A Purely Functional Computer Algebra System Embedded in Haskell

Hiromi ISHII
University of Tsukuba, Japan

19th September 2018
CASC 2018 @ Lille

Motivation

- We apply methods in Functional Programming such as:
  - Dependent Types, Property Based Testing, Purity and Rewriting Rules, ...
- ... to implement a computer algebra system with:
  - Safety, Correctness, and Composability.

Our System

- Embedded Domain Specific Language (EDSL) in Haskell.
    - Haskell: A Statically-typed lazy functional programming language
    - We take advantages of powerful extensions of Glasgow Haskell Compiler (GHC) to design a computer algebra system
- Spoiler: Some methods are applicable also in other languages or paradigms!

Table of Contents

- Type System for Safety and Composability
- Property-based Testing: Lightweight correctness
- Current Status & Examples
- Future Works & Conclusions

Table of Contents

Type System for Safety and Composability

Type-systems for algebra

- Type System: system to decide how values must be typed.
    - Types: Tags, or Invariants on values to enforce safety; "Typed terms never get stuck"
- There are some existing works; e.g:
    - Less-typed: DoCon[1]: Haskell, Java Algebra System[2].
    - Dependently-typed: DoCon-A[3], Coquand et al.[4]
- Our system sits between of the above two
    - We utilise a weak form of "Dependent Types"

Dependent Types?

- Types depending on expressions
  - FULL Dep Types can simulate the higher-order logic, used in proof assistants
- We use WEAK dependent types depending on naturals and list of strings to…
  - distinguish the # of vars and
  - label variables with unique name

"Safety" we want here!

Example: Polynomial arity

- Suppose we have two polyn rings: $R[X_1, \ldots X_n]$ and $R[Y_1, \ldots, Y_m]$, possibly $n \neq m$.
  - Less-typed approach: both are represented by the same type `Poly r`.
  - Compiler should refuse such a confusion of different rings, since it's unclear how vars must be mapped!
- Let's make Poly dependent on n or m!

Arity parametrised polynomials

- Old Poly has "kind" (type of type):

  `Poly :: Type       → Type`

- Our type: **Poly** r n

  `Poly :: Type → ℕ → Type`

- Now depends on Nat, not only Types!
  - Such types are NOT directly available in Java or Plain Haskell.
  - We can still simulate type nats by phantom types, but it adds burden w/o native support.

Throwing errors at Compile-time

```
f1 :: Poly ℕ 1
f1 = 3 * #x ^ 2 + 2 * #x + 1
f2 :: Poly ℚ 1
f2 = 3 * #x ^ 2 + 2 * #x + 1

λ> f1 + f2
Couldn't match type ‘ℚ’ with ‘ℕ’

g1 :: Poly ℚ 2
g1 = let [x,y] = vars in x ^ 2 + y
g2 :: Poly ℚ 3
g2 = let [x,y,z] = vars in z * y + x

λ> g1 + g2
Couldn't match type ‘2’ with ‘3’
```

Different Coeff. (so what?)

Type-level naturals!

Different Arity!

Generic I/F for Polyns

- We also provide a generic interface with type-classes.

  ➡ More

  - Making library more composable
  - polyns optimised for univariate case or homogenisation, …
- We use type-level functions to repr. their arities, monomial orderings and coeffs.

```
class
 (Module (Coeff poly) poly,
  Ring poly, Ring (Coeff poly),
  IsMonomialOrder (MOrder poly))
⇒ IsOrdPoly poly where
  type Arity  poly :: ℕ
  type MOrder poly :: Type
  type Coeff  poly :: Type
  liftMap
    :: (Module (Coe                s,
        Ring alg)
    ⇒ (ℕ<Arity poly →                ly
    → alg
  ...
```

Type-level functions!

➡ Examples

Casting functions

- We cannot add directly polyns with exactly the same setting but with different types, by design.
  - (f :: Unipol ℚ)+(g :: OrdPol ℚ Lex 1) → Error!
  - We provide various casters for explicit casting!

```
convPoly :: (Coeff r ~ Coeff r', MOrder r ~ MOrder r',
             Arity r ~ Arity r')
         ⇒ r → r'

injVars  :: (Arity r ≤ Arity r', Coeff r ~ Coeff r')
         ⇒ r → r'
```

Exactly the same settings

Cast into "more" variables

Labeled Polyns

- We want more flexible control of reordering of vars!
- LabPoly converts **any** polyn type into "**labelled**" one, each variables with the **unique** name (LabPoly' is a synonym).
- canonicalMap does exactly what we expect!

```
data LabPoly poly (vs :: [Symbol])
type                                y (OrdPoly r ord (Len vs)) vs

f :: LabPoly' Rational Grevlex '["x", "y"]
f = 5 * #x ^ 2 * #y ^ 3 - #y + 1

f' :: LabPoly' Rational  Lex '["a", "y", "b", "x", "z"]
f' = canonicalMap f

λ> canonicalMap f :: LabPoly' Rational Grevlex '["y"]
error: Couldn't match type ‘'False’ with ‘'True’
```

Specify variable intuitively

Type-level strings!

