Structure and Interpretation of Computer Programs
Second Edition
Sample Problem Set

Math Set

A Generic Arithmetic Package

This problem set is based on Sections 2.5 and 2.6 of the notes, which discuss a generic arithmetic system that is capable of dealing with rational functions (quotients of polynomials). You should study and understand these sections and also carefully read and think about this handout before attempting to solve the assigned problems.

There is a larger amount of code for you to manage in this problem set than in previous ones. Furthermore, the code makes heavy use of data-directed techniques. We do not intend for you to study it all—and you may run out of time if you try. This problem set will give you an opportunity to acquire a key professional skill: mastering the code *organization* well enough to know what you need to understand and what you don't need to understand.

Be aware that in a few places, which will be explicitly noted below, this problem set modifies (for the better!) the organization of the generic arithmetic system described in the text.

Generic Arithmetic

The generic arithmetic system consists of a number of pieces. The complete code is attached at the end of the handout. All of this code will be loaded into Scheme when you load the files for this problem set. You will not need to edit any of it. Instead you will augment the system by adding procedures and installing them in the system.

Hand in your PreLab work, computer listings of all the procedures you write in lab, and transcripts showing that the required functionality was added to the system. The transcript should include enough tests to exercise the functionality of your modifications and to demonstrate that they work properly.

The basic generic arithmetic system

There are three kinds, or *subtypes*, of generic numbers in the system of this Problem Set: generic ordinary numbers, generic rational numbers, and generic polynomials. Elements of these subtypes are tagged items with one of the tags `number`, `rational`, or `polynomial`, followed by a data structure representing an element of the corresponding subtype. For example, a generic ordinary number has tag `number` and another part, called its *contents*, which represents an ordinary number.

We can summarize this in a type equation:

$$\text{Generic-Num} = (\{\texttt{number}\} \times \text{RepNum}) \cup (\{\texttt{rational}\} \times \text{RepRat}) \cup (\{\texttt{polynomial}\} \times \text{RepPoly}).$$

The type tagging mechanism is the simple one described on p. 165 of the text, and the `apply-generic` is the one *without coercions* described in section 2.5.3. The code for these is in `types.scm`.

We will also assume that the commands `put` and `get` are available to automagically update the table of methods around which the system is designed. You needn't be concerned in this problem set how `put` and `get` are implemented[1].

Some familiar arithmetic operations on generic numbers are

```
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

These are all of type (Generic-Num, Generic-Num) $\rightarrow$ Generic-Num. We also have

```
(define (negate x) (apply-generic 'negate x))
```

of type Generic-Num $\rightarrow$ Generic-Num, and

```
(define (=zero? x) (apply-generic '=zero? x))
```

of type Generic-Num $\rightarrow$ Sch-Bool.

Using these operations, compound generic operations can be defined, such as

```
(define (square x) (mul x x))
```

Packages

The code for the generic number system of this problem set has been organized in `ps5-code.scm` into groups of related definitions labelled as "packages." A package generally consists of all the procedures for handling a particular type of data, or for handling the interface between packages.

The packages described in the text are enclosed in a package installation procedure that sets up internal definitions of the procedures in the package. An example is `the-number-package` on p. 178. This ensures there will be no conflict if a procedure with the same name is used in another package, allowing packages to be developed separately with minimal coordination of naming conventions.

In this assignment it will be more convenient *not* to enclose the packages into internal definitions. Instead, the code is laid out textually in packages, but essentially everything is defined at "top level." You will see that we have therefore been careful to choose different names for corresponding procedures in different packages, e.g., `+` which adds in the number package and `+poly` which adds in the polynomial package.

Ordinary numbers

To install ordinary numbers, we must first decide how they are to be represented. Since Scheme already has an elaborate system for handling numbers, the most straightforward thing to do is to use it, namely, let

$$\text{RepNum} = \text{Sch-Num}.$$

[1]This will be explained when we come to section 3.3.3 of the Notes.

This allows us to define the methods that handle generic ordinary numbers simply by calling the Scheme primitives `+`, `-`, ..., as in section 2.6.1. So we can immediately define interface procedures between RepNum's and the Generic Number System:

