Introduction to Polynomial Parser

Pavel Emeliyanenko

August 10, 2012

1 Introduction

The purpose of the polynomial parser is to provide an alternative way to input CGAL polynomials in a human-readable format. By default, `CGAL::Polynomial` class provides an `operator >>` to read polynomials from an input stream in an ASCII format:

$$\begin{aligned}&P[8(0,P[8(0,P[8(0,2)(2,-16)(4,80)(6,-128)(8,64)])(2,P[0(0,-16)])(4,P[0(0,80)])\\&\quad(6,P[0(0,-128)])(8,P[0(0,64)])])(2,P[0(0,P[0(0,-16)])])(4,P[0(0,P[0(0,80)])])\\&\quad(6,P[0(0,P[0(0,-128)])])(8,P[0(0,P[0(0,64)])])]\end{aligned}$$

which corresponds to the following trivariate polynomial:

$$\begin{aligned}&64z^8+(-128)z^6+80z^4+(-16)z^2+(64y^8+(-128)y^6+80y^4+(-16)y^2+\\&\quad(64x^8+(-128)x^6+80x^4+(-16)x^2+2)).\end{aligned}$$

The parser can handle polynomials in any number of indeterminates (variables) and can parse arbitrary algebraic expressions, e.g., those of the form:

$$\begin{aligned}&((y-1)^4-x*y^3)^5+(x+z)^2-(2123234523*x^2-2*y*y*x+3^{10}*x*132123)^3,\\&(y+x-z+w-3)^3=x+y-123/12312+1.00001*z*x^2.\end{aligned}$$

Note that, the multiplication symbol ($*$) is optional and can be omitted in the above expressions.

The parser's behavior is fully customizeable vie the so-called *parser policy* which is supplied as a template parameter to the parser. In the first place, the policy is responsible for reading polynomial coefficients from the input stream and performing all necessary type conversions for the output polynomial. This, for instance, allows us to mix integer, rational and floating-point numbers in a single expression as demonstrated in the above examples. Besides, the parser policy can be used to perform maximal degree check, detect various erroneous situations (such as division by zero) or rename the polynomial variables.

2 Interface

The parser is instantiated with a polynomial type and the corresponding parser policy:

```
template <class Poly_d_, class ParserPolicy =
                        Default_parser_policy< Poly_d_ > >
struct Polynomial_parser_d {
    // this instance's template argument
    typedef Poly_d_ Poly_d;
    // this instance's second template argument
    typedef ParserPolicy Policy;
public:
    // default constructor
    Polynomial_parser_d(const Policy& policy_ = Policy()) :
            policy(policy_) {
    }
    // functor invokation operator
    bool operator()(const std::string& in, Poly_d& poly);
    ...
};
```

In the listing, `Poly_d_` defines the type of output polynomial to be constructed and `ParserPolicy` is a parser policy. The function invokation operator takes an input string `in` and constructs a polynomial of type `Poly_d_`. It returns `true` if the parsing was successful.

The default policy provides no type explicit conversions: in other words, polynomials are parsed "as it is" which means that there must be implicit type coercion between coefficients read from the input stream and the resulting polynomial coefficient. The default parser policy has the following interface which any user-defined policy must provide:

```
template < class Poly_d_, class InputCoeff >
struct Default_parser_policy {
    // first template argument type
    typedef Poly_d_ Poly_d;
    // second template argument type
    typedef InputCoeff Input_coeff;
    // coefficient type
    typedef typename CGAL::Polynomial_traits_d< Poly_d >::
        Innermost_coefficient_type Coeff;

    // default constructor
    Default_parser_policy() { }

    // reads a coefficient from the input stream
    Coeff read_coefficient(std::istream& is) const;

    // check for degree overflow
    virtual bool exponent_check(unsigned) const {
        return true;
    }

    virtual ~Default_parser_policy() { }

    // variable names listed in the order from innermost to the outermost
    // variable as they appear in the resulting equation
    static const int n_var_names = 4;
    static const char *var_names_lower, *var_names_upper;
    ...
};
```

The first template parameter, `Poly_d_`, is a type of output polynomial while the second one, `InputCoeff`, defines the type of polynomial coefficients as they are read from the input stream via `operator >>`. There must be an implicit type coercion between `InputCoeff` and the type of innermost (scalar) coefficients of `Poly_d_`.

Besides, the "parser policy" offers a great flexibility in controlling the way how the output polynomials are constructed. All type conversions, precision and rounding of polynomial coefficients are ultimately handled by a given parser policy, which completely separates the parser from the number type issues.

References