Why not full Dependent Types?

- Encoding everything in Dependent Types means proving everything (including termination)
  - We sometimes want to implement algorithms whose termination is remain unknown!
  - We require proofs only for arity arithmetic
    - We developed lemma collection and compiler plugin for Presburger arith to minimize burden.

➡ Example

  - Floating Point Numbers doesn't form a ring; it can't be treated directly in such settings!

Type-system: Summary

- We use weak dependent-types for type-safety:
  - distinguish polynomials with different # of vars
    - Type-naturals are simulatable in other langs.
  - We save "manual proofs" by compiler plugin.
- Automatically induces maps between polyns.
  - Rewriting Rules to reduce overheads, thanks to the Purity (difficult in impure langs). ➡ More
- Omitted: safer quotient rings, without full dep types but with higher polymorphism! ➡ More

Table of Contents

- Type System for Safety and Composability
- Property-based Testing: Lightweight correctness
- Current Status & Examples
- Future Works & Conclusions

Table of Contents

Property-based Testing: Lightweight correctness

Property-based Testing (PBT) [6]

- Tests program against formal spec, feeding some # of randomly generated or enumerated inputs.
- More robust than unit tests w/ fixed inputs.
- Multiple strategies for generating inputs
- Not limited to Haskell; e.g. Hypothesis[7] in Python
- No proof required; we can even verify algorithms whose validity is unknown.

Example

```
prop_division :: ℚ → Property
prop_division q =
  q ≠ 0 ⟹ (recip q * q == 1 && q * recip q == 1)
                  && 1 * q = q

p                 t n =
  forAll (idealOfArity n) $ \ i →
  let gs = calcGroebnerBasis i
  in all (isZero . (`modPoly` gs))
         [sPoly f g | f ← gs, g ← gs, f ≠ g]
```

Every non-zero rational has inverse

1 is mult unit

For all n-variate ideal

S-poly of any two distinct elements reduces to zero

Drawbacks

- Not as rigorous as formal theorem proving as in DoCon-A; trade-off for flexibility.
- Testing may take much time
  - Since G.b. comp has doubly-exponential worst time complexity, tests may explode.
  - We can reduce # of inputs, at the sacrifice of the confidence.
- By its nature, not so good at treating existential props.
  - Invoke external decision proc in such cases if available

➡ More

PBT: Summary

- Checks if formal specs are satisfied by testing against generated inputs
  - More rigorous than fixed-input unit tests, but less than theorem proving
  - Applicable to experimental algorithms
  - Available in many other languages
- Worst complexity and existential properties are bottlenecks for PBT.

Table of Contents

- Type System for Safety and Composability
- Property-based Testing: Lightweight correctness
- Current Status & Examples
- Future Works & Conclusions

Table of Contents

Current Status & Examples

Implemented Algorithms

- Groebner Basis Computation
  - Buchberger (naive, syzygy, sugar)
  - Basis conversion (FGLM, deg-by-deg, Hilbert)
  - Faugère's F_5 and F_4
- Quotients by Zero-dimensional ideals
- Cantor-Zassenhaus factorisation
- Fields: $\mathbb{Q}$, $\mathbb{F}_p$, Galois Fields, $\bar{\mathbb{Q}}$ (naïve)

F_5: Pseudocode in CLO[8]

$\mathbf{G} := \emptyset$

$\mathbf{P} := \{\mathbf{e}_1, \ldots, \mathbf{e}_s\}$

$\mathbf{S} := \{-f_j \mathbf{e}_i + f_i \mathbf{e}_j \mid 1 \leq i < j \leq s\}$

WHILE $\mathbf{P} \neq \emptyset$ DO

    $\mathbf{g}$:= the element of smallest signature in $\mathbf{P}$

    $\mathbf{P} := \mathbf{P} \setminus \{\mathbf{g}\}$

    IF Criterion$(\mathbf{g}, \mathbf{G} \cup \mathbf{S}) =$ false THEN

        $\mathbf{h}$:= a regular $\mathfrak{s}$-reduction of $\mathbf{g}$ by $\mathbf{G}$

