POLYNOMIAL REPRESENTATION IN ACL2

KEONHO LEE

1. Introduction

Originally, I built a calculator which interprets an ACL2 list input and evaluates what the expression returns in ACL2. After building it, I wanted to extend this and add more features in the calculator. Then I came up with a polynomial calculator that evaluates lists with not only numbers but also variables, such as x and y. To do this, I focused on how to represent polynomials and operations related to this, i.e. addition, subtraction, and multiplication.

2. Polynomial Representation

To make the problem simple, we are only considering single-variable, integer-coefficient polynomials. Also, to avoid confusion, all polynomials described after use the variable x only. Notice that polynomials have similar properties as the ordinals: polynomials are the sum of some number or infinite number of terms, where each term is a product of a number and a power of variable. The critical difference is that the coefficient in ordinals is a positive integer, where a polynomial can have any coefficient depending on how it is defined. In our case, the coefficient is integer. Now the following table shows some examples of polynomials and their representations in ACL2:

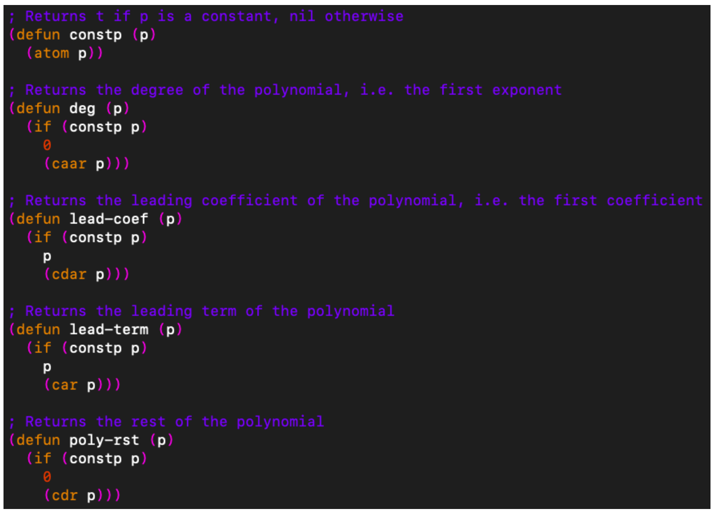
Polynomial	ACL2 Representation
0	0
3	3
x	((1 . 1) . 0)
$x+3$	((1 . 1) . 3)
$2x-5$	((1 . 2) . -5)
x^2	((2 . 1) . 0)
x^2+2x+3	((2 . 1) (1 . 2) . 3)
$-3x^5+4x^2+7x-13$	((5 . -3) (2 . 4) (1 . 7) . -13)

3. Function Definitions

We built polynomial representation above, so now we define some functions related to this. The following are the five functions which are frequently used, constp, deg, lead-coef, lead-term, and poly-rst, shown in Figure 1:

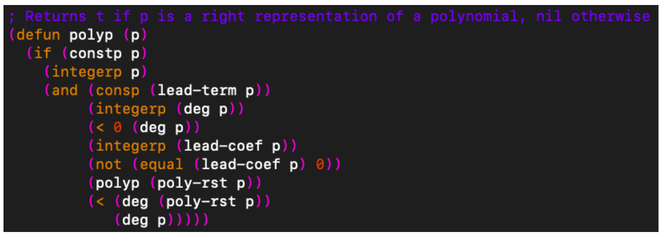
Using those functions, we define polyp, which determines whether a given expression is a correct form of representation of a polynomial we defined above, shown in Figure 2.

Date: May 10, 2019.



```lisp
; Returns t if p is a constant, nil otherwise
(defun constp (p)
  (atom p))

; Returns the degree of the polynomial, i.e. the first exponent
(defun deg (p)
  (if (constp p)
    0
    (caar p)))

; Returns the leading coefficient of the polynomial, i.e. the first coefficient
(defun lead-coef (p)
  (if (constp p)
    p
    (cdar p)))

; Returns the leading term of the polynomial
(defun lead-term (p)
  (if (constp p)
    p
    (car p)))

; Returns the rest of the polynomial
(defun poly-rst (p)
  (if (constp p)
    0
    (cdr p)))
```

Figure 1. Frequently used functions

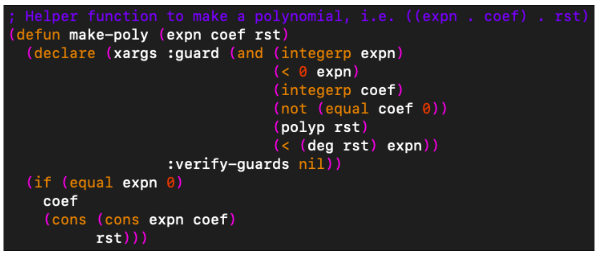


```lisp
; Returns t if p is a right representation of a polynomial, nil otherwise
(defun polyp (p)
  (if (constp p)
    (integerp p)
    (and (consp (lead-term p))
         (integerp (deg p))
         (< 0 (deg p))
         (integerp (lead-coef p))
         (not (equal (lead-coef p) 0))
         (polyp (poly-rst p))
         (< (deg (poly-rst p))
            (deg p)))))
```

Figure 2. polyp

Note that those functions resemble the functions related to ordinals, such as o-first-expt, o-first-coeff, and o-p. This is because we choose the representation of polynomials to be similar to that of ordinals.

