3740 Group Project Report

By Kegan Rieder and Justin Wolfenden

Overview

The project aimed to design a collection of functions for manipulating polynomials represented as either sparse or dense types. This led to the issue that one representation contained a sub-list (sparse type), and another was just a list of values (dense type). The solution to this problem was to have a main function check the kind of a given polynomial and then call a helper function, converting the given polynomial to whichever type we consider easier for the current operation. Then, the main function would convert the return value if it's a polynomial to the proper representation of either sparse (both polynomials are sparse) or dense (one or both polynomials are dense).

Function Overview

Is-sparse? (Polynomial)

- Checks if the polynomial is empty. If yes, returns false (not sparse).
- Checks if the first element is a sub-list. If yes, returns true (sparse); otherwise, returns false (dense).

To-sparse (Polynomial)

- Checks if the polynomial is already sparse. If yes, returns the polynomial.
- If not sparse, calls the helper function make-sparse(polynomial 0) to convert and returns the result.

Make-sparse(polynomial Degree)

- Checks if the list is empty; if yes, returns ().
- If not empty, checks if the next element is 0. If yes, skips it; otherwise, calls itself with the next element and increased degree.

Is-dense? (Polynomial)

- Checks if the first item is a sub-list. If yes, returns false (sparse); otherwise, returns true (dense).

To-dense (Polynomial)

- Checks if the polynomial is already dense. If yes, returns the polynomial.
- If not dense, calls make-dense(Polynomial Degree) to make it dense.

Make-dense(Polynomial Degree)

- Checks if the list is empty. If yes, returns ().
- If not empty, checks if **y** is equal to the second item in the first list. If yes, adds the first item to the new list; otherwise, adds 0. Calls itself with the next element and increased degree.

Is-zero?(polynomial)

- Checks if the list is empty. If yes, returns true.
- If not empty, checks if the list is sparse. If yes, checks if the first item in the first list is zero. If yes, returns true; otherwise, returns false.
- If not sparse, checks if the first item in the list is zero and returns true if yes, false if not.

Coeff(polynomial degree)

- Checks if the list is empty. If yes, returns 0.
- If not empty, checks if it's sparse. If yes, checks the second item in the first list; if equal to y, returns the first item. If not equal, calls itself with the next element and y.

- If not sparse, returns Coeff(to-sparse x 0 y).

Degree(polynomial)

- Checks if the polynomial is zero. If yes, returns -inf.0.
- If not zero, checks if it's sparse. If yes, returns the last item; if not sparse, returns Degree(to-sparse polynomial 0).

Eval(polynomial k)

- Checks if the polynomial is empty. If yes, returns 0.
- If not empty, checks if it's dense. If dense, returns Eval(to-sparse polynomial 0). If sparse, applies an equation to each item and returns the result.

Add(poly1 poly2)

- Checks if both polynomials are empty. If yes, returns (0).
- If one is empty, returns the other.
- If both are not empty, checks if both are sparse. If yes, returns To-sparse(Add-Poly(To-dense Poly1) (To-dense Poly2)). If not, returns Add-Poly(To-dense Poly1) (To-dense Poly2).

Add-poly(poly1 poly2)

- Adds to polynomials following type check and conversion in parent function add or subtract
- Checks if both lists are empty. If yes, returns ().
- If Poly1 is empty, (append (list (car poly2)) (add-Poly poly1 (cdr poly2))).
- If Poly2 is empty, (append (list (car poly2)) (add-Poly poly1 (cdr poly2))).
- Run recursion, adding the two polynomials until both are empty

Invert-coef(polynomial)

- Only works for dense. Returns the inverse of polynomial

Subtract(poly1 poly2)

- Perform subtraction on to polynomials following checking there type checks, which it uses output the difference in the correct representation.
- If both are sparse run (to-sparse (add-Poly (to-dense poly1) (invert-coef (to-dense poly2)))) which inverts and makes to dense, then converts the difference to sparse
- If both are sparse run (add-Poly (to-dense poly1) (invert-coef (to-dense poly2))) which inverts and makes to dense, then just returns result

Multiply-terms(poly1 poly2)

- Multiplies corresponding terms of sparse lists poly1 and poly2, returning a list with the new coefficient and exponent for the first item.