        IF $\phi(\mathbf{h}) = 0$ THEN

            $\mathbf{S} := \mathbf{S} \cup \{\mathbf{h}\}$

        ELSE

            $\mathbf{h} := \dfrac{1}{\mathrm{LC}(\phi(\mathbf{h}))}\mathbf{h}$

            $\mathbf{P} := \mathbf{P} \cup \{S(\mathbf{k}, \mathbf{h}) \mid \mathbf{k} \in \mathbf{G}$ and $S(\mathbf{k}, \mathbf{h})$ is regular$\}$

            $\mathbf{G} := \mathbf{G} \cup \{\mathbf{h}\}$

RETURN $\phi(\mathbf{G})$

Example: F_5 impl

```
f5 :: (Field (Coeff p), IsOrdPoly pol)⇒ Vector p → [(Vector p, p)]
f5 i = runST $ do
  let n = length i
  gs ← newSTRef []
  ps ← newSTRef $ fromList [ basis n k | k ← [0..n-1]]
  syzs ← newSTRef [ sVec i_m i_n | m ← [0..n-1], n ← [0..j-1] ]
  whileJust_ (H.viewMin <$> readSTRef ps) $ \ (Entry sig g, ps') → do
    ps .= ps'
    (gs', ss') ← (,) <$> readSTRef gs <*> readSTRef syzs
    unless (standardCriterion sig ss') $ do
      let (h, ph) = reduceSignature i g gs'
          h' = map (* injCoeff (recip $ leadCoeff ph)) h
      if isZero ph then syzs .%= (mkEntry h : )
        else do
        let adds = fromList $ mapMaybe (regSVec (ph, h')) gs'
        ps .%= H.union adds
        gs .%= ((monoize ph, mkEntry h') :)
  map (\ (p, Entry _ a) → (a, p)) <$> readSTRef gs
```

Other Impls

- Generic Matrix I/F for F_4 ➡ More
  - Pluggable Gaussian Elimination; users can use custom matrices.
- Using laziness and parallelism combinators in Hilbert-driven alg. ➡ More
  - Power series as infinite list, computing convolutions parallelly

Benchmarks (ms)

Fastest

2nd

Faster than Singular's gb!

		I_1 (lex)	I_1 (grevlex)	I_2 (lex)	I_2 (grevlex)	I_3 (grevlex)
	…uch	1.861	13.59	14.28	4.204	800.3
	…ub	104.4	160.2	25.64	16.76	7785
	F_5	0.5623	3.869	2.992	1.389	7.173
Singular	gb	2.5550	1.0554	2.5037	0.8904	0.9090
	sba	0.2717	0.3768	0.2403	0.2592	0.4221

Intel Xeon E5-2690 at 2.90 GHz, RAM 128GB, Linux 3.16.0-4 (SMP), using 10 cores

$$I_1 := \langle 35y^4 - 30xy^2 - 210y^2z + 3x^2 + 30xz - 105z^2 + 140yt - 21u,$$
$$5xy^3 - 140y^3z - 3x^2y + 45xyz - 420yz^2 + 210y^2t - 25xt + 70zt + 126yu\rangle$$
$$I_2 := \langle w + x + y + z, wx + xy + yz + zw, wxy + xyz + yzw + zwx, wxyz - 1\rangle$$
$$I_3 := \langle x^{31} - x^6 - x - y, x^8 - z, x^{10} - t\rangle$$

- Our F_4 impl took much execution time and not included
- Slower than state-of-the-art impl in most cases

More on F_5

Fastest

2nd

3rd

	Signature Order	I_1 (lex)	I_1 (grevlex)	I_2 (lex)	I_2 (grevlex)	I_3 (grevlex)
Our F_5	POT	0.4138	4.262	2.837	1.286	17.62
	TOP	0.5977	4.288	5.728	3.461	6.781
	t-POT	0.5623	3.869	2.992	1.389	7.173
	t-TOP	0.4860	3.879	3.100	1.360	7.319
	d-POT	2.986	3.764	3.297	1.342	7.040
	d-TOP	3.631	4.138	5.178	3.521	6.709
Singular	gb	2.5550	1.0554	2.5037	0.8904	0.9090
	sba	0.2717	0.3768	0.2403	0.2592	0.4221

➡ Different Env

- Some heuristics can help? (TOP for high degree.... etc.)