Now we present four operations related to polynomial: addition, negation, subtraction, and multiplication. Here is a helper function that makes the code more compact, shown in Figure 3: Using the functions above, we define addition, negations and sub-

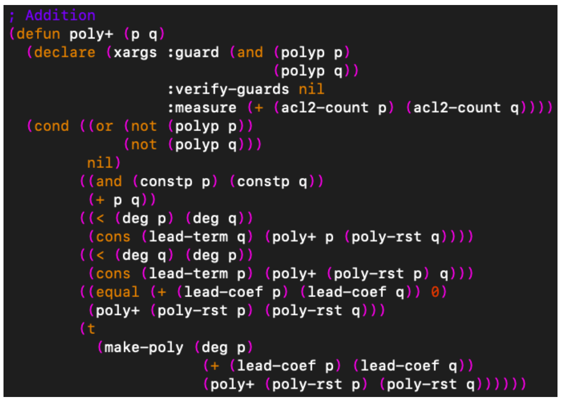


```lisp
; Helper function to make a polynomial, i.e. ((expn . coef) . rst)
(defun make-poly (expn coef rst)
  (declare (xargs :guard (and (integerp expn)
                              (< 0 expn)
                              (integerp coef)
                              (not (equal coef 0))
                              (polyp rst)
                              (< (deg rst) expn))
                  :verify-guards nil))
  (if (equal expn 0)
    coef
    (cons (cons expn coef)
          rst)))
```

Figure 3. make-poly

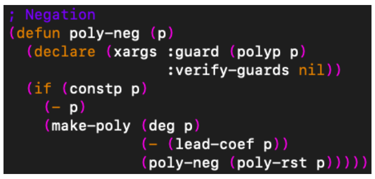
traction, shown in Figure 4, Figure 5, and Figure 6. Note that subtraction is a composition of addition and negation.

For multiplication, we have to define a helper function function. We first consider a multiplication of a monomial, a polynomial with a single term, and a polynomial. Here is a function determines whether a given polynomial is a monomial shown in Figure 7. Note that a monomial can be either containing a variable or not, i.e. the representation of a monomial can be either consp or not.



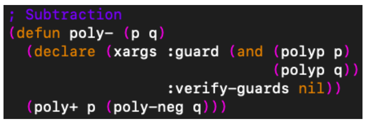
```
; Addition
(defun poly+ (p q)
  (declare (xargs :guard (and (polyp p)
                              (polyp q))
                  :verify-guards nil
                  :measure (+ (acl2-count p) (acl2-count q))))
  (cond ((or (not (polyp p))
             (not (polyp q)))
         nil)
        ((and (constp p) (constp q))
         (+ p q))
        ((< (deg p) (deg q))
         (cons (lead-term q) (poly+ p (poly-rst q))))
        ((< (deg q) (deg p))
         (cons (lead-term p) (poly+ (poly-rst p) q)))
        ((equal (+ (lead-coef p) (lead-coef q)) 0)
         (poly+ (poly-rst p) (poly-rst q)))
        (t
          (make-poly (deg p)
                     (+ (lead-coef p) (lead-coef q))
                     (poly+ (poly-rst p) (poly-rst q)))))))
```

FIGURE 4. Addition



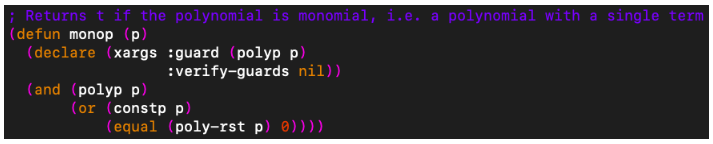
```
; Negation
(defun poly-neg (p)
  (declare (xargs :guard (polyp p)
                  :verify-guards nil))
  (if (constp p)
    (- p)
    (make-poly (deg p)
               (- (lead-coef p))
               (poly-neg (poly-rst p)))))
```

FIGURE 5. Negation



```
; Subtraction
(defun poly- (p q)
  (declare (xargs :guard (and (polyp p)
                              (polyp q))
                  :verify-guards nil))
  (poly+ p (poly-neg q)))
```