Apply-poly1-to-p2term(poly1 poly2)

- Only works with sparse,
- Multiplies each term of poly1 into poly2 recursively, utilizing multiply-terms. Returns an empty list if poly1 is empty.

Multiply-poly(poly1 poly2)

- Checks the length of poly2 and multiplies every term in poly1 by the terms in poly2. Converts to dense for ease of implementation using apply-poly1-to-p2term.

- - If poly2 length is greater than 2, recursively adds the results of apply-poly1-to-p2term for the first and second terms of poly2.
- If poly2 length is 1, applies poly1 to the single term in poly2.

Multiply(poly1 poly2)

- If either polynomial is zero, returns 0.
- If both polynomials are sparse, multiplies them using Multiply-poly.
- If at least one polynomial is dense, converts both to sparse, multiplies them using Multiply-poly, and converts the result back to dense.

div-terms (term1 term2)

- Divides sparse polynomial terms.
- Returns a list with the result of term1 divided by term2

quotient (poly1 poly2)

- function finds the quotient of poly1 divided by poly2.
- If poly2 is zero, it returns -inf.0. If one or both polynomials are dense, it converts them to sparse and then back to dense after the calculation in poly-div. If both polynomials are sparse, it returns the result of reversing the order of the arguments to poly-div, running poly-div, and then reversing the result.

get-remainder (poly1 poly2)

- function finds the remainder of dividing poly1 by poly2.
- It first checks if there is a remainder by using the subtract function. If there is no remainder, it returns an empty list. If there is a remainder, It performs the following operation to find the remainder poly1 - poly2 * (poly-div (reverse poly1) (reverse poly2)) = remainder

remainder (poly1 poly2)

- If poly2 is zero, it returns -inf.0. If one or both polynomials are dense, it converts them to sparse and then back to dense after the calculation in get-remainder. If both polynomials are sparse, it returns the result of get-remainder,

power-rule(poly1)

- Applies the power rule to a sparse term in poly1.
- Returns an empty list if the second term is less than 1, otherwise applies the power rule.

derivative(poly1)

- This function returns the derivative of poly1. It can use both sparse and dense polynomials but will convert them to dense and then back to the correct representation following derivative calculation.

normalize-gcd(polynomial coefficient)

polynomial is the polynomial you want to divide, coefficient is the int you want to divide by, only works with dense.

Returns (map (lambda (term) (/ term coefficent)) Polynomial) which is a list of all terms in polynomial divided by coefficent

find-gcd(poly1 poly2)

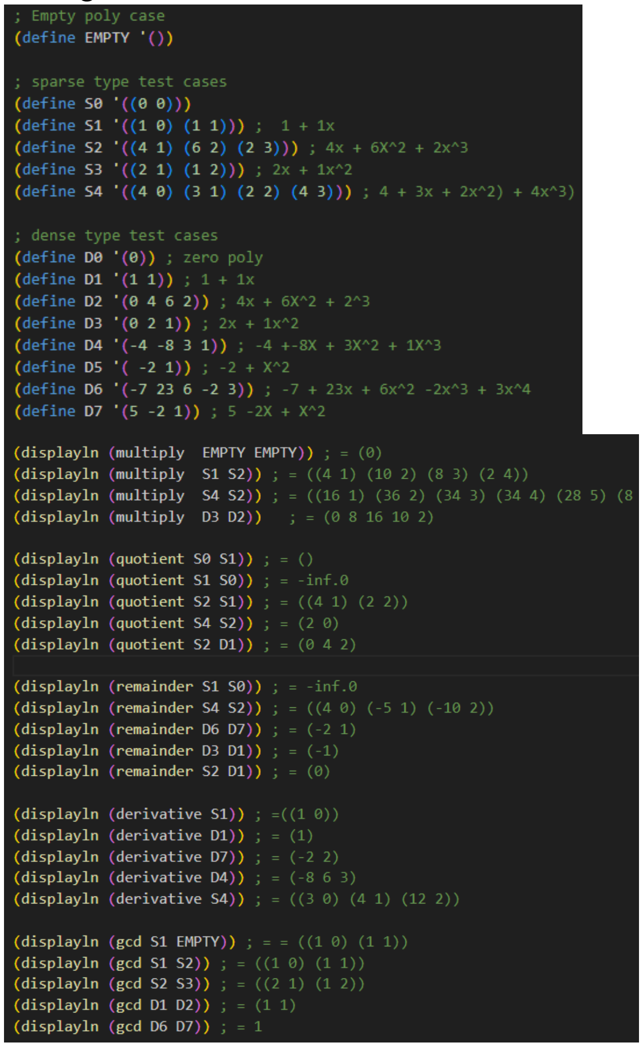
- Finds the greatest common divisor (gcd) of two polynomials (sparse).
- Uses normalize-gcd, degree, and remainder.