Table of Contents

- Type System for Safety and Composability
- Property-based Testing: Lightweight correctness
- Current Status & Examples
- Future Works & Conclusions

Table of Contents

Future Works & Conclusions

Future Works

- More performance tuning needed for F_4
  - Hensel lifting or Chinese Remaindering for Matrices...
  - Parallelism and GPU
    - Haskell's Purity empowers parallelism
    - There are several parallel matrices [9, 10]
  - More Aggressive Heuristics and Rewriting?
- Mixture of theorem-proving, automated proving and property-based testing

Conclusions

- With weak dependent-types and higher polymorphism, we can achieve more type-safety, retaining flexibility as an experimental env.
- Type-class and type naturals enables us to make generic and composable interface
  - Rewriting Rules can reduce the overhead
- Property-based testing enables us to verify the impl. in a lightweight manner.
- Some methods are applicable in other paradigms!

References

1. Mechveliani, S. D.: *Computer algebra with Haskell: applying functional–categorial–"lazy" programming*. In: Proceedings of International Workshop CAAP, pp. 203–211. (2001)
2. Kredel, H., Jolly, R.: *Generic, type-safe and object oriented computer algebra software*. In: Computer Algebra in Scientific Computing, pp. 162–177. Springer, Berlin, Heidelberg (2010)
3. Mechveliani, S. D.: *DoCon-A a Provable Algebraic Domain Constructor*. http: //www.botik.ru/pub/local/Mechveliani/docon-A/2.02/ manual.pdf (2018). Accessed 06/05/2018
4. Coquand, T., Persson, H.: *Gröbner bases in type theory*. In: Types for Proofs and Programs, pp. 33–46. Springer, Berlin, Heidelberg (1999)

References

5. Kiselyov, O., Shan, C.: *Functional Pearl: implicit configurations -- or, type class reflect the values of types*. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, pp.33-44. ACM (2004).
6. Claessen, K., Hughes, J.: *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. ICFP '00, pp. 268–279. ACM (2000).
7. Hypothesis: Most testing is ineffective - Hypothesis. https://hypothesis.works (2018).
8. Cox, D.A., Little, J., O'Shea, D.: *Ideals, Varieties and Algorithms*. Springer (2015).

References

9. Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: *Regular, shape-polymorphic, parallel arrays in Haskell*. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP '10, pp. 261–272. ACM, Baltimore, Maryland, USA (2010)

10. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: *Accelerating Haskell array codes with multicore GPUs*. In Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP '11). ACM (2011).

Thank You!

Any Questions?

- With weak dependent-types and higher polymorphism, we can achieve more type-safety, retaining flexibility as an experimental env.
- Type-class and type naturals enables us to make generic and composable interface
  - Rewriting Rules can reduce the overhead
- Property-based testing enables us to verify the impl. in a lightweight manner.
- Some methods are applicable in other paradigms!

Appendix

Algebraic Hierarchy as Type-classes

- Expressing algebraic hierarchy intuitively
    - Not so new idea
- NO proofs of algebraic laws required
    - Property-based testing can fix this

```
class Additive a where
  (+) :: a → a → a

class Additive a
    ⇒ Monoidal a where
  zero :: a

...

class (Multiplicative a,
       Monoidal a)
    ⇒ Ring a where
  fromInteger :: ℕ → a
```

Examples of Polynomial Rings

```haskell
data Unipol r = Unipol [r]
instance CoeffRing r ⇒ IsOrdPoly (Unipol r) where
  type Arity  (Unipol r) = 1
  type Coeff  (Unipol r) = r
  type MOrder (Unipol r) = Lex
  ...

data OrdPoly r ord n = Unipol (Map (OMonom ord n) r)
instance (IsMonomialOrder ord, CoeffRing r)
      ⇒ IsOrdPoly (OrdPoly r ord n) where
  type Arity  (Unipol r ord n) = n
  type Coeff  (Unipol r ord n) = r
  type MOrder (Unipol r ord n) = ord
  ...
```

Univariate poly,
impl. as a coeff list

Multivar poly,
as a fin map from
monomials

Example: Easy arity proofs

- Example: mapping vars to the end!
- We want: `injVarsEnd :: (Arity r ≤ Arity r') ⇒ r → r'` with $X_1, \ldots, X_n \mapsto X_{m-n+1}, \ldots, X_m$.
- We use: `injVarsOffset:: (k + Arity r ≤ Arity r') ⇒ Sing k → r → r'` with $X_1, \ldots, X_n \mapsto X_{k+1}, \ldots, X_{k+n}$.

Singleton: type-level argument

- Solution? `injVarsOffset (sing :: Sing (m - n))`
  - GHC cannot see $m - n + n \leq m$!