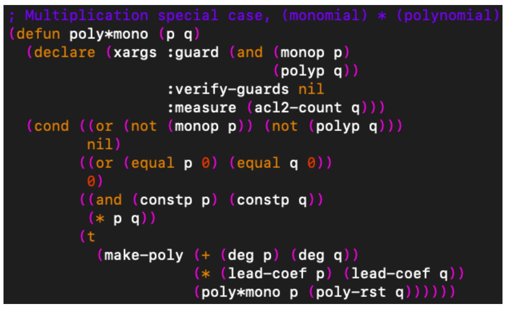
FIGURE 6. Subtraction



```
; Returns t if the polynomial is monomial, i.e. a polynomial with a single term
(defun monop (p)
  (declare (xargs :guard (polyp p)
                  :verify-guards nil))
  (and (polyp p)
       (or (constp p)
           (equal (poly-rst p) 0))))
```

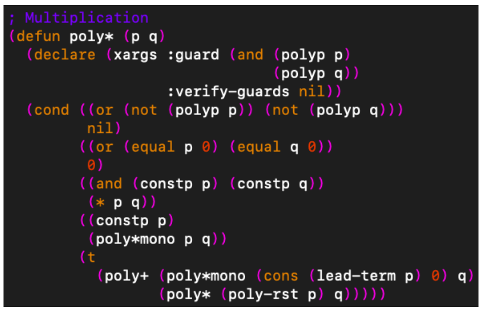
FIGURE 7. monop

With this in mind, we define a special case multiplication shown in Figure 8, and a general multiplication function shown in Figure 9.



```
; Multiplication special case, (monomial) * (polynomial)
(defun poly*mono (p q)
  (declare (xargs :guard (and (monop p)
                              (polyp q))
                  :verify-guards nil
                  :measure (acl2-count q)))
  (cond ((or (not (monop p)) (not (polyp q)))
         nil)
        ((or (equal p 0) (equal q 0))
         0)
        ((and (constp p) (constp q))
         (* p q))
        (t
          (make-poly (+ (deg p) (deg q))
                     (* (lead-coef p) (lead-coef q))
                     (poly*mono p (poly-rst q))))))
```

FIGURE 8. Monomial multiplication

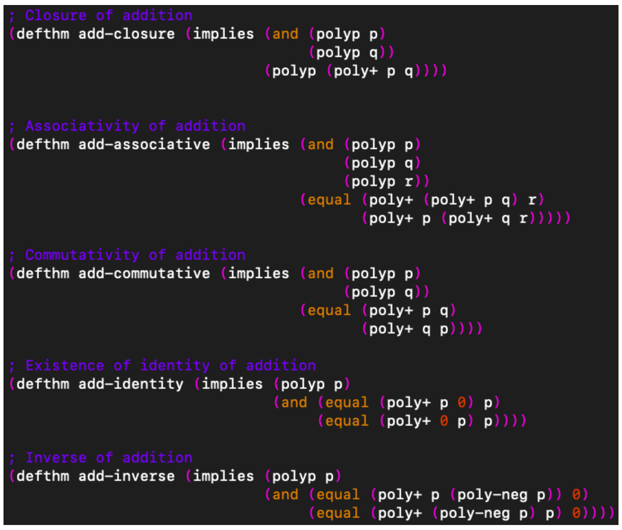


```
; Multiplication
(defun poly* (p q)
  (declare (xargs :guard (and (polyp p)
                              (polyp q))
                  :verify-guards nil))
  (cond ((or (not (polyp p)) (not (polyp q)))
         nil)
        ((or (equal p 0) (equal q 0))
         0)
        ((and (constp p) (constp q))
         (* p q))
        ((constp p)
         (poly*mono p q))
        (t
          (poly+ (poly*mono (cons (lead-term p) 0) q)
                 (poly* (poly-rst p) q)))))
```

FIGURE 9. Multiplication

4. THEOREMS

Now we explore the powerful advantage of ACL2; we will prove theorems related to the operations defined above. Here are five theorems relative to addition shown in Figure 10.



```
; Closure of addition
(defthm add-closure (implies (and (polyp p)
                                  (polyp q))
                             (polyp (poly+ p q))))


; Associativity of addition
(defthm add-associative (implies (and (polyp p)
                                      (polyp q)
                                      (polyp r))
                                 (equal (poly+ (poly+ p q) r)
                                        (poly+ p (poly+ q r)))))

; Commutativity of addition
(defthm add-commutative (implies (and (polyp p)
                                      (polyp q))
                                 (equal (poly+ p q)
                                        (poly+ q p))))

; Existence of identity of addition
(defthm add-identity (implies (polyp p)
                              (and (equal (poly+ p 0) p)
                                   (equal (poly+ 0 p) p))))

; Inverse of addition
(defthm add-inverse (implies (polyp p)
                             (and (equal (poly+ p (poly-neg p)) 0)
                                  (equal (poly+ (poly-neg p) p) 0))))
```

FIGURE 10. Addition theorems

And here are five theorems relative to multiplication shown in Figure 11.

