A Purely Functional Computer Algebra System Embedded in Haskell

Hiromi Ishii(✉)

University of Tsukuba, Tsukuba, Ibaraki 305-8571, Japan
h-ishii@math.tsukuba.ac.jp

Abstract. We demonstrate how methods in *Functional Programming* can be used to implement a computer algebra system. As a proof-of-concept, we present the `computational-algebra` package. It is a computer algebra system implemented as an embedded domain-specific language in *Haskell*, a purely functional programming language. Utilising methods in functional programming and prominent features of Haskell, this library achieves safety, composability, and correctness at the same time. To demonstrate the advantages of our approach, we have implemented advanced Gröbner basis algorithms, such as Faugère's F_4 and F_5, in a composable way.

Keywords: Gröbner basis · Signature-based algorithms
Computational algebra · Functional programming · Haskell
Type system · Formal methods · Property-based testing
Implementation report

1 Introduction

In the last few decades, the area of computational algebra has grown larger. Many algorithms have been proposed, and there have emerged plenty of computer algebra systems. Such systems must achieve *correctness*, *composability* and *safety* so that one can implement and examine new algorithms within them. More specifically, we want to achieve the following goals:

Composability means that users can easily implement algorithms or mathematical objects so that they work seamlessly with existing features.

Safety prevents users and implementors from writing "wrong" code. For example, elements in different rings, e.g. $\mathbb{Q}[x, y, z]$ and $\mathbb{Q}[w, x, y]$, should be treated differently and must not directly be added. Also, it is convenient to have handy ways to convert, inject, or coerce such values.

Correctness of algorithms, with respect to prescribed formal specifications, should be guaranteed with a high assurance.

We apply methods in the area of *functional programming* to achieve these goals. As a proof-of-concept, we present the `computational-algebra` package [12]. It is implemented as an embedded domain-specific language in the

Table 1. Symbols in code fragments

Symbol	Code	Symbol	Code	Symbol	Code	Symbol	Code
$\underline{\mathbb{N}}$	`Nat`	$\underline{\mathbb{Z}}$	`Integer`	$\underline{\mathbb{Q}}$	`Rational`	$\underline{\mathbb{F}_p}$	`F p`
$::$	`::`	$=$	`==`	$\neq$	`/=`	$\lambda\ \vec{\mathtt{x}} \rightarrow \mathtt{e}$	$\backslash\vec{\mathtt{x}}$ `-> e`
$\times$	`*`	$\ltimes$	`!*`	$\frown$	`++`	$\ominus$	`%-`
$\simeq$	`:~:`	$\sim$	`~`	$\rightarrow$	`->`	$\leftarrow$	`<-`
$\Longrightarrow$	`==>`	$\Rightarrow$	`=>`	$:=$	`.=`	$:\Leftarrow$	`.%=`
$\subseteq$	`` `Subset` ``	$\leq$	`<=`	$\circ$	`.`	$\wedge$	`.&&.`
$\bullet$	`.*`	$\langle\$\rangle$	`<$>`	$\langle*\rangle$	`<*>`	$\forall$	`forall`

Haskell Language [10]. More precisely, we adopt the *Glasgow Haskell Compiler* (GHC) [7] as our hosting language. We use GHC because: its *type-system* allows us to build a safe and composable interface for computer algebra; *lazy evaluation* enables us to treat infinite objects intuitively; *declarative style* sometimes reduces a burden of writing mathematical programs; *purity* permits a wide range of equational optimisation; and there are plenty of libraries for functional methods, especially *property-based testing*. These methods are not widely adopted in this area; an exception is *DoCon* [23], a pioneering work combining Haskell and computer algebra. Our system is designed with more emphasis on safety and correctness than DoCon, adding more ingredients. Although we use a functional language, some methods in this paper are applicable in imperative languages.

This paper is organised as follows. In Sect. 2, we discuss how the progressive type-system of GHC enables us to build a safe and expressive type-system for a computer algebra. Then, in Sect. 3, we see how the method of *property-based testing* can be applied to verify the correctness of algebraic programs in a lightweight and top-down manner. To demonstrate the practical advantages of Haskell, Sect. 4 gives a brief description of the current implementations of the Hilbert-driven, F_4 and F_5 algorithms. We also take a simple benchmark there. We summarise the paper and discuss related and future works in Sect. 5.

In what follows, we use symbols in Table 1 in code fragments for readability.

2 Type System for Safety and Composability

In this section, we will see how the progressive type-level functionalities of GHC can be exploited to construct a safe, composable and flexible type-system for a computer algebra system. There are several existing works on type-systems for computer algebra, such as in Java and Scala [15,18], and DoCon. However, none of them achieves the same level of safety and composability as our approach, which utilises the power of *dependent types* and *type-level functions*.

