Generic Modular Computations in Aldor

Jinlong Cai, Steve Wilson & Marc Moreno Maza

(University of Western Ontario)

CATLAN-04, 8-9 July

1. Without modular computations, Computer Algebra would remain theory.
2. The key ideas in modular algorithms are quite simple, but implementing them efficiently is often more tricky than the corresponding non-modular algorithms.
3. The main needs are
   (a) good interface with the machine arithmetic
   (b) good data-structures (primitive arrays, ...)
   (c) memory management care (in-place methods, dispose!, ...),
4. Modular methods are generally specialized to a particular polynomial or matrix ring.
5. But in fact, they are essentially based on two recipes:
   (a) The *Chinese Remaindering Theorem*
   (b) The *Hensel Lemma*

Input: $f, g \in \mathbb{Z}[x]$ primitive.

output: $\gcd(f, g)$.

$b := \gcd(\mathrm{lc}(f), \mathrm{lc}(g))$; $d := \min(\deg(f), \deg(g))$; $(m, g_m) := (1, 0)$

repeat

    choose a prime p not dividing $m\,b$

    $g_p := b\,\mathsf{monicGcd}(f \mod p, g \mod p)$ in $\mathbb{Z}/\langle p\rangle[x]$

    $\deg(g_p) = 0 \Rightarrow$ **return** 1

    $\deg(g_p) < d \Rightarrow (m, g_m, d) := (p, g_p, \deg(g_p))$ { previous unlucky }

    $\deg(g_p) > d \Rightarrow$ **iterate** { unlucky reduction }

    $w := \mathsf{combine}(p, m)(g_p, g_m)$; $w := \mathsf{symmetricMod}(w, m\,p)$

    if $w = g_m$ **then** { stabilization }

        $h := \mathrm{pp}(w)$

        if $h \mid f$ **and** $h \mid g$ **then return** h

    $(m, g_m) := (m\,p, w)$

Let E be an Euclidean domain with an Euclidean size δ together with

1. a stream of unassociated primes $p_1, p_2, p_3, \ldots$, such that $\delta(p_1) < \delta(p_1 p_2) < \delta(p_1 p_2 p_3) < \cdots$.
2. a mapping scs from $E \times E \setminus \{0\}$ to E such that

   Simplification. For any $a \in E$ and any $m \in E \setminus \{0\}$ we have:

   $$a \equiv \mathsf{scs}(a, m) \mod m. \tag{1}$$

   Canonicity. For any $m \in E \setminus \{0\}$, any two elements $a, b \in E$, we have:

   $$(a \equiv b \mod m) \iff (\mathsf{scs}(a, m) = \mathsf{scs}(b, m)). \tag{2}$$

   Recoverage = symmetry. All elements of a bounded degree are recovered by the simplifier if the modulus is sufficiently large. That is, for any $B > 0$, there exists $M \in \mathbb{N}$ such that

   $$(\forall (a, m) \in E \times E \setminus \{0\}) \begin{cases} \delta(m) \geq M(B) \\ \delta(a) < B \end{cases} \Rightarrow \mathsf{scs}(a, m) = a. \tag{3}$$

Input: E, Euclidean domain and $f, g \in E[x]$ primitive.

output: $\gcd(f, g)$.

$b := \gcd(\mathrm{lc}(f), \mathrm{lc}(g))$; $d := \min(\deg(f), \deg(g))$; $(m, g_m) := (1, 0)$
repeat
  choose a prime p not dividing $m\,b$
  $g_p := b\ \mathsf{monicGcd}(f \mod p, g \mod p)$ in $E/\langle p\rangle[x]$
  $\deg(g_p) = 0 \Rightarrow$ **return** 1
  $\deg(g_p) < d \Rightarrow (m, g_m, d) := (p, g_p, \deg(g_p))$ { previous unlucky }
  $\deg(g_p) > d \Rightarrow$ **iterate** { unlucky reduction }
  $w := \mathsf{combine}(p, m)(g_p, g_m)$; $w := \mathsf{symmetricMod}(w, m\,p)$
  if $w = g_m$ **then** { stabilization }
    $h := \mathrm{pp}(w)$
    if $h \mid f$ **and** $h \mid g$ **then return** h
  $(m, g_m) := (m\,p, w)$