gcd(poly1 poly2)

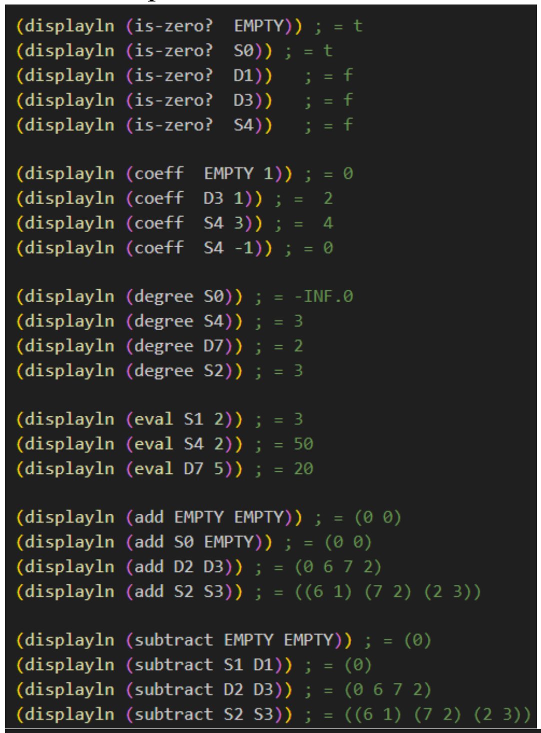
- Finds the gcd of two polynomials (sparse or dense
- Converts dense polynomials to sparse for ease of implementation.
- Returns (0) if both polynomials are zero. Returns poly2 if poly1 is zero. Returns poly1 if poly2 is zero.
- Compares degrees and uses find-gcd to find the gcd
- Converts the results back to there proper polynomial type.