2.1 Type Classes to Encode Algebraic Hierarchy

We use *type-classes*, an ad-hoc polymorphism mechanism in Haskell, to encode an algebraic hierarchy. This idea is not particularly new (for example, see

Mechveliani [23] or Jolly [15]), and we build our system on top of the existing `algebra` package [17], which provides a fine-grained abstract algebraic hierarchy.

Code 1. Group structure, coded in the `algebra` package

```
1 class Additive a where
2   (+) :: a → a → a
3 class Additive a ⇒ Monoidal a where
4   zero :: a
5 class Monoidal a ⇒ Group a where
6   negate :: a → a
```

Code 1 illustrates a simplified version of the algebraic hierarchy up to `Group` provided by the `algebra` package. Each statement between `class` or ⇒ and `where`, such as `Additive a` or `Monoidal a`, expresses the constraint for types. For example, Lines 1 and 2 express "a type `a` is `Additive` if it is endowed with a binary operation +", and Lines 3 and 4 that "a type `a` is `Monoidal` if it is `Additive` and has a distinguished element called `zero`".

Note that none of these requires the "proof" of algebraic axioms. Hence, one can accidentally write a non-associative `Additive`-instance, or non-distributive `Ring`-instance[1]. This sounds rather "unsafe", and we will see how this could be addressed reasonably in Sect. 3.

2.2 Classes for Polynomials and Dependent Types

Expressing algebraic hierarchy using type-class hierarchy, or class inheritance, is not so new and they are already implemented in DoCon or JAS. However, these systems lack a functionality to distinguish the arity of polynomials or the denominator of a quotient ring. In particular, DoCon uses sample arguments to indicate such parameters, and they cannot be checked at compile-time. To overcome these restrictions, we use *Dependent Types*.

For example, Code 2 presents the simplified definition of the class `IsOrdPoly` for polynomials. We provide an abstract class for polynomials, not just an implementation, to enable users to choose appropriate internal representations fitting their use-cases.

The class definition includes not only functions, but also *associated types*, or *type-level functions*: `Arity`, `MOrder` and `Coeff`. Respectively, they correspond to the number of variables, the monomial ordering and the coefficient ring.

Note that `liftMap` corresponds to the universality of the polynomial ring $R[X_1, \ldots, X_n]$; i.e. the free associative commutative R-algebra over $\{1, \ldots, n\}$.

[1] Indeed, one can use *dependent types*, described in the next subsection, to require such proofs. However, this is too heavy for the small outcome, and does not currently work for primitive types.

Code 2. A type-class for polynomials

```
class (Module (Coeff poly) poly, Commutative poly, Ring poly,
       CoeffRing (Coeff poly), IsMonomialOrder (MOrder poly))
   ⇒ IsOrdPoly poly where
  type Arity  poly :: ℕ
  type MOrder poly :: Type
  type Coeff  poly :: Type
  liftMap :: (Module (Scalar (Coeff poly)) alg, Ring alg)
         ⇒ (ℕ<Arity poly → alg) → poly → alg
  leadTerm :: poly → (Coeff poly, OrdMonom (MOrder poly) n)
  ...
```

Code 3. Examples for polynomial instances

```
1   instance (IsMonomialOrder ord, CoeffRing r)
2         ⇒ IsOrdPoly (OrdPoly r ord n) where
3     type Arity  (OrdPoly r ord n) = n
4     type MOrder (OrdPoly r ord n) = ord
5     type Coeff  (OrdPoly r ord n) = r
6     ...
7
8   f :: OrdPoly ℚ Grevlex 3
9   f = let [x,y,z] = vars in x ^ 2 × y + 3 × x + z + 1
10
11  instance (CoeffRing r) ⇒ IsOrdPoly (Unipol r) where
12    type Arity  (OrdPoly r ord n) = 1
13    type MOrder (OrdPoly r ord n) = Lex
14    type Coeff  (OrdPoly r ord n) = r
15    ...
```

In theory, this function suffices to characterise the polynomial ring. However, for the sake of efficiency, we also include some other operations in the definition.

Code 3 shows example instance definitions for the standard multivariate and univariate polynomial ring types. Note that, in Lines 8 and 12, number literal *expressions* 1 and 3 occur in *type* contexts. Types depending on expressions are called *Dependent Types* in type theory. GHC supports them via the *Promoted Data-types* language extension [27] since version 7.4. Our library heavily uses this functionality, and achieves the type-safety preventing users from unintendedly confusing elements from different rings.

