zero nat_mapping_dense ⇒ (ℕ ⇒₀ 'a)"
  is ...
lift_definition Sparse :: "'a::zero nat_mapping_sparse ⇒ (ℕ ⇒₀ 'a)"
  is ...
code_datatype Dense Sparse
```

```
typedef 'a::zero nat_mapping_dense =
  "{xs :: 'a list. no_trailing_zeros xs}"
  by auto
setup_lifting type_definition_nat_mapping_dense

typedef 'a::zero nat_mapping_sparse =
  "{xs :: (nat × 'a) list.
      sorted (map fst xs) ∧ distinct (map fst xs) ∧ 0 ∉ snd ‘ set xs}"
  by (auto intro: exI [of _ "[]"])
setup_lifting type_definition_nat_mapping_sparse
```

Two executable representation of infinite sequences (that are almost everywhere zero) for coefficients and exponents.

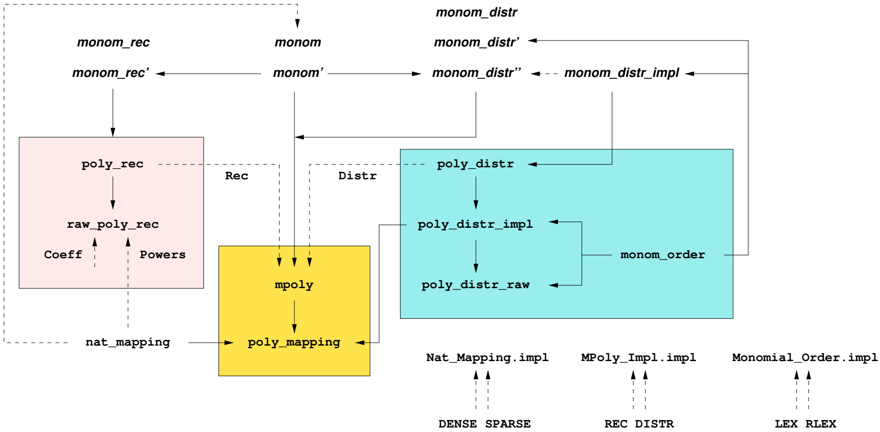
Infinite Sequences

```
value "single' DENSE 3 (1::nat)"
value "single' SPARSE 3 (1::nat)"

"Dense (Abs_nat_mapping_dense [0, 0, 0, Suc 0])" :: "ℕ ⇒₀ nat"
"Sparse (Abs_nat_mapping_sparse [(Suc (Suc (Suc 0)), Suc 0)])" :: "ℕ ⇒₀ nat"
```

```
lift_definition single_dense :: "nat ⇒ 'a::zero ⇒ 'a nat_mapping_dense"
  is ... by ...
lift_definition single_sparse :: "nat ⇒ 'a::zero ⇒ 'a nat_mapping_sparse"
  is ... by ...

datatype impl = IMPL (* Nat_Mapping.thy *)
definition DENSE :: impl where [code del, simp]: "DENSE = IMPL"
definition SPARSE :: impl where [code del, simp]: "SPARSE = IMPL"
code_datatype DENSE SPARSE

definition single' :: "impl ⇒ nat ⇒ 'a::zero ⇒ (ℕ ⇒₀ 'a)"
where [code del, simp]: "single' _ = single" (* a single non-zero value *)