1. How to implement these *good* Euclidean domains E? We must take into account the fact that the Hensel lifting is *generic* too:
   Let R be a commutative ring with identity element. Let f, g_0, h_0 be univariate polynomials in $R[x]$ and let $m \in R$. We assume that the following relation holds
   $$f \equiv g_0\, h_0 \mod m \tag{4}$$
   We assume also that g_0 and h_0 are relatively prime modulo m, that is there exists $s, t \in R$ such that
   $$s g_0 + t h_0 \equiv 1 \mod m \tag{5}$$
   Then, for every integer ℓ there exist $g^{(\ell)}, h^{(\ell)} \in R[x]$ such that we have
   (a) $f \equiv g^{(\ell)} h^{(\ell)} \mod m^{\ell}$,
   (b) $g_0 \equiv g^{(\ell)} \mod m$.
2. How to implement the residue class rings $E/\langle p \rangle$? We want *genericity* but want to preserve *efficiency*.

```
CanonicalSimplification: Category == CommutativeRing with
        mod: (%, %) -> %
        mod-: (%, %) -> %
        mod+: (%, %, %) -> %
        mod-: (%, %, %) -> %
        mod*: (%, %, %) -> %
        mod^: (%, AldorInteger, %) -> %
        recipMod: (%, %) -> Partial(%)
        invMod: (%, %) -> %
        if (% has EuclideanDomain) then symmetricMod: (%, %) -> %
        default
                mod-(a: %, p: %): % == ..
                mod-(a: %, b: %, p: %): % == ..
                mod+(a: %, b: %, p: %): % == ..
                mod*(a: %, b: %, p: %): % == ..
                mod^(a: %, n: AldorInteger, p: %): % == ..
                invMod(a: %, b: %): % == ..
```

```
SourceOfPrimes: Category == CommutativeRing with
        prime?: % -> Partial(Boolean)
        prime?: % -> Boolean
        getPrime: () -> Partial(%)
        nextPrime: % -> Partial(%)

        if (% has EuclideanDomain) then
             getPrimeOfSize: MachineInteger -> Partial(%)

        default prime?(x: %): Boolean == ..
```

```
ResidueClassRing(R: CommutativeRing, p: R): Category ==
    CommutativeRing with
        modularRep: R -> %
        canonicalPreImage: % -> R

        if (R has EuclideanDomain) then
                symmetricPreImage: % -> R

                if (R has SourceOfPrimes) then
                        import from R pretend SourceOfPrimes
                        if (prime?(p)) then Field
```

```
ModularComputation: Category == CanonicalSimplification with
     residueClassRing: (p: %) -> ResidueClassRing(%, p)

     if (% has EuclideanDomain) then
             combine: (%, %) -> (%, %) -> %
             if (% has IntegerCategory) then
                 combine: (%, MachineInteger) -> (%, MachineInteger) -> %

     default
     if (% has EuclideanDomain) then
             combine(M1: %, M2: %): (%, %) -> % == ..

             if (% has IntegerCategory) then
                  combine(M: %, m: MachineInteger): (%, MachineInteger) -> % == ..
```

```
UnivariatePolynomialCategory0(R: Join(ArithmeticType,
                                      ExpressionType)): Category ==
.........................................................................
if (R has CommutativeRing) then ModularComputation
.........................................................................
.........................................................................
UnivariatePolynomialResidueClass(R: CommutativeRing,
                                 U: UnivariatePolynomialCategory0(R),
                                 p: U):  ResidueClassRing(U, p)
.........................................................................
.........................................................................

UnivariatePolynomialCategory0(R: Join(ArithmeticType,
                                      ExpressionType)): Category ==
.........................................................................
if (R has CommutativeRing) then
    residueClassRing(p: %): ResidueClassRing(%, p) ==
```

```
UnivariatePolynomialResidueClassRing(R pretend CommutativeRing,%,p)
```

- A similar treatement has been applied to integers.
- Now, one can implement the *Generic Modular Gcd Algorithm* as we saw it before.
- But we decided to have fun and implement an optimized one following the implementation of gcd in $\mathbb{Z}[x]$ by Laurent Bernardin and Manuel Bronstein.
  - Their package is parametrized as follows

    `ModularUnivariateGcd(Z:IntegerCategory,U:UnivariatePolynomialCategory0 Z)`
  - It uses a *local* gcd in $\mathbb{Z}/p\mathbb{Z}[x]$ rather than instantiating prime fields.
  - To do so, each polynomial `u: U` modulo a small prime `p` becomes a `PrimitiveArray MachineInteger`.
  - function signatures look like

    `tryprime(a:P, da:SI, amodp:ARR SI, b:P, db:SI, bmodp:ARR SI, lcg:Z, p:SI):P`

```
GenericModularPolynomialGcdPackage(
   R: Join(EuclideanDomain, SourceOfPrimes, ModularComputation),
   U: UnivariatePolynomialCategory(R)): with {
modularGcd: (U, U) -> Partial(U);
tryprime: (U, SI, ARR R, U, SI, ARR R, R) -> (R, SI, ARR R);
remainder!: (SI, ARR R, SI, ARR R, R) -> (SI, ARR R, SI, ARR R);
} == add {
    tryprime(a:U, da:SI, amodp:ARR R, b:U, db:SI, bmodp:ARR R,  p:R): (R, SI, ARR R) == {
       amodp := arrayMod(a, p);   da:= machine degree a;
       bmodp := arrayMod(b, p);   db:= machine degree b; local lb, lr: R;
       repeat {
          (dq, qmodp, dr, rmodp) := remainder!(da, amodp, db, bmodp, p);
          (lr, dr) := leadingCoeffcient(rmodp);
          if zero? dr and zero? lr then break;
          amodp := bmodp; da := db; bmodp := rmodp; db := dr; lb := lr;
       };
       (lb, db, bmodp); }
```

d_x, d_y	sub-resultants	gen mod gcd
10	1600	30
12	4180	50
14	9230	60
16	18570	100
18	34970	130
20	59740	160
30	508440	560

Comparison between *subresultant gcd* and *generic modular gcd* for $\mathbb{Z}/p\mathbb{Z}[x][y]$

d	spe mod gcd	gen mod gcd
200	120	240
250	170	350
300	250	500
350	310	640
400	410	820
450	530	1050
500	700	1280

Comparison between the *specialized modular gcd* and *optimized generic modular gcd* for $\mathbb{Z}[x]$

1. The ratio between the *specialized modular gcd* and *optimized generic modular gcd* is satisfactory. Indeed, the *specialized modular gcd* uses an optimized CRT for integers whereas the *optimized generic modular gcd* uses a generic CRT.
2. What we saw in this talk is part of Aldor 1.0.3 to be downloaded at `www.aldor.org` soon ...
3. We need to compare *optimized generic modular gcd* and *paper-like generic modular gcd*.
4. We are implementing a *generic multivariate Hensel lifting* (Steve Wilson).