2.3 Proofs in Dependent Types and Type-Driven Casting Function

In theory, we can use `liftMap` to cast between any elements of "compatible" polynomial rings. To reduce the burden to write boilerplate casting functions, our library comes with smart functions, as shown in Code 4. The `convPoly` function maps a polynomial into one with the same setting but different representation; e.g. `OrdPoly` ℚ `Lex` `1` into `Unipol` ℚ. The next `injVars` function maps an

Code 4. Various casting function, with simplified type-signatures

```
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'
injVarsOffset :: (n + Arity r ≤ Arity r', Coeff r ∼ Coeff r')
                ⇒ Sing n → r → r'
```

element of $R[X_1, \ldots, X_n]$ into another polynomial ring with the same coefficient ring, but with more number of variables, e.g. $R[X_1, \ldots, X_{n+m}]$, regardless of ordering. For example, it maps Unipol $\mathbb{Q}$ into OrdPoly $\mathbb{Q}$ `Grevelx 3`. Then, `injVarsOffset` is a variant of `injVars` which maps variables with offset; for example,

```
injVarsOffset [sn|3|] :: Unipol ℚ → Polynomial ℚ 5
```

maps $\mathbb{Q}[X]$ into $\mathbb{Q}[X_0, \ldots, X_4]$ with $X \mapsto X_3$. Here, `[sn|3|]` is called a *singleton* for the type-level natural number `3`, first introduced by Eisenberg et al. [4]. More precisely, for any *type-level natural* `n`, there is the unique *expression* `sing :: Sing n` and we can use it as a tag for type-level arguments.

To work with type-level naturals, we sometimes have to *prove* some constraints. For example, suppose we want to write a variant of `injVars` mapping variables to *the end of* those of the target polynomial ring, instead of *the beginning*. We might first write it as follows:

```
injVarsAtEnd :: (Arity r ≤ Arity r', Coeff r ∼ Coeff r')
               ⇒ r → r'
injVarsAtEnd =
  let sn = sing :: Sing (Arity r)
      sm = sing :: Sing (Arity r')
  in injVarsOffset (sm ⊖ sn) -- Errors!
```

However, GHC cannot see `Arity r' - Arity r + Arity r ≤ Arity r'`. Although this constraint is rather clear to us, we have to give the compiler its proof. We have developed the `type-natural` package [14] which includes typical "lemmas". For example, we can use the `minusPlus` lemma to fix this:

```
-- From type-natural:
minusPlus :: Sing n → Sing m
          → IsTrue (m ≤ n) → ((n - m) + m) ≃ n

injVarsAtEnd :: (Arity r ≤ Arity r', Coeff r ∼ Coeff r')
               ⇒ r → r'
injVarsAtEnd =
  let sn = sing :: Sing (Arity r)
      sm = sing :: Sing (Arity r')
  in withRefl (minusPlus sm sn Witness) $
     injVarsOffset (sm ⊖ sn)
```

Since giving such a proof each time is rather tedious, we can use type-checker plugins to let the compiler try to prove constraints automatically. In particular, the author developed the `ghc-typelits-presburger` plugin [13] to resolve propositions in Presburger arithmetic at compile time.

Our library also provides the `LabPoly` type, which converts existing polynomial types into "*labelled*" ones. For example, one can write as follows:

```
f :: LabPoly (Polynomial ℚ 3) '["x", "y", "z"]
f = 5 × #x ^ 2 × #y ^ 3 - #y × #z + 1
```

This relies on the `DataKinds` and `OverloadedLabels` language extensions of GHC. GHC's type system is strong enough to reject illegal terms and types, such as `#w :: LabPoly (Unipol ℚ)'["a"]` (w is not listed as a variable) or `LabPoly (Polynomial ℚ 3) '["x", "y", "x"]` (the variable x occurs twice). Using the type-level information, one can invoke the canonical inclusion maps naturally as follows:

```
f :: LabPoly' ℚ Grevlex '["x", "y", "z"]
f = #x × #y × #z + 2 × #y - 3  × #z × #x + 1
g :: LabPoly' ℚ Lex '["w", "z", "y", "u", "x"]
g = canonicalMap f

-- Where:
canonicalMap :: (xs ⊆ ys, Wraps xs poly, Wraps ys poly',
                  IsPolynomial poly, IsPolynomial poly',
                  Coeff poly ∼ Coeff poly')
             ⇒ LabPoly poly xs → LabPoly poly' ys
```

2.4 Optimising Casting Functions with Rewriting Rules

Since the casting functions are implemented generically, they sometimes introduce unnecessary overhead. For example, if one uses `injVars` with the *same* source and target types, it should just be the identity function. Fortunately, we can use the type-safe *Rewriting Rule* functionality of GHC to achieve this:

```
{-# RULES "injVars/identity" injVars = id #-}
```

Each rewriting rule fires at compile-time, if there is a term matching the left-hand side of the rule and having the same type as the right-hand side.