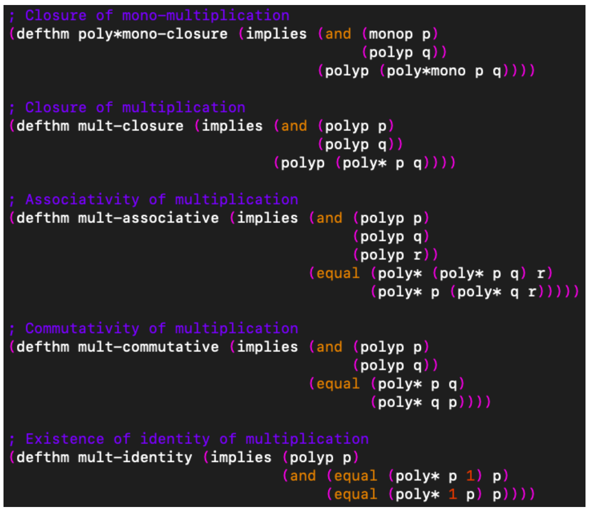
Finally, distributive law and theorem related to the degree of polynomials shown in Figure 12.

Note that showing the theorems above shows that the polynomial and operations, addition and multiplication, form a commutative ring.

5. CONCLUSION

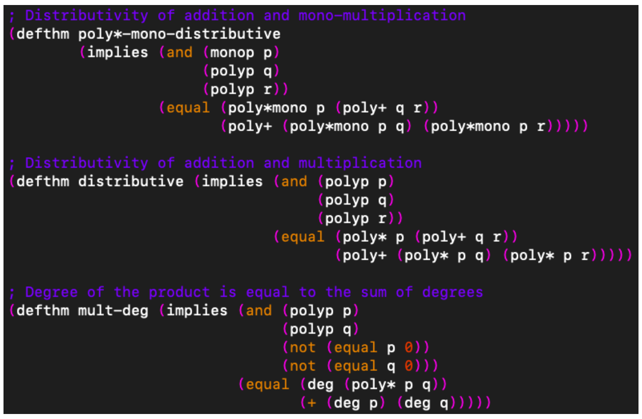
The theorems I did not get are associativity of multiplication, commutativity of multiplication, and distributivity of addition and multiplication. Most of the theorems require a lots of case splits, which made the proof hard to handle and complicated.

The biggest possible extension of it is a polynomial ring with field coefficient. It is possible to change the domain of the coefficients from integer to ring or field, which is built in ACL2 book, and there are numerous theorems related to this field of mathematics.



```
; Closure of mono-multiplication
(defthm poly*mono-closure (implies (and (monop p)
                                        (polyp q))
                                   (polyp (poly*mono p q))))

; Closure of multiplication
(defthm mult-closure (implies (and (polyp p)
                                   (polyp q))
                              (polyp (poly* p q))))

; Associativity of multiplication
(defthm mult-associative (implies (and (polyp p)
                                       (polyp q)
                                       (polyp r))
                                  (equal (poly* (poly* p q) r)
                                         (poly* p (poly* q r)))))

; Commutativity of multiplication
(defthm mult-commutative (implies (and (polyp p)
                                       (polyp q))
                                  (equal (poly* p q)
                                         (poly* q p))))

; Existence of identity of multiplication
(defthm mult-identity (implies (polyp p)
                               (and (equal (poly* p 1) p)
                                    (equal (poly* 1 p) p))))
```

Figure 11. Multiplication theorems



```
; Distributivity of addition and mono-multiplication
(defthm poly*-mono-distributive
        (implies (and (monop p)
                      (polyp q)
                      (polyp r))
                 (equal (poly*mono p (poly+ q r))
                        (poly+ (poly*mono p q) (poly*mono p r)))))

; Distributivity of addition and multiplication
(defthm distributive (implies (and (polyp p)
                                   (polyp q)
                                   (polyp r))
                              (equal (poly* p (poly+ q r))
                                     (poly+ (poly* p q) (poly* p r)))))

; Degree of the product is equal to the sum of degrees
(defthm mult-deg (implies (and (polyp p)
                               (polyp q)
                               (not (equal p 0))
                               (not (equal q 0)))
                          (equal (deg (poly* p q))
                                 (+ (deg p) (deg q)))))
```

Figure 12. Distributivity & Degree