```
(define (+number x y) (make-number (+ x y)))
(define (-number x y) (make-number (- x y)))
(define (*number x y) (make-number (* x y)))
(define (/number x y) (make-number (/ x y)))
```

These are of type (RepNum, RepNum) $\to$ ({`number`} $\times$ RepNum). Also,

```
(define (negate-number x) (make-number (- x)))
```

of type RepNum $\to$ ({`number`} $\times$ RepNum),

```
(define (=zero-number? x) (= x 0))
```

of type RepNum $\to$ Sch-Bool, and

```
(define (make-number x) (attach-tag 'number x))
```

of type RepNum $\to$ ({`number`} $\times$ RepNum).

All but the last of these procedures get installed in the table as methods for handling generic ordinary numbers:

```
(put 'add '(number number) +number)
(put 'sub '(number number) -number)
(put 'mul '(number number) *number)
(put 'div '(number number) /number)
(put 'negate '(number) negate-number)
(put '=zero? '(number) =zero-number?)
```

The number package should provide a means for a user to create generic ordinary numbers, so we include a user-interface procedure[2] of type Sch-Num $\to$ ({`number`} $\times$ RepNum), namely,

```
(define (create-number x) (attach-tag 'number x))
```

Exercise 5.1A The generic equality predicate

$$\texttt{equ?} : (\text{Generic-Num}, \text{Generic-Num}) \to \text{Sch-Bool}$$

is supposed to test equality of its arguments. Define an `=number` procedure in the Number Package suitable for installation as a method allowing generic `equ?` to handle generic ordinary numbers. Include the type of `=number` in comments accompanying your definition.

Lab exercise 5.1B Install `equ?` as an operator on numbers in the generic arithmetic package. Test that it works properly on generic ordinary numbers.

[2]In Exercise 2.76 in the text, the implementation of the type tagging system is modified to maintain the illusion that generic ordinary numbers have a `number` tag, without actually attaching the tag to Scheme numbers. This implementation has the advantage that generic ordinary numbers are represented exactly by Scheme numbers, so there is no need to provide the user-interface procedure `create-number`. Note that in Section 6 following Exercise 2.76, the text implicitly assumes that this revised implementation of tags has been installed. In this problem set we stick to the straightforward implementation with actual `number` tags.

Rational numbers

The second piece of the system is a Rational Number package like the one described in section 2.1.1. The difference is that the arithmetic operations used to combine numerators and denominators are *generic* operations, rather than the primitive `+`, `-`, etc. This difference is important, because it allows "rationals" whose numerators and denominators are arbitrary generic numbers, rather than only integers or ordinary numbers. The situation is like that in Section 2.6.3 in which the use of generic operations in `+terms` and `*terms` allowed manipulation of polynomials with arbitrary coefficients.

We begin by specifying the representation of rationals as *pairs* of Generic-Num's:

$$\text{RepRat} = \text{Generic-Num} \times \text{Generic-Num}$$

with constructor

```
(define (make-rat numerator denominator)
  (cons numerator denominator))
```

of type Generic-Num, Generic-Num $\to$ RepRat, and selectors

```
(define numer car)
(define denom cdr)
```

Note that `make-rat` does not reduce rationals to lowest terms as in Section 2.1.1, because `gcd` makes sense only in certain cases—such as when numerator and denominator are integers—but we are allowing arbitrary numerators and denominators.

Now we define basic procedures of type (RepRat, RepRat) $\to$ RepRat within the Rational Number package:

```
(define (+rat x y)
  (make-rat (add (mul (numer x) (denom y))
                 (mul (denom x) (numer y)))
            (mul (denom x) (denom y))))
(define (-rat x y)
  (make-rat (sub (mul (numer x) (denom y))
                 (mul (denom x) (numer y)))
            (mul (denom x) (denom y))))
(define (*rat x y)
  (make-rat (mul (numer x) (numer y))
            (mul (denom x) (denom y))))
(define (/rat x y)
  (make-rat (mul (numer x) (denom y))
            (mul (denom x) (numer y))))
```

There is also