In Haskell, it suffices just to consider algebraic laws to write down custom rewriting rules. This is due to the *purity* of Haskell. That is, every expression in Haskell is pure, in a sense that they evaluate to the same result when given the same arguments. Note that this does not mean that Haskell cannot treat values with side-effects; indeed, the type-system of Haskell distinguishes pure and impure values at type-level, and one can treat impure operations without violating purity as a whole. The trick behind this situation is to describe side-effects as some kind of abstract instructions, instead of treating impure values directly. Hence, for example, duplicating the same term does not make any difference in its meaning, provided that it is algebraically correct. Such a rewriting rule is used extensively in Haskell. For example, Stream Fusion [3] uses them to eliminate unnecessary intermediate expressions and fuse complicated functions into efficient one-path constructions. Yet, DoCon did not do any optimisation using rewriting rules.

In our library, we also use rewriting rules to remove idempotent applications such as "grading" a monomial ordering twice, e.g:

```
{-# RULES "graded/graded" ∀ ord.
  graded (graded ord) = graded ord #-}
```

2.5 Notes on Applicability in Imperative Languages

The safety we achieved in this section cannot be achieved at compile-time without dependent types and type-level functions. Existing works using type-classes or class inheritance to encode algebraic hierarchy, such as JAS or DoCon, lack this level of safety. In theory, one can achieve the same level of safety even in a statically-typed *imperative* language, if it supports a kind of dependent types. For example, in C++, templates with non-type arguments can be used to simulate dependent types. On the other hand, in Java, Generics do not allow non-type arguments and we need to mimic Peano numerals with classes. In either case, it requires much effort to prove the properties of naturals within them, because they lack dedicated support for type-level naturals or type-checker plugins.

On the other hand, to make use of rewriting rules, we need purity as discussed above.

3 Lightweight Correctness: Property-Based Testing

3.1 Property-Based Testing Introduced

In this section, we will address the correctness issue, in a top-down, or *lightweight* manner. Especially, we apply the method of *property-based testing* [1] to verify the correctness of our implementation. The idea is that one specifies the formal properties that the implemented algorithms and types must satisfy, and checks if they hold by testing them against randomly or exhaustively generated inputs. Although it is not as rigorous as a theorem proving, it still gives a guarantee of the correctness at high assurance, after repeating tests time after time.

Code 5 presents the example specifications for algebraic programs. In Lines 1 through 4, `prop_division` states that the implementation of $\mathbb{Q}$ must satisfy the axioms of division ring. The `prop_passesSTest` function demand the result of `calcGroebnerBasis` to pass the S-test. The tester accepts the specifications above, generates a specified number of inputs (default: 100) and tests against them. If all the inputs satisfy the specifications, it successfully halts; otherwise, it reports counterexamples, which is useful while debugging.

Code 5. Formal Specification of Algebraic Programs

```
prop_division :: ℚ → Property
prop_division q =
    q ≠ 0 ⟹ (recip q × q = 1 ∧ q × recip q = 1)
  ∧ q × 1 = q ∧ 1 × q = q

prop_passesSTest n =
  forAll (idealOfArity n) $ λ ideal →
  let gs = calcGroebnerBasis (toIdeal ideal)
  in all (isZero ∘ (‘modPoly‘ gs))
         [sPoly f g | f ← gs, g ← gs, f ≠ g]
```

3.2 Discussion

There are several libraries for property-based testing adopting different strategies to generate inputs. For example, QuickCheck [1] generates inputs randomly, while SmallCheck [26] exhaustively enumerates inputs in the depth-increasing order. Even though there are other implementations of property-based testers in languages other than Haskell [11], it does not seem that it is applied in existing systems, such as Singular [9], JAS or DoCon.

By its *generative* nature, property-based testing has several drawbacks and pitfalls. First, evidently, it cannot assure the validity as rigorously as the *formal theorem proving*, unless the input space is finite. There are several pieces of research that combine formal theorem proving and computational algebra to rigorously certify correctness of implementations (for example, [2,24]). These first formalise the theory of Gröbner basis in the constructive type-theory. Then, they execute them within the host theorem proving language, or extract the program into other languages. However, by its nature, this approach requires everything to be proven formally. It is not so easy a task to prove the correctness of every part of a program, even with help from automatic provers. Even if one manages to finish the proof of the validity of some algorithm, when one wants to optimise it afterwards, then one must prove the “equivalence” or validity of that optimisation. Moreover, it is sometimes the case that the validity, or even termination, of the algorithm remains unknown when it is implemented; e.g. the correctness and termination of Faugerè’s F_5 [6] are proven very recently [25]. Furthermore, there is an obvious restriction that we can extract programs only into the languages supported by the theorem prover. We consider these conditions too restrictive, and decided to adopt theorem proving only in trivial arity arithmetic.

