Eddy Lin (ELL22) and Nathan Kim (NJK24)

Problem Set 5 Extra Credit

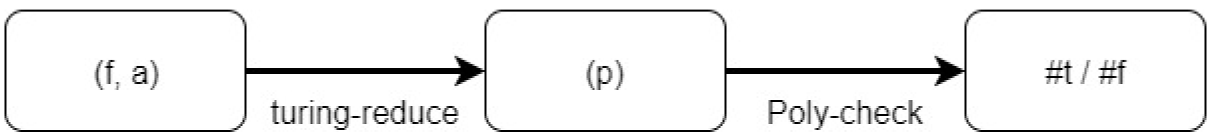
Problem 1

A procedure *Poly-check* that takes any argument *p* and infallibly returns whether or not *p* will correctly return all the roots of its argument *y* cannot possibly exist correctly, as we can show through a Turing reduction from the Halting Problem to Poly-check. To perform this Turing reduction, where (HP $\leq_T$ Y), we need to show that the halting problem would be determined if we could write Poly-check correctly through the following procedure definitions:

```
(define turing-reduce
  (lambda (f a)
    (list (lambda () (f a) p))))

(define safe?
  (lambda (f a)
    (apply poly-safe? (turing-reduce f a))))
```

This can be shown through the diagram:



Here, a function (f) and argument (a), are passed as arguments to turing-reduce. However, turing-reduce will only halt if (f a) successfully halts. If (f a) halts, turing-reduce will return another procedure *p* which DOES correctly return the roots of a function. For the sake of this proof, we have correctly implemented *p* (it has been written correctly; please refer to the Code Appendix at the end of this document). Then, Poly-check would return #t if and only if *p* is successfully returned by turing-reduce, which only happens if (f a) halts. Thus, safe? returns #t if (f a) halts, which would solve the Halting Problem, but that is impossible. Because the only assumption (besides *p* being correctly implemented) was that Poly-check existed, it must be incorrect. Thus, Poly-check cannot possibly exist and be correct for all procedures *p** that are passed as arguments in (Poly-check *p**).
QED.

Problem 2

A procedure *expr-zero?* that takes any argument *p* and infallibly returns whether or not *p* will correctly return all the roots of its argument *y* cannot possibly exist correctly, as we can show through a Turing reduction from the Halting Problem to Poly-check. To perform this Turing reduction, where (HP $\leq_T$ Y), we need to show that the halting problem would be determined if we could write Poly-check correctly through the following procedure definitions:

```
(define turing-reduce-2
  (lambda (f a)
    (list (eval (lambda () (f a) 0)))))

(define safe?-2
  (lambda (f a)
    (apply expr=zero? (turing-reduce-2 f a))))
```

Here, a function (f) and argument (a), are passed as arguments to turing-reduce-2. However, turing-reduce-2 will only halt if (f a) successfully halts. If (f a) halts, turing-reduce-2 will return an expression which DOES equal zero. Then, expr-zero? would return #t if and only if the expression equal to zero is successfully returned by turing-reduce-2, which only happens if (f a) halts. Thus, safe?-2 returns #t if (f a) halts, which would solve the Halting Problem, but that is impossible. Because the only assumption was that expr-zero? exists, it must be incorrect. Thus, expr-zero? cannot possibly exist and be correct for all expressions *expr** that are passed as arguments in (expr-zero? *expr**).
QED.

Code Appendix

```
(require racket/base)

(define turing-reduce
  (lambda (f a)
    (list (lambda () (f a) p))))

(define safe?
  (lambda (f a)
    (apply poly-safe? (turing-reduce f a))))

(define turing-reduce-2
  (lambda (f a)
    (list (eval (lambda () (f a) 0)))))

(define safe?-2
  (lambda (f a)
    (apply expr=zero? (turing-reduce-2 f a))))

(define factors
  (lambda (n)
    (letrec ((loop
              (lambda (try lst)
                (cond [(= try (- -1 n)) lst]
                      [(= try 0) (loop (- try 1) lst)]
                      [(zero? (remainder n try))
                       (loop (- try 1) (cons try lst))]
                      [else
                       (loop (- try 1) lst)]))))
      (loop n `()))))

(factors 10)
; ==> (1 2 5 10)
(factors 20)
; ==> (1 2 4 5 10 20)
(factors 12)
; ==> (1 2 3 4 6 12)
(factors 17)
; ==> (1 17)


; check-root takes in a list and checks if each one is a root
(define check-root
```

```
  (lambda (n coeffs try)
    (zero?
     (+ (expt try n)
        (apply
         +
         (foldr
          (lambda (elem init)
            (cons (* elem (expt try (length init))) init))
            `()
            coeffs))))))

(check-root 3 `(0 -7 6) -1)
; ==> #f
(check-root 3 `(0 -7 6) 0)
; ==> #f
(check-root 3 `(0 -7 6) 1)
; ==> #t
(check-root 3 `(0 -7 6) 2)
; ==> #t
(check-root 3 `(0 -7 6) (- 3))
; ==> #t

(define find-roots
  (lambda (n coeffs possible-roots)
    (foldr
     (lambda (root init)
       (if (check-root n coeffs root)
           (cons root init)
           init))
     `()
     possible-roots)))

(find-roots 3 `(0 -7 6) `(-3 -2 -1 0 1 2 3))
; ==> (-3 1 2)
```

```
(define p
  (lambda ((lst <list>))
    (let* [(n (car lst))
           (coeffs (last lst))
           (a-0 (last coeffs))
           (possible-roots (factors a-0))]
      (find-roots n coeffs possible-roots))))
```

```
(p (list 3 `(0 -7 6)))
; ==> (-3 1 2)
```