```
(define (negate-rat x)
  (make-rat (negate (numer x))
            (denom x)))
```

of type RepRat $\to$ RepRat,

```
(define (=zero-rat? x)
  (=zero? (numer x)))
```

of type RepRat $\to$ Sch-Bool, and finally,

```
(define (make-rational x) (attach-tag 'rational x))
```

of type RepRat $\to$ ({`rational`} $\times$ RepRat).

Next, we provide the interface between the Rational package and the Generic Number System, namely the methods for handling rationals.

```
(define (+rational x y) (make-rational (+rat x y)))
(define (-rational x y) (make-rational (-rat x y)))
(define (*rational x y) (make-rational (*rat x y)))
(define (/rational x y) (make-rational (/rat x y)))
```

of type (RepRat, RepRat) $\to$ ({`rational`} $\times$ RepRat),

```
(define (negate-rational x) (make-rational (negate-rat x)))
```

of type RepRat $\to$ ({`rational`} $\times$ RepRat), and

```
(define (=zero-rational? x) (=zero-rat? x))
```

of type RepRat $\to$ Sch-Bool.

To install the rational methods in the generic operations table, we evaluate

```
(put 'add '(rational rational) +rational)
(put 'sub '(rational rational) -rational)
(put 'mul '(rational rational) *rational)
(put 'div '(rational rational) /rational)
(put 'negate '(rational) negate-rational)
(put '=zero? '(rational) =zero-rational?)
```

The Rational Package should also provide a means for a user to create Generic Rationals, so we include an external procedure of type (Generic-Num, Generic-Num) $\to$ ({`rational`} $\times$ RepRat), namely,

```
(define (create-rational x y)
   (make-rational (make-rat x y)))
```

Exercise 5.2 Produce expressions that define `r5/13` to be the rational number 5/13 and `r2` to be the rational number 2/1. Assume that the expression `(define rsq (square (add r5/13 r2)))` is evaluated. Draw a box and pointer diagram that represents `rsq`.

Exercise 5.3A Define a predicate `equ-rat?` inside the Rational package that tests whether two rationals are equal. What is its type?

Exercise 5.3B Install the relevant method in the generic arithmetic package so that `equ?` tests the equality of two generic rational numbers as well as two generic ordinary numbers. Test that `equ?` works properly on both subtypes of generic numbers.

Operations across Different Types At this point all the methods installed in our system require all operands to have the subtype—all `number`, or all `rational`. There are no methods installed for operations combining operands with distinct subtypes. For example,

```
(define n2 (create-number 2))
(equ? n2 r2)
```

will return a “no method” error message because there is no equality method at the subtypes (`number rational`). We have not built into the system any connection between the number 2 and the rational 2/1.

Some operations across distinct subtypes are straightforward. For example, to combine a rational with a number, n, coerce n into the rational $n/1$ and combine them as rationals.

Exercise 5.4A Define a procedure

$$\texttt{repnum->reprat} : \text{RepNum} \rightarrow \text{RepRat}$$

which coerces n into $n/1$.

Now, for any type, T, you can obtain a

$$(\text{RepNum}, \text{RepRat}) \rightarrow T$$

method from a

$$(\text{RepRat}, \text{RepRat}) \rightarrow T$$

method by applying the procedure `RRmethod->NRmethod`:

```
(define (RRmethod->NRmethod method)
  (lambda (num rat)
    (method
     (repnum->reprat num)
     rat)))
```

Use `RRmethod->NRmethod` to define methods for generic `add`, `sub`, `mul`, and `div` at argument types (`number rational`). Define methods for these operations at argument types (`rational number`). Also define `equ?` these argument types.

Exercise 5.4B Install your new methods. Test them on `(equ? n2 r2)` and

```
(equ? (sub (add n2 r5/13) r5/13) n2)
```

Polynomials

The Polynomial package defines methods for handling generic polynomials which are installed by