Secondly, if the algorithm has a bad time complexity, property-based tests can easily explode. Specifically, since Gröbner bases have double-exponential worst time complexity, randomly generated input can take much time to be processed. One might reduce the burden by combining randomised and enumerative generation strategies carefully, but there is still a possibility that there are small inputs which take much time. To avoid such a circumstance, one can reduce the number of inputs, however it also reduces the assurance of validity.

Finally, they are not so good at treating *existential properties*. Although SmallCheck provides the existential quantifier in its vocabulary, it just tries to find solutions up to a prescribed depth. If solutions are relatively "larger" than their inputs, this results in *false-negative* failures. For example, one can write the following specification that demands each element of the result of `calcGroebnerBasis` to be a member of the original ideal, however it does not work as expected:

```
prop_gbInc ideal =
  let j = calcGroebnerBasis ideal
  in exists $ λ cs →
     and (zipWith (λ f gs → f == dot ideal gs) j cs)
```

In the above, `dot i g` denotes the "dot-product". As a workaround, we currently combine inter-process communication with property-based testing. More specifically, we invoke a reliable existing implementation, such as SINGULAR, inside the spec as follows:

```
prop_gbInc = forAll arbitrary $ λ i → monadicIO $ do
  let gs = calcGroebnerBasis i
  is ← evalSingularIdealWith [] [] $
        funE "reduce" [
          idealE gs, funE "groebner" [idealE i]]
  return $ all isZero is
```

Thus, if the existential property in question is decidable and has an existing reliable implementation, then it might be better to call it inside specifications.

4 Case Study: Implementing the Hilbert-Driven, F_4 and F_5 Algorithms for Calculating Gröbner Bases

In this section, we will focus on three algorithms as case-studies: the Hilbert-driven, F_4 and F_5 algorithms. Firstly, we demonstrate the power of laziness and parallelism by the Hilbert-driven algorithm. Then by the F_4 interface, we illustrate the practical example of composability. Finally, we skim through the simplified version of the main routine of F_5 and see how imperative programming with mutable states can be written purely in Haskell. For our purpose, we will discuss only a fragment of implementations that elucidates the advantages of Haskell, rather than the entire implementation and theoretical details.

4.1 Homogenisation and Hilbert-Driven Basis Conversion

Homogenisation is a powerful tool in Gröbner basis computation. If $I \subseteq k[\mathbf{X}]$ is a non-homogeneous ideal and $\bar{I} \subseteq k[x, \mathbf{X}]$ its homogenisation, then one can get a Gröbner basis for I by unhomogenising the Gröbner basis $\bar{G}$ for $\bar{I}$ w.r.t. a suitably induced monomial ordering. In this way, any Gröbner basis algorithm for homogeneous ideals can be converted into one for non-homogeneous ones.

Code 6. Basic API for homogenisation

```
1  data Homogenised poly
2  instance IsOrdPoly poly ⇒ IsOrdPoly (Homogenised poly) where
3    type Arity  (Homogenised poly) = 1 + Arity poly
4    type MOrder (Homogenised poly) = HomogOrder (MOrder poly)
5    type Coeff  (Homogenised poly) = Coeff poly
6    ...
7  homogenise   :: IsOrdPoly poly ⇒ poly → Homogenised poly
8  unhomogenise :: IsOrdPoly poly ⇒ Homogenised poly → poly
9
10 calcGBViaHomog :: (Field (Coeff poly), IsOrdPoly poly)
11                ⇒ (∀ r. (Field (Coeff r), IsOrdPoly r)
12                        ⇒ Ideal r → [r])
13                → Ideal poly → [poly]
14 calcGBViaHomog calc i
15   | all isHomogeneous i = calc i
16   | otherwise = map unhomogenise ( calc (fmap homogenise i))
```

Code 6 is an API for these operations. The type `Homogenised poly` represents polynomials obtained by homogenising polynomials of type `poly`. Then `calcGBViaHomog calc i` first checks if the input `i` is homogeneous. If it is so, then it applies the argument `calc` to its input directly (Line 15); otherwise, it first homogenises the input, applies `calc`, and then unhomogenises it to get the final result (Line 16). Note that, though it uses the same term `calc` in both cases, they have different types. In the first case, since it just feeds an input directly, `calc` has type `Ideal poly → [poly]`. On the other hand, in the non-homogeneous case, it is applied *after* homogenisation, hence it is of type `Ideal (Homogenised poly) → [Homogenised poly]`. Thus, `calcGBViaHomog` takes a *polymorphic function* as its first argument and this is why we have ∀ inside the type of the first argument. Such a nested polymorphic type is called a *rank n polymorphic type*, and it is supported by GHC's `RankNTypes` language extension[2].