lemma [code]: "single' DENSE k v = Dense (single_dense k v)" ...
lemma [code]: "single' SPARSE k v = Sparse (single_sparse k v)" ...
```

Constructors on these representations.

The Big Picture



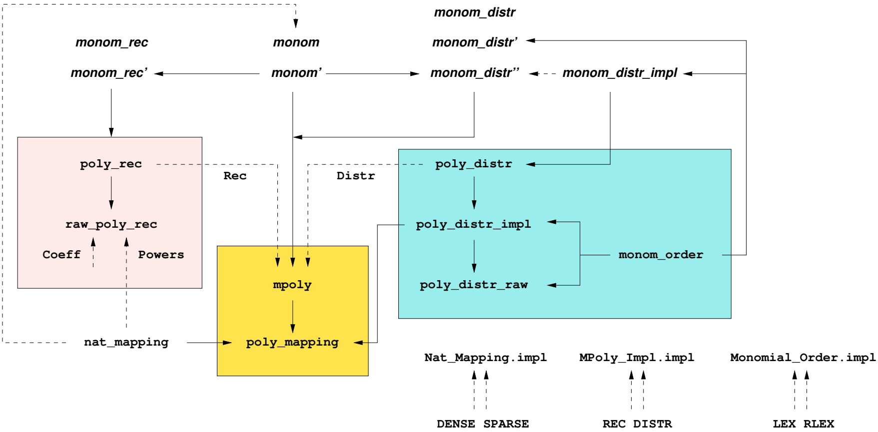
Recursive Polynomials

The recursive representation of a polynomial with n variables is either a coefficient (if $n = 0$) or a list of polynomials with $n - 1$ variables.

```
lift_definition Rec :: "'a :: {ring, one} poly_rec ⇒ 'a mpoly"
is ...
```

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

fun no_trailing_zeros_raw_poly_rec ::
  "'a::zero raw_poly_rec ⇒ bool" where ...

typedef 'a poly_rec =
  "{p:: 'a::zero raw_poly_rec. no_trailing_zeros_raw_poly_rec p}"
  proof show "(Coeff_raw 0) ∈ ?poly_rec" by simp qed
setup_lifting (no_code) type_definition_poly_rec

lift_definition Coeff :: "'a::zero ⇒ 'a poly_rec" is
  ... by ...

definition Powers :: "'a::zero poly_rec nat_mapping ⇒ 'a poly_rec" where
  "Powers ps = ..."

code_datatype Coeff Powers
```

The Big Picture



Distributive Polynomials (Mappings)

The distributive representation of a polynomial is a sequence of pairs of a coefficient and a monomial (exponent sequence).

```
lift_definition Distr :: "'a::zero poly_distr ⇒ 'a mpoly" is ...
```

```
type_synonym monom = "nat nat_mapping"
typedef monom_order = "{compare :: monom comparator. comparator_eq compare}"
  by (auto intro: comparator_eq_on_comparator_of_le)

lift_definition lex_mo :: monom_order is ... by ...
lift_definition rlex_mo :: monom_order is ... by ...
code_datatype rlex_mo lex_mo

type_synonym 'a poly_distr_impl = "(monom ⇒₀ 'a) × monom_order"

typedef 'a poly_distr = "UNIV :: 'a poly_distr_impl set" ..
setup_lifting type_definition_poly_distr
```

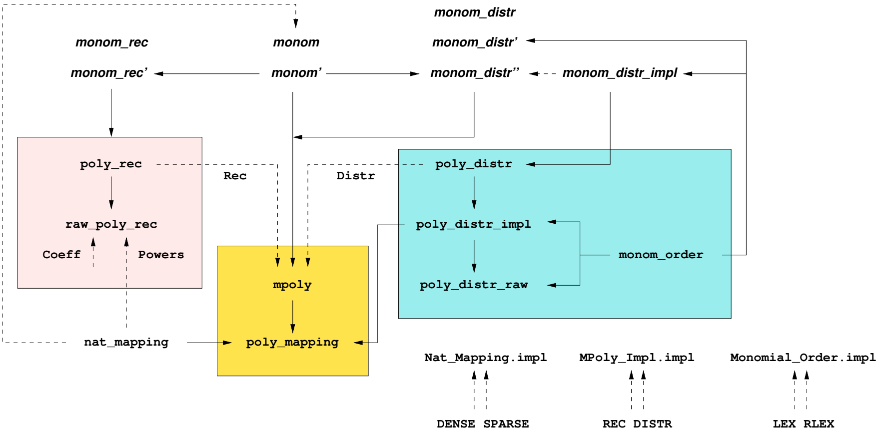
Two (almost) executable representations of multivariate polynomials.