```
(put 'add '(polynomial polynomial) +polynomial)
(put 'mul '(polynomial polynomial) *polynomial)
(put '=zero? '(polynomial) =zero-polynomial?)
```

The package also includes an external procedure so the user can construct generic polynomials. Namely,

$$\texttt{create-polynomial} : (\text{Variable}, \text{List}(\text{Generic-Num})) \rightarrow (\{\texttt{polynomial}\} \times \text{RepPoly})$$

constructs generic polynomials from a variable and the list of coefficients starting at the high order term (this is the preferred representation for *dense* polynomials described in Section 2.6.3).

Within the Polynomial package, polynomials are represented by *abstract* term lists, using the list format preferred for *sparse* polynomials as described in Section 2.6.3. These abstract term lists are not necessarily Scheme lists, but have their own constructors and selectors. (They are, in fact, implemented as ordinary lists in `ps5-code.scm`, but the abstraction makes it easier to change to a possibly more efficient term list representation without changing code outside the Term List package.) So we have the type equations

$$\begin{aligned}\text{RepPoly} &= \text{Variable} \times \text{RepTerms}\\ \text{RepTerms} &= \text{Empty-Term-List} \cup (\text{RepTerm} \times \text{RepTerms})\\ \text{RepTerm} &= \text{Sch-NatNum} \times \text{Generic-Num}\end{aligned}$$

with term list constructors

$$\begin{aligned}\texttt{the-empty-termlist} &: \text{Empty-type} \to \text{RepTerms}\\ \texttt{adjoin-term} &: (\text{RepTerm}, \text{RepTerms}) \to \text{RepTerms},\end{aligned}$$

and selectors `first-term` and `rest-terms`[3].

In this problem set, we do modify the definition of `*-term-by-all-terms` given on p. 193 of the test. The new definition is:

```
(define (*-term-by-all-terms t1 tlist)
  (map-terms
   (lambda (term) (*term t1 term))
   tlist))

(define (*term t1 t2)
  (make-term
   (+ (order t1) (order t2))
   (mul (coeff t1) (coeff t2))))
```

Exercise 5.5A What is the type of the procedure `map-terms`? Supply its definition.

Exercise 5.5B Define a procedure `create-numerical-polynomial` which, given a variable name, `x`, and list of Sch-Num, returns a generic polynomial in `x` with the list as its coefficients.

Use `create-numerical-polynomial` to define `p1` to be the generic polynomial

$$p_1(x) = x^3 + 5x^2 + -2.$$

Exercise 5.5C Evaluate your definition of `map-terms`, thereby completing the definition of multiplication of generic polynomials. Use the generic `square` operator to compute the square of `p1`, and the square of its square. Turn in the the **pretty-printed** results of the squarings, as computed in lab.

[3]The Empty-type has no elements. The type statement

$$\texttt{make-an-element} : \text{Empty-type} \to T$$

indicates that the procedure `make-an-element` take no arguments, and evaluating `(make-an-element)` returns a value of type T. Such procedures are sometimes called "thunks." There wasn't any special need to use a thunk as constructor for empty term lists—a constant equal to the empty term list would have served as well (better? `:-)`)—but it serves as a reminder that term lists are created differently than Scheme's lists.

There are still few methods installed which work with operands of mixed types. This means that generic arithmetic on polynomials with generic coefficients of different types is likely to fail. For example, a representation of the polynomial $p_2(z,x) = p_1(x)z^2 + 3z + 5$ is defined in buffer `ps5-ans.scm` as:

```
(define p2-mixed
  (create-polynomial
   'z
   (list
    p1
    (create-number 3)
    (create-number 5))))
```

Now squaring `p2-mixed` will generate a "no method" error message, because there is no method for multiplying the numerical coefficients 3 and 5 by the polynomial coefficient `p1`. A definition which will work better in our system would be to replace 3 and 5 by the corresponding constant polynomials in `x`:

```
(define p2
  (create-polynomial
   'z
   (list
    p1
    (create-polynomial 'x (list (create-number 3))
    (create-polynomial 'x (list (create-number 5)))))))
```

Exercise 5.6A Use `create-rational` and `create-numerical-polynomial` to define the following rationals whose numerators and denominators are polynomials in `y`:

$$3/y,\ (y^2+1)/y,\ 1/(y+-1),\ 2$$

Then define a useful representation for $p_3(x,y) = (3/y)x^4 + ((y^2+1)/y)x^2 + (1/(y+-1))x + 2$.

Exercise 5.6B Use the generic `square` operator to compute the square of `p2` and `p3`, and the square of the square of `p2`. Turn in the definitions you typed to create `p3` and the **pretty-printed** results of the squarings, as computed in lab.

Completing the polynomial package

If you construct a chart of the dispatch table we have been building, you will see that there are some unfilled slots dealing with polynomials. Notably, the generic `negate` and `sub` operations do not know how to handle polynomials. (There is also no method for polynomial `div`, but this is more problematical since polynomials are not closed under division, e.g., dividing $x+1$ by x^2 yields a *rational function*

$$\frac{x+1}{x^2}$$

which is not equivalent to any polynomial.)

Exercise 5.7A Use the procedure `map-terms` to write a procedure `negate-terms` that negates all the terms of a term list. Then use `negate-terms` to define a procedure `negate-poly`, and a method `negate-polynomial`. Include the types in the comments accompanying your code.

Exercise 5.7B Using the `negate-poly` procedure you created in exercise 5.7A, and the procedure `+poly`, implement a polynomial subtraction procedure `-poly`, and a method `-polynomial`. Use `-poly` and `=zero-poly?` to implement `equ-poly?` and `equ-polynomial?`

Exercise 5.7C Install `negate-polynomial` in the table as the generic `negate` method for polynomials. Install `-polynomial` and `equ-polynomial?` as the generic `sub` and `equ?` operations on polynomials. Test your procedures on the polynomials `p1`, `p2`, and `p3` of exercises 5.5 and 5.6.

More Operations across Different Types

To combine a polynomial, p, with a number, we coerce the number into a constant polynomial over the variable of p, and combine them as polynomials.

Exercise 5.8A Define a procedure `repnum->reppoly` : (Variable, RepNum) $\rightarrow$ RepPoly which coerces n into a constant polynomial n over a given variable. Define methods for generic `add`, `sub`, `mul` and `equ?` at types (`number polynomial`) and (`polynomial number`), and for generic `div` at types (`polynomial number`).

Lab exercise 5.8B Install your new methods. Test them on `(square p2-mixed)` and

```
(equ? (sub (add p1 p3) p1) p3).
```

Exercise 5.8C To multiply a rational by a number, it was ok to coerce the number n into the rational $n/1$. Give an example illustrating why handling multiplication of a polynomial and a rational by coercing the polynomial p into the rational $p/1$ is not always a good thing to do. How about coercing the rational into a constant polynomial over the variable of p?

Polynomial Evaluation

Polynomials are generic numbers on the one hand, but on the other hand, they also describe *functions* which can be applied to generic numbers. For example, the polynomial $p_1(x) = x^3+5x^2-2$ evaluates to 26 when $x = 2$. Similarly, when $z = x + 1$, the polynomial $p_2(z, x)$ evaluates to the polynomial

$$x^5 + 7x^4 + 11x^3 + 3x^2 - x + 6.$$

It is easy to define an `apply-polynomial` procedure:

```
(define (apply-term t gn)
  (mul (coeff t)
       (power gn (order t))))

(define (power gn k)
  (if (< k 1)
      (create-number 1)
      (mul gn (power gn (dec 1)))))

(define (apply-terms terms gn)
  <**blob5.9A**>)

(define (apply-polynomial p gn)
  (apply-terms
   (term-list (contents p))
   gn))
```

Exercise 5.9A Fill in `<**blob5.9A**>` to complete the definition of `apply-terms`.

Exercise 5.9B Test your definition by applying p_1 to 2, p_2 to $x + 1$, and verifying

```
(define x (create-numerical-polynomial 'x '(1 0)))
(equ? (apply-polynomial p1 x) p1)
```