Limitations/Bugs

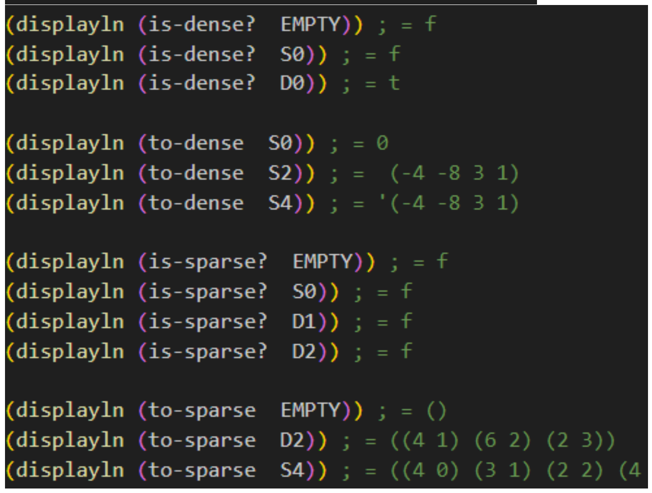
Using the following sample tests (more can be found under test section in polynomial.rkt) no bugs were able to be found, and all test cases were able to pass.



```
; Empty poly case
(define EMPTY '())

; sparse type test cases
(define S0 '((0 0)))
(define S1 '((1 0) (1 1))) ;  1 + 1x
(define S2 '((4 1) (6 2) (2 3))) ; 4x + 6X^2 + 2x^3
(define S3 '((2 1) (1 2))) ; 2x + 1x^2
(define S4 '((4 0) (3 1) (2 2) (4 3))) ; 4 + 3x + 2x^2) + 4x^3)

; dense type test cases
(define D0 '(0)) ; zero poly
(define D1 '(1 1)) ; 1 + 1x
(define D2 '(0 4 6 2)) ; 4x + 6X^2 + 2^3
(define D3 '(0 2 1)) ; 2x + 1x^2
(define D4 '(-4 -8 3 1)) ; -4 +-8X + 3X^2 + 1X^3
(define D5 '( -2 1)) ; -2 + X^2
(define D6 '(-7 23 6 -2 3)) ; -7 + 23x + 6x^2 -2x^3 + 3x^4
(define D7 '(5 -2 1)) ; 5 -2X + X^2

(displayln (multiply  EMPTY EMPTY)) ; = (0)
(displayln (multiply  S1 S2)) ; = ((4 1) (10 2) (8 3) (2 4))
(displayln (multiply  S4 S2)) ; = ((16 1) (36 2) (34 3) (34 4) (28 5) (8
(displayln (multiply  D3 D2))   ; = (0 8 16 10 2)

(displayln (quotient S0 S1)) ; = ()
(displayln (quotient S1 S0)) ; = -inf.0
(displayln (quotient S2 S1)) ; = ((4 1) (2 2))
(displayln (quotient S4 S2)) ; = (2 0)
(displayln (quotient S2 D1)) ; = (0 4 2)

(displayln (remainder S1 S0)) ; = -inf.0
(displayln (remainder S4 S2)) ; = ((4 0) (-5 1) (-10 2))
(displayln (remainder D6 D7)) ; = (-2 1)
(displayln (remainder D3 D1)) ; = (-1)
(displayln (remainder S2 D1)) ; = (0)

(displayln (derivative S1)) ; =((1 0))
(displayln (derivative D1)) ; = (1)
(displayln (derivative D7)) ; = (-2 2)
(displayln (derivative D4)) ; = (-8 6 3)
(displayln (derivative S4)) ; = ((3 0) (4 1) (12 2))

(displayln (gcd S1 EMPTY)) ; = = ((1 0) (1 1))
(displayln (gcd S1 S2)) ; = ((1 0) (1 1))
(displayln (gcd S2 S3)) ; = ((2 1) (1 2))
(displayln (gcd D1 D2)) ; = (1 1)
(displayln (gcd D6 D7)) ; = 1
```



```
(displayln (is-zero?  EMPTY)) ; = t
(displayln (is-zero?  S0)) ; = t
(displayln (is-zero?  D1))   ; = f
(displayln (is-zero?  D3))   ; = f
(displayln (is-zero?  S4))   ; = f

(displayln (coeff  EMPTY 1)) ; = 0
(displayln (coeff  D3 1)) ; =  2
(displayln (coeff  S4 3)) ; =  4
(displayln (coeff  S4 -1)) ; = 0

(displayln (degree S0)) ; = -INF.0
(displayln (degree S4)) ; = 3
(displayln (degree D7)) ; = 2
(displayln (degree S2)) ; = 3

(displayln (eval S1 2)) ; = 3
(displayln (eval S4 2)) ; = 50
(displayln (eval D7 5)) ; = 20

(displayln (add EMPTY EMPTY)) ; = (0 0)
(displayln (add S0 EMPTY)) ; = (0 0)
(displayln (add D2 D3)) ; = (0 6 7 2)
(displayln (add S2 S3)) ; = ((6 1) (7 2) (2 3))

(displayln (subtract EMPTY EMPTY)) ; = (0)
(displayln (subtract S1 D1)) ; = (0)
(displayln (subtract D2 D3)) ; = (0 6 7 2)
(displayln (subtract S2 S3)) ; = ((6 1) (7 2) (2 3))
```



```
(displayln (is-dense?  EMPTY)) ; = f
(displayln (is-dense?  S0)) ; = f
(displayln (is-dense?  D0)) ; = t

(displayln (to-dense  S0)) ; = 0
(displayln (to-dense  S2)) ; =  (-4 -8 3 1)
(displayln (to-dense  S4)) ; = '(-4 -8 3 1)

(displayln (is-sparse?  EMPTY)) ; = f
(displayln (is-sparse?  S0)) ; = f
(displayln (is-sparse?  D1)) ; = f
(displayln (is-sparse?  D2)) ; = f

(displayln (to-sparse  EMPTY)) ; = ()
(displayln (to-sparse  D2)) ; = ((4 1) (6 2) (2 3))
(displayln (to-sparse  S4)) ; = ((4 0) (3 1) (2 2) (4
```