Distributive Polynomials (Lists)

```
type_synonym 'a poly_distr_raw = "(monom × 'a) list × monom_order"

lift_definition poly_distr_of_raw ::
  "'a poly_distr_raw ⇒ 'a::zero poly_distr"
is "(λ(xs, mo). (of_oalist (monom_compare mo) xs, mo))" .

lift_definition raw_of_poly_distr ::
  "'a::zero poly_distr ⇒ 'a poly_distr_raw"
is "(λ(p, mo). (to_oalist (monom_compare mo) p, mo))" .
```

Reduction of infinite monomial sequences to finite monomial lists that are sorted according to the specified monomial order.

The Big Picture



Construction of Monomials

Polynomial cz^d for some coefficient c, variable(s) z and exponent(s) d.

```
lift_definition monom' :: "MPoly_Impl.impl ⇒ monom ⇒ 'a::zero ⇒ 'a mpoly"
is "λ_. monom" .

(* for a recursive polynomial, the (sparse/distributive) representation of
   each variable level (sparse/distributive) may be chosen; for a
   distributive polynomial, the monomial order may be chosen. *)
lemma [code]: "monom' (REC impl) m c = Rec (monom_rec' impl m c)" ...
lemma [code]: "monom' (DISTR mo) m c = monom_distr'' mo m c" ...
```

```
lift_definition monom :: "monom ⇒ 'a::zero ⇒ 'a mpoly"
  is "Poly_Mapping.single :: monom ⇒ _" .

datatype impl = IMPL (* Monom_Order.thy *)
definition RLEX :: impl where [simp, code del]: "RLEX = IMPL"
definition LEX :: impl where [simp, code del]: "LEX = IMPL"
code_datatype RLEX LEX

datatype impl = IMPL (* MPoly_Impl.thy *)
definition REC :: "(nat ⇒ Nat_Mapping.impl) ⇒ impl" where "REC _ = IMPL"
definition DISTR :: "Monom_Order.impl ⇒ impl" where "DISTR _ = IMPL"
code_datatype REC DISTR
```

Basis for the construction of general polynomials.

Construction of Monomials

```
definition monom_rec' ::
  "(nat ⇒ Nat_Mapping.impl) ⇒ nat nat_mapping ⇒ 'a::zero ⇒ 'a poly_rec"
  where [simp, code del]: "monom_rec' _ = ..."
lemma [code]: "monom_rec' impl m c = ..." by ...

definition monom_distr' ::
  "monom_order ⇒ monom ⇒ 'a::zero ⇒ 'a mpoly" where ...
definition monom_distr'' ::
  "Monom_Order.impl ⇒ monom ⇒ 'a::zero ⇒ 'a mpoly" where ...
lift_definition monom_distr_impl ::
  "monom_order ⇒ monom ⇒ 'a::zero ⇒ 'a poly_distr" is ...

lemma [code]: "raw_of_poly_distr (monom_distr_impl mo m c) = ..." by ...

lemma [code]: "monom_distr' mo m c = Distr (monom_distr_impl mo m c)" by ...
lemma [code]: "monom_distr'' = ..." by ...
```

Recursive and distributive implementation of monomials.

Example: Construction of Recursive Monomials

```
value "(Nat_Mapping.single' SPARSE 2 (3::nat))" (* z^3 *)
"Sparse (Abs_nat_mapping_sparse
  [(Suc (Suc (0::nat)), Suc (Suc (Suc (0::nat))))])"
  :: "ℕ ⇒_0 nat"

value "monom' (REC (λ_. SPARSE))
  (Nat_Mapping.single' SPARSE 2 3) (5::int)" (* 5z^3 *)
"Rec (Powers (Sparse (Abs_nat_mapping_sparse
    [(0::nat, Powers (Sparse (Abs_nat_mapping_sparse
        [(0::nat, Powers (Sparse (Abs_nat_mapping_sparse
            [(Suc (Suc (Suc (0::nat))), Coeff (5::int))])))])))])))"
  :: "int mpoly"

value "monom' (REC (λ_. DENSE))
  (Nat_Mapping.single' SPARSE 2 (3::nat)) (5::int)" (* 5z^3 *)
"Rec (Powers (Dense (Abs_nat_mapping_dense
   [Powers (Dense (Abs_nat_mapping_dense
     [Powers (Dense (Abs_nat_mapping_dense
       [Coeff (0::int), Coeff (0::int), Coeff (0::int), Coeff (5::int)]))]))])))"
  :: "int mpoly"
```