For example, one can use the so-called *Hilbert-driven algorithm* as the first argument to `calcGBViaHomog`. It first computes a Gröbner basis w.r.t. a lighter monomial ordering, compute the Hilbert–Poincaré series (HPS) with it and use it to compute Gröbner basis w.r.t. the heavier ordering. In this procedure, we need the following operations on HPS: Equality test on HPS's, n^{th} Taylor coefficient of the given HPS, and the $\mathbb{Z}[X]$-module operation on HPS. Code 7 illustrates such an interface for HPS. For equality test, we use the numerator `hpsNumerator` of the closed form, and an *infinite list* `taylor` maintains Taylor coefficients. By the *lazy* nature of Haskell, we can intuitively treat infinite lists and write a convolution on them. In Line 12, `par` and `seq` specify the *evaluation strategy*. In brief, expressions `x` and `y` in "`x ‘par‘ y`" (resp. `seq`) are evaluated *parallelly*

[2] This can be achieved in object-oriented language with subtyping and Generics.

Code 7. Data-type of and operations on Hilbert–Poincaré series

```
data HPS n = HPS { taylor :: [ℤ], hpsNumerator :: Unipol ℤ }

instance Eq (HPS a) where
  (==) = (==) `on` hpsNumerator
instance Additive (HPS n) where
  HPS cs f + HPS ds g = HPS (zipWith (+) cs ds) (f + g)
instance LeftModule (Unipol ℤ) (HPS n) where
  f • HPS cs g = HPS (conv (taylor f ⌒ repeat 0) cs) (f × g)

conv :: [ℤ] → [ℤ] → [ℤ]
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)
```

(resp. *sequentially*). Since every expression is pure in Haskell, we can safely take advantage of parallelism, without a possibility of changing results.

4.2 A Composable Implementation of F_4

F_4 is one of the most efficient algorithms for Gröbner basis computation and was introduced by Faugère [5]. Briefly, F_4 reduces more than two polynomials at once, replacing S-polynomial remaindering in the Buchberger Algorithm with the *Gaussian elimination* of the matrices. This means that the efficiency of F_4 reduces to that of Gaussian elimination and the internal representation of matrices. Thus, it is useful if we can easily switch internal representations and elimination algorithms. For this purpose, we provide type-classes for mutable and immutable matrices which admit row operations and a dedicated Gaussian elimination. Code 8 demonstrates the interface for immutable and mutable matrices (`Matrix` and `MMatrix`) and the type signature of our F_4 implementation (`f4`). In Lines 1 and 6, the last type argument `a` of `Matrix` and `MMatrix` corresponds to the type of coefficients. Note that one can give different instance definitions for the same `mat` but different coefficient types `a`. For example, one can implement efficient Gaussian elimination on $\mathbb{F}_p$ for `Matrix Mat` $\mathbb{F}_p$, and then use it in the definition of `Matrix Mat` $\mathbb{Q}$, with the Hensel lifting or Chinese remaindering.

In Line 15, the first argument of `f4` of type `proxy mat` specifies the internal representation `mat` of matrices. In addition, `f4` takes a *selection strategy* as the second argument. Here, the selection strategy is abstracted as a weighting function to some ordered types, and we store intermediate polynomials in a heap and select all the polynomials with the minimum weight at each iteration.

4.3 The F_5 Algorithm

Finally, we present the simplified version of the main routine of Faugère's F_5 [6] (Code 9). Readers may be surprised that the code looks much imperative. This is

Code 8. Matrix classes and the F_4 function

```
class MMatrix mat a where
  fromRows :: [Vector a] → ST s (mat s a)
  scaleRow :: Multiplicative a ⇒ Int → a → mat s a → ST s ()
  ...

class MMatrix (Mutable mat) a ⇒ Matrix mat a where
  type Mutable mat :: ⋆ → ⋆
  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, IsOrdPoly poly, Field (Coeff poly),
        Matrix mat (Coeff poly))
     ⇒ proxy mat → Strategy poly w → Ideal poly → [poly]
```

made possible by the *ST monad* [19], which encapsulates side-effects introduced by mutable states and prevents them from leaking outside. We use a functional heap to choose the polynomial vectors with the least signature, demonstrating the fusion of functional and imperative styles.

