Generic Programming Techniques in Aldor

Manuel Bronstein *, Marc Moreno Maza, and Stephen M. Watt

Ontario Research Centre for Computer Algebra
University of Western Ontario
London Canada N6A 5B7

Abstract. Certain problems in mathematical computing present unusual challenges in structuring software libraries. Although generics, or templates as they are sometimes called, have been in wide use now for many years, new ways to use them to improve library architecture continue to be discovered. We find mathematical software to be particularly useful in exploring these ideas because the domain is extremely rich while being well-specified. In this paper we present two techniques for using generics in mathematical computation: The first allows efficient formulation of generic algorithms for modular computations. The second allows mathematical domains to be endowed with additional algorithms after they are defined and to have these algorithms used in generic libraries.

1 Introduction

Generics, that is modules with parametric polymorphism, are now widely accepted as a useful software structuring technique. Indeed, it could be argued that *every* software library can be improved by presenting it in generic form. There are certain difficulties, however, that arise in the construction and use of generic libraries and this impedes there more widespread adoption. Two of the important problems that arise are:

- expressing efficient algorithms while programming via parameterized generic interfaces, and
- accessing specialized algorithms when they exist for particular cases of the generic parameters.

In this paper we show how these problems are solved in the Aldor programming language, using examples from the field of computer algebra. For the first problem, we show how efficient generic algorithms may be obtained for generic modular computation. For the second problem, we show how specialized algorithms can be added as needed to parameter types using *post facto* domain extensions, and illustrate this using an example from linear algebra.

* MB was a close colleague and collaborator in this work. Prior to his untimely death in 2005 he was the leader of Projet Café at INRIA Sophia Antipolis.

2 Generic Modular Computation

We present a technique for *generic modular computation* to illustrate how genericity, efficiency and adaptability may be combined. Modular arithmetic and computations by homomorphic images are essential techniques in Computer Algebra. The key ideas behind modular computations are generally quite simple, but implementing them efficiently is often more tricky than the corresponding non-modular algorithms. The main needs for successful implementation of these ideas are efficient machine arithmetic interface and low-level data-structures (primitive arrays, records) together with some control on memory management and traffic.

Modular methods for polynomial and matrix computations use particular coefficient rings, such as finite fields or residue class rings of Euclidean domains. Moreover, two recipes are used in order to recover the "true" results from their modular images: the *Chinese Remaindering Theorem* and the *Hensel Lemma*. This suggests that, despite of the specialization involved in modular algorithms, a certain level genericity is necessary in order to factorize code. For instance, Kaltofen and Monagan in [3] have observed that one can design a generic modular algorithm for computing univariate polynomials over an Euclidean domain satisfying some reasonable assumptions.

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 $m\,b$
  $g_p := b$ monicGcd$(f \mod p, g \mod p)$ in $E/\langle p\rangle[x]$
  $\deg(g_p) = 0$ => **return** 1
  $\deg(g_p) < d$ => $(m, g_m, d) := (p, g_p, \deg(g_p))$ { previous unlucky }
  $\deg(g_p) > d$ => **iterate** { unlucky reduction }
  $w :=$ combine$(p, m)(g_p, g_m)$; $w :=$ 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
  $(m, g_m) := (m\,p, w)$

The Euclidean domain E of the above algorithm, with Euclidean size δ, must support the following:

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:

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

Recovery. 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

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

We have implemented the above definition through the following four categories in the BasicMath [6] library in Aldor.

```
CanonicalSimplification: Category == CommutativeRing with {
        mod:   (%, %) -> %;
        mod_-: (%, %) -> %;
        mod_+: (%, %, %) -> %;
        mod_-: (%, %, %) -> %;
        mod_*: (%, %, %) -> %;
        mod_^: (%, AldorInteger, %) -> %;
        recipMod: (%, %) -> Partial(%);
        invMod: (%, %) -> %;
        if (% has EuclideanDomain) then symmetricMod: (%, %) -> %;
}

SourceOfPrimes: Category == CommutativeRing with {
        prime?: % -> Partial(Boolean);
        prime?: % -> Boolean;
        getPrime: () -> Partial(%);
        nextPrime: % -> Partial(%);
        if (% has EuclideanDomain) then
             getPrimeOfSize: MachineInteger -> Partial(%);
}

ResidueClassRing(R: CommutativeRing, p: R): Category ==
    CommutativeRing with {
        modularRep: R -> %;
        canonicalPreImage: % -> R;
        if (R has EuclideanDomain) then {
                symmetricPreImage: % -> R;
         }
    }
```

```
ModularComputation: Category == CanonicalSimplification with {
     residueClassRing: (p: %) -> ResidueClassRing(%, p);
     if (% has EuclideanDomain) then {
             combine: (%, %) -> (%, %) -> %;
             if (% has IntegerCategory) then  {
                 combine: (%, MachineInteger) -> (%, MachineInteger) -> %;
             }
     }
}
```