$5z^3$ in recursive representation with sparse/dense monomials.

Example: Construction of Distributive Monomials

```
value "monom' (DISTR RLEX)
  (Nat_Mapping.single' SPARSE 2 (3::nat)) (5::int)" (* 5z^3 *)
"Distr (poly_distr_of_raw
    ([(Sparse (Abs_nat_mapping_sparse
        [(Suc (Suc (0::nat)), Suc (Suc (Suc (0::nat))))]), 5::int)],
     rlex_mo))"
  :: "int mpoly"

value "(Nat_Mapping.single' DENSE 2 (3::nat))"
"Dense (Abs_nat_mapping_dense [0::nat, 0::nat, Suc (Suc (Suc (0::nat)))])"
  :: "ℕ ⇒₀ nat"

value "monom' (DISTR RLEX)
  (Nat_Mapping.single' DENSE 2 (3::nat)) (5::int)" (* 5z^3 *)
"Distr (poly_distr_of_raw
    ([(Dense (Abs_nat_mapping_dense
        [0::nat, 0::nat, Suc (Suc (Suc (0::nat)))]), 5::int)],
     rlex_mo))"
  :: "int mpoly"
```

$5z^3$ in distributive representation with sparse/dense monomials.

Adding Recursive Polynomials

```
instantiation poly_rec :: (monoid_add) monoid_add
begin
fun plus_poly_rec :: "'a poly_rec ⇒ 'a poly_rec ⇒ 'a poly_rec"
where ...
instance ...
end

lemma plus_poly_rec [code]:
  "Coeff x    + Coeff y    = Coeff (x + y)"
  "Powers ps + Powers qs = Powers (ps + qs)"
  "Coeff x    + Powers qs =
    Powers (Nat_Mapping.single' (implT qs) 0 (Coeff x) + qs)"
  "Powers ps + Coeff y    =
    Powers (ps + Nat_Mapping.single' (implT ps) 0 (Coeff y))"
...

lemma plus_rec [code]:
  "Rec p + Rec q = Rec (p + q)"
...
```

Recursive construction of a polynomial $p + q$.

Adding Distributive Polynomials

```
instantiation poly_distr :: (monoid_add) plus begin
lift_definition plus_poly_distr ::
  "'a poly_distr ⇒ 'a poly_distr ⇒ 'a poly_distr"
is "..." .
instance ..
end

fun plus_distr_raw
  :: "'a :: {zero, plus} poly_distr_raw ⇒ 'a poly_distr_raw ⇒
     'a poly_distr_raw"
where
  "plus_distr_raw (p1, cmp1) (p2, cmp2) = ..."

lemma plus_poly_distr_code [code]:
  "raw_of_poly_distr (p + q) =
     plus_distr_raw (raw_of_poly_distr p) (raw_of_poly_distr q)"
by ...
```

Distributive construction of a polynomial $p + q$.