4.4 Benchmarks

We also take a simple benchmark and the result is shown in Table 2 (examples are taken from Giovini et al. [8]). This compares the algorithms implemented in our `computational-algebra` package and Singular. The first four rows correspond to the alrorithms implemented in our library; i.e. the Buchberger algorithm optimised with syzygy and sugar strategy (B), the degree-by-degree algorithm for homogeneous ideals (DbyD), the Hilbert-driven algorithm (Hilb), and F_5. S(gr) and S(sba) stand for the `groebner` and `sba` functions in the Singular computer algebra system 4.0.3. The complete source-code is available on GitHub [12][3]. The benchmark program is compiled with GHC 8.2.2 with flags `-O2 -threaded -rtsopts -with-rtsopts=-N`, and ran on an Intel Xeon E5-2690 at 2.90 GHz, RAM 128GB, Linux 3.16.0-4 (SMP), using 10 cores in parallel. We used the Gauge framework to report the run-time of our library, and the `rtimer` primitive for Singular. For actual benchmark codes, see http://bit.ly/hbench1 and hbench2. Unfortunately, in our system, F_4 takes much more computing time, hence we did not include the result. The results show that, among the algorithms implemented in our system, F_5 works fine in general, though it takes much time in some specific cases. Nevertheless, there remains much room for improvement to compete with the state-of-the-art implementations such as Singular, although there is one case where our implementation is slightly faster than Singular's `groebner` function.

[3] More specifically, we used the implementation in commit 70e6e7b.

Code 9. Main Routine of the F_5 Algorithm

```
f5 :: (Field (Coeff pol), IsOrdPoly pol)
   ⇒ Vector pol → [(Vector pol, pol)]
f5 (map monoize → i0) = runST $ do
  let n = length i0
  gs ← newSTRef []
  ps ← newSTRef $ H.fromList [ basis n i | i ← [0..n-1]]
  syzs ← newSTRef
    [ sVec (i0 ! m) (i0 ! n) | m ← [0..n-1], n ← [0..j-1] ]
  whileJust_ (H.viewMin <$> readSTRef ps) $
  λ (Entry sig g, ps') → do
    ps := ps'
    (gs0, ss0) ← (,) <$> readSTRef gs <*> readSTRef syzs
    unless (standardCriterion sig ss0) $ do
      let (h, ph) = reduceSignature i0 g gs0
          h' = map (× injectCoeff (recip $ leadingCoeff ph)) h
      if isZero ph then syzs :⇐ (mkEntry h : )
        else do
        let adds = fromList $ mapMaybe (regSVec (ph, h')) gs0
        ps :⇐ H.union adds
        gs :⇐ ((monoize ph, mkEntry h') :)
  map (λ (p, Entry _ a) → (a, p)) <$> readSTRef gs
```

Table 2. Benchmark results (ms)

	I_1 (Lex)	I_1 (Grevlex)	I_2 (Lex)	I_2 (Grevlex)	I_3 (Grevlex)
B	1.820×10^0	1.593×10^1	1.400×10^1	4.129×10^0	6.689×10^2
DbyD	6.364×10^1	9.162×10^2	1.147×10^2	5.647×10^1	4.125×10^2
Hilb	1.644×10^2	2.313×10^2	5.265×10^1	3.414×10^1	9.645×10^3
F_5	1.851×10^0	4.314×10^2	7.129×10^0	2.648×10^0	1.290×10^3
S(gr)	2.300×10^0	8.493×10^{-1}	2.651×10^0	8.210×10^{-1}	9.511×10^{-1}
S(sba)	2.279×10^{-1}	8.711×10^{-1}	2.343×10^{-1}	7.958×10^{-1}	1.541×10^{-1}

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

5 Conclusions

In this paper, we have demonstrated how we can adopt the methods developed in the area of functional programming to build a computer algebra system. Some of these methods are also applicable in imperative languages.

In Sect. 2, we presented a type-system strong enough to detect algebraic errors at compile-time. For example, our system can distinguish number of variables of polynomial rings at type-level thanks to dependent types. It also enables us to automatically generate casting functions and we saw how their overhead can be reduced using rewriting rules. As for type-systems for a computer algebra system, there are several existing works [18,23]. However, these systems are not safe enough for discriminating variable arity at type-level and don't make use of rewriting rules.

In Sect. 3, we successfully applied the method of *property-based testing* for verification of the implementation, which is lightweight compared to the existing theorem-prover based approach [2,24]. Although property-based testing is not as rigorous as theorem proving, it is lightweight and can be applied to algorithms not yet proven to be valid or terminate and available also for imperative languages.

We have seen that, in Sect. 4, other features of Haskell, such as higher-order polymorphism, parallelism and laziness, can also be easily applied to computer algebra by actual examples. Even though they are shown as fragments of code, we expect them to be convincing.

Since some of the methods in this paper, such as dependent types or property-based testing, are not limited to the functional paradigm, it might be interesting to investigate their applicability in the imperative settings.