Convincing GHC with Proofs

- We developed the type-natural package with many proofs on natural numbers
  - Answer:

```
withRefl (minusPlus m n Witness) $
  injVarsOffset (sing :: Sing (m - n))
```

:: (m - n) + n = m

- We also devised a type-checker plugin to automatically proof props within Presburger arithmetic.
- With helps from these, much less effort is needed to convince GHC.

Purity and Rewriting Rules: Reducing Overhead

- Casting functions are implemented genericaly; imposes extra overhead if the mapping is trivial
  - We can use Rewriting Rules to reduce overhead!
  - LHS will be replace by RHS if the type matches.
- Since every expr in Haskell is pure (w/o side-effect), we can concentrate on algebraic validity!

convPoly b/w same types must be identity

```
{-# RULES
 "convPoly/id" convPoly = id
 "injVars/monotone"
   injVars (Poly dic) = Poly (mapKeysMonotone padZero dic) #-}
```

More efficient var shifting within specific impl.

Quotient Ring Example (omitted in the paper)

- We can still achieve type-safe quotient rings!
  - Enabled by "Implicit Configurations" [5] tech, which utilises Rank-N Polymorphism.

```haskell
data Quot r i
modIdeal  :: Reifies i (Ideal r) ⇒ r → Quot r i
withQuot  :: Ideal poly
          → (∀i. Reifies i (Ideal poly) ⇒ Quot poly i)
          → poly

instance (Reifies i (Ideal r), IsOrdPoly r) ⇒ Ring (Quot r i)
```

Quotient of ring r by an ideal i

This ∀ prevents ideal from leaking

We cannot add Quot's with different i's.

Reading: "i carries info of an ideal / R"

Existential Properties

- Naive spec for gb(I) ⊆ I:

  $\forall \vec{f}\ \forall g \in \mathbf{gb}(\langle f_1, \ldots f_n \rangle)\ \exists (c_1, \ldots, c_\mathrm{n})$ s.t. $f = c_1 f_1 + \ldots + c_\mathrm{n} f_n$

  - There is no guarantee that the tester can find c's by its generation strategy, resulting in false-negatives!

- Solution: resort to existing, external reliable decision procedure
  - Our package calls SINGULAR in the spec.

Matrix I/FM for F_4

```haskell
class MMatrix mat a where
 scaleRow :: Mult a ⇒ Int → a → mat s a → ST s ()
  ...

class MMatrix (Mutable mat) a ⇒ Matrix mat a where
  type Mutable mat :: Type → Type
  freeze :: Mutable mat s a → ST s (mat a)
  ...
  gaussReduction :: Field a ⇒ mat a → mat a

type Strategy f w = f → f → w
f4 :: (Ord w, ..., Matrix mat (Coeff p))
   ⇒ proxy mat → Strategy p w → Ideal p → [p]
```

- F4 algorithm reduces g.b. to Gauss elimination
- We provide im/mutable matrices classes to make algorithm composable
- We also abstract selection strategies as weighting function

Hilbert Driven

```haskell
data HPS n = HPS { taylor :: [ℕ], numerator :: Unipol ℕ }

instance Eq (HPS a) where
  (==) = (==) `on` numerator
instance Additive (HPS n) where
  HPS cs f + HPS ds g = HPS (zipWith (+) cs ds) (f + g)
instance LeftModule (Unipol Integer) (HPS n) where
  f .* HPS cs g = HPS (conv (taylor f ++ repeat 0) cs) (f * g)

conv :: [ℕ] → [ℕ] → [ℕ]
conv (x : xs) (y : ys) =
  let parSum a b c = a `par` b `par` c `seq` (a + b + c) in
  x * y :
   zipWith3 parSum (map (x*) ys) (map (y*) xs) (0 : conv xs ys)
```

- Power series as inf list, and numerator for equality
- We use parallelism combinators in convolution!

Benchmark in Other Env

	Signature Order	I_1 (lex)	I_1 (grevlex)	I_2 (lex)	I_2 (grevlex)	I_3 (grevlex)
Our F5	POT	0.3543	2.391	1.889	0.9322	11.14
	TOP	0.3315	2.641	4.342	2.774	5.328
	t-POT	0.7545	3.175	2.091	1.071	5.611
	t-TOP	0.6171	3.458	2.148	1.026	8.779
	d-POT	3.523	2.304	1.694	0.9248	4.740
	d-TOP	3.164	2.822	4.204	2.697	5.519
Singular	gb	3.1955	1.3231	2.9396	1.0662	1.2709
	sba	0.4670	0.4502	0.3742	0.3758	0.4801

Fastest

2nd

3rd

- NB: Benchmarked on my Laptop (2.8 GHz Intel Core i7, 16GB RAM; different than other results)