Example: Adding Polynomials

```
value "monom' (REC (λ_. SPARSE)) (Nat_Mapping.single' DENSE 0 1) 2 +
       monom' (REC (λ_. SPARSE)) (Nat_Mapping.single' DENSE 2 4) 3"
"Rec (Powers (Sparse (Abs_nat_mapping_sparse
   [(0::nat, Powers (Sparse (Abs_nat_mapping_sparse
     [(0::nat,Powers (Sparse (Abs_nat_mapping_sparse
       [(Suc (Suc (Suc (Suc (0::nat)))), Coeff (3::int))]))))]))),
          (Suc (0::nat), Coeff (2::int))])))"
  :: "int mpoly"

value "monom' (DISTR RLEX) (Nat_Mapping.single' SPARSE 0 1) 2 +
       monom' (DISTR RLEX) (Nat_Mapping.single' SPARSE 2 4) 3"
"Distr (poly_distr_of_raw
         ([(Sparse (Abs_nat_mapping_sparse
             [(Suc (Suc (0::nat)), Suc (Suc (Suc (Suc (0::nat)))))]), 3::int),
           (Sparse (Abs_nat_mapping_sparse [(0::nat, Suc (0::nat))]), 2::int)],
          rlex_mo))"
  :: "int mpoly"
```

$2x + 3z^4$ in recursive and in distributive representation.

Exported Code

```
definition double_int :: "int mpoly ⇒ int mpoly" where "double_int p = p + p"
export_code double_int in SML module_name Double_Int

structure Double_Int : sig ... end = struct ...
fun plus_distr_raw (A1_, A2_, A3_) (p1, cmp1) (p2, cmp2) =
  (if equal_monom_order cmp1 cmp2
    then (oa_zip_with (monom_compare cmp1) (fn _ => SOME) ...
    else let ... val p2a = sort_by (compare_vimage fst cmp) p2;
         in ((oa_zip_with cmp ...);
fun plus_poly_distr (A1_, A2_) p q =
  Poly_distr_of_raw
    (plus_distr_raw
      ((plus_semigroup_add o semigroup_add_monoid_add) A1_, zero_monoid_add A1
      (raw_of_poly_distr (zero_monoid_add A1_) p)
      (raw_of_poly_distr (zero_monoid_add A1_) q));
fun plus_mpoly (A1_, A2_, A3_) (Rec p) (Rec q) =
  Rec (plus_poly_reca
        ((monoid_add_group_add o group_add_ab_group_add o ab_group_add_ring)
           A3_, A2_) p q)
  | plus_mpoly (A1_, A2_, A3_) (Distr p) (Distr q) =
    Distr (plus_poly_distr
            ((monoid_add_group_add o group_add_ab_group_add o ab_group_add_rin
               A3_, A2_) p q);
fun double_int p = plus_mpoly (one_int, equal_int, ring_int) p p;
end; (*struct Double_Int*)
```

Key Results

What do we gain?

- Multiple executable representations.
  - Execution in Isabelle or export of (SML, Haskell, Scala, ...) code.
- Every algorithm works with every representation.
  - But the execution is not necessarily efficient (e.g., the computation of the leading monomial of a polynomial in recursive representation or of a polynomial in distributive representation whose monomial order does not correspond to the requested order).
- Executable code satisfies the properties of abstract operations.
  - Properties of abstract operations have to be proved.
  - Preservation of the semantics of the abstract type by the executable representation has to be proved.

Key Features

Some notable properties of the abstract/executable model.

- The number of variables is not visible in a polynomial type.
  - Polynomials have conceptually infinitely many variables; the executable representation is implicitly extended as needed.
- The representation details are not visible in a polynomial type.
  - Only the representation carries this information; e.g., a polynomial in distributive representation carries a tag that describes the order of its monomials.
- Issue: controlling the representations.
  - Only polynomials with matching representations are combined.
  - The result inherits the representation of its parents.
  - But explicit conversions of representations is possible.

Next Steps

What is needed?

- Parsers and pretty printers.
  - From text to executable representations and vice versa.
  - Convenient input and output of polynomials.
- Formalization of basic arithmetic.
  - $+$ and $*$ are there but not much else yet.
  - Proofs of basic (e.g. ring) properties.
- Implementation of basic arithmetic.
  - In both recursive and in distributive representation.
  - Proof of correspondence to mathematical definitions.
- Higher-level algorithms on top of the basis.
  - Formalization and verification.

Much room for contributions.