A ring R satisfying `ModularComputation` must implement an operation `residueClassRing`, such that, given an element $p \in R$, this function implements R/pR. It follows from the above categories that any Euclidean domain R satisfying `ModularComputation` can implement the generic modular algorithm for univariate polynomials over R. Actually, we have implemented this algorithm in a package

```
GenericModularPolynomialGcdPackage(
   R: Join(EuclideanDomain, SourceOfPrimes, ModularComputation),
   U: UnivariatePolynomialCategory(R)): with {
       modularGcd: (U, U) -> Partial(U);
}
```

For computing GCDs in $\mathbb{Z}/p\mathbb{Z}[x][y]$, we have realized benchmarks between our package and a non-modular implementation based on the sub-resultants. The timings below are in ms.

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

Them for computing GCDs in $\mathbb{Z}[x]$, we have realized benchmarks between a specialized implementation of the modular gcd algorithm to $\mathbb{Z}[x]$ and our generic modular implementation instantiated at $\mathbb{Z}[x]$.

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

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.

3 Specialized Algorithms

Another of the difficulties with generic programming is that there are often specialized algorithms that apply over certain domains. In C++ this is handled by template specialization and is resolved statically. However, in Aldor types may be constructed dynamically so we need some other mechanism to access specialized algorithms. Post facto extension, combined with conditional category tests, allows generic code to use special purpose algorithms, when applicable, without revising library components. We illustrate this point with an example from linear algebra, presented in [7].

A linear algebra package can be defined generically over any commutative ring. More efficient algorithms may be used, however, when the ring is known to be an integral domain or a field. We may thus assemble these algorithms into a package as follows:

```
LinearAlgebra(R:CommutativeRing, M:MatrixCategory R):
with {...} == add {
    local Elim: LinearEliminationCategory(R, M) == {
        R has Field =>
            OrdinaryGaussElimination(R, M);
        R has IntegralDomain =>
            TwoStepFractionFreeGaussElimination(R,M);
        DivisionFreeGaussElimination(R, M);
    }

    determinant(m:M):R == determinant(m)$Elim;
}
```

Certain coefficient rings may support efficient specialized algorithms. For example, we may want to compute over the integers using Chinese remaindering. However, we do not want to have to modify the `LinearAlgebra` package whenever a new method is incorporated into the library. We therefore define a category that a ring can implement to provide linear algebra algorithms over itself:

```
LinearAlgebraRing: Category == with {
    determinant: (M:MatrixCategory %) -> M -> %;
    rank:        (M:MatrixCategory %) -> M -> Integer;
    ...
}
```

We make one modification to the `LinearAlgebra` package to take advantage of special-case algorithms carried in a `LinearAlgebraRing` view: we replace the `determinant` function with the following version

```
determinant(m:M):R == {
    if R has LinearAlgebraRing then
        determinant(M)(a)$R;
    else
        determinant(m)$Elim;
}
```

When we have special algorithms for some domain, we extend the domain to know about them:

```
extend Integer: LinearAlgebraRing == add {
    determinant(M: MatrixCategory %)(m: M): % ==
        ChineseRemainderingDeterminant(M, m);
    rank(M: MatrixCategory %)(m: M): % ==
        ChineseRemainderingRank(M, m);
    ...
}
```

Whenever we use the `LinearAlgebra` package, it will use the designated algorithm even if the coefficient ring is determined dynamically.

The technique of using post facto extensions to endow domains with special-case algorithms has been used in the in the construction of the $\sum^{IT}$ library [1]. The notion of rings knowing how to perform operations in structures over themselves has been explored earlier in relation to composing factorization algorithms [2].

References

1. Bronstein, M.: *$\sum^{IT}$: A strongly-typed embeddable computer algebra library*, Proc. DISCO'96, LNCS 1128 pp. 22-33, Springer Verlag.
2. Davenport, J., Gianni, P., Trager, B.: *Scratchpad's view of algebra II: a categorical view of factorization*, Proc. 1991 International Symposium on Symbolic and Algebraic Computation, pp. 32–38, ACM Press.
3. Kaltofen, E., Monagan, M.: *On the Genericity of the Modular Polynomial GCD Algorithm*, Proc. 1999 International Symposium on Symbolic and Algebraic Computation, ACM Press.
4. Watt, S.M.: Aldor. In Grabmeier, J., Kaltofen, E., Weispfenning, V., eds.: Handbook of Computer Algebra, Springer Verlag (2003) 265–270
5. Aldor.org: Aldor user guide. `http://www.aldor.org/AldorUserGuide` (2003)
6. The Computational Mathematics Group: The BasicMath library. NAG Ltd, Oxford, UK. `http://www.nag.co.uk/projects/FRISCO.html` (1998)
7. Watt, S.M.: Post facto Type Extensions for Mathematical Programming, Proc. Domain-Specific Aspect langauges (SIGPLAN/SIGSOFT DSAL 2006), October 23, 2006, Portland OR, USA.