From the viewpoint of efficiency, there is much to be done. For example, efficiency of our current F_4 implementation is far inferior to that of the naïve Buchberger algorithm, and other algorithms are far much slower than state-of-the-art implementations such as Singular. To optimise implementations, we can make more use of Rewriting Rules and efficient data structures. Also, the parallelism must undoubtedly play an important role. Fortunately, there are plenty of the parallel computation functionalities in Haskell, such as Regular Parallel Arrays [16] and `parallel` package [22], and another book by Marlow [21] on general topics in parallelism in Haskell. Also, there is an existing work by Lobachev et al. [20] on parallel symbolic computation in Eden, a dialect of Haskell with parallelism support. Although Eden is retired, the methods introduced there might be helpful.

Acknowledgments. The author would like to thank my supervisor, Prof. Akira Terui, for discussions, and to anonymous reviewers for helpful comments. This research is supported by Grant-in-Aid for JSPS Research Fellow Number 17J00479, and partially by Grants-in-Aid for Scientific Research 16K05035.

References

1. 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 2000, pp. 268–279. ACM, New York (2000). https://doi.org/10.1145/351240.351266
2. Coquand, T., Persson, H.: Gröbner bases in type theory. In: Altenkirch, T., Reus, B., Naraschewski, W. (eds.) TYPES 1998. LNCS, vol. 1657, pp. 33–46. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48167-2_3

3. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion. from lists to streams to nothing at all. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007 (2007)
4. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. ACM SIGPLAN Not. **47**(12), 117–130 (2012). Haskell 2012
5. Faugére, J.-C.: A new efficient algorithm for computing Gröbner bases (F_4). J. Pure Appl. Algebra **139**(1), 61–88 (1999)
6. Faugére, J.-C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, pp. 75–83. ACM, Lille (2002)
7. GHC Team: The Glasgow Haskell Compiler (2018). https://www.haskell.org/ghc/. Accessed 2018
8. Giovini, A., Mora, T., Niesi, G., Robbiano, L., Traverso, C.: "One sugar cube, please" or selection strategies in the Buchberger algorithm. In: Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation, ISSAC 1991, pp. 5–4. ACM (1991)
9. Greuel, G.-M., Pfister, G.: A Singular Introduction to Commutative Algebra, 2nd edn. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-662-04963-1
10. Haskell Committee: The Haskell Programming Language. http://haskell.org/
11. Hypothesis: Most testing is ine ective - Hypothesis (2018). https://hypothesis.works. Accessed 06 May 2018
12. Ishii, H.: The computational-algebra package (2018). https://konn.github.io/computational-algebra
13. Ishii, H.: The ghc-typelits-presburger package (2017). http://hackage.haskell.org/package/ghc-typelits-presburger
14. Ishii, H.: The type-natural package (2013). http://hackage.haskell.org/package/type-natural
15. Jolly, R.: Categories as type classes in the scala algebra system. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2013. LNCS, vol. 8136, pp. 209–218. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02297-0_18
16. 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 2010, pp. 261–272. ACM, Baltimore (2010)
17. Kmett, E.A.: The algebra package (2011). http://hackage.haskell.org/package/algebra. Accessed 2018
18. Kredel, H., Jolly, R.: Generic, type-safe and object oriented computer algebra software. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2010. LNCS, vol. 6244, pp. 162–177. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15274-0_14
19. Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 24–35. ACM, Orlando (1994)
20. Lobachev, O., Loogen, R.: Implementing data parallel rational multiple-residue arithmetic in Eden. In: Gerdt, V.P., Koepf, W., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2010. LNCS, vol. 6244, pp. 178–193. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15274-0_15
21. Marlow, S.: Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly Media, Sebastopol (2013)

22. Marlow, S., Maier, P., Loidl, H.-W., Aswad, M.K., Trinder, P.: Seq no more: better strategies for parallel Haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell 2010, pp. 91–102. ACM, Baltimore (2010). https://doi.org/10.1145/1863523.1863535
23. Mechveliani, S.D.: Computer algebra with Haskell: applying functional-categorial-“lazy” programming. In: Proceedings of International Workshop CAAP, pp. 203–211 (2001)
24. Mechveliani, S.D.: DoCon-A a Provable Algebraic Domain Constructor (2018). http://www.botik.ru/pub/local/Mechveliani/docon-A/2.02/manual.pdf. Accessed 06 May 2018
25. Pan, S., Hu, Y., Wang, B.: The termination of the F5 algorithm revisited. In: Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation, ISSAC 2013, pp. 291–298. ACM, Boston (2013)
26. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell, Haskell 2008, pp. 37–48. ACM, Victoria (2008). https://doi.org/10.1145/1411286.1411292
27. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a promotion. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2012, pp. 53–66. ACM, Philadelphia (2012)