SICP section 2.1.4

July 27, 2007 at 07:52 Tags SICP

Exercise 2.7

Although in the definition of make-interval given by the authors it can't be seen explicitly, from the interval arithmetic functions presented it is clear that the lower bound is the first argument to make-interval. Therefore I prefer to give more explicit names to the arguments of make-interval:

```
(defun make-interval (lower upper)
  (cons lower upper))

(defun upper-bound (interval)
  (cdr interval))

(defun lower-bound (interval)
  (car interval))
```

Exercise 2.8

```
(defun sub-interval (x y)
  (make-interval
    (- (lower-bound x) (upper-bound y))
    (- (upper-bound x) (lower-bound y))))
```

Exercise 2.9

Suppose we're adding two intervals: `x1,y1` and `x2,y2`. If we shift both bounds of each interval by the same amount, keeping the width intact, it is obvious that the sum of the intervals will have the same width as the original – the sum of the widths. To prove it mathematically, let's represent an interval as only one bound and the width:

```
interval = (x, y)
Width: w = (y - x) / 2
=>
interval = (x, x + 2*w)
```

Adding two intervals:

```
(x_1, x_1 + 2*w_1) + (x_2, x_2 + 2*w_2)
=
(x_1 + x_2, x_1 + x_2 + 2*w_1 + 2*w_2)
```

The width of the sum is:

```
(y - x) / 2
=>
w_1 + w_2
```

Q.E.D. – the width of the sum is dependent only on the widths of the addends, not the addends themselves. The same applies to subtracting two intervals and can be proven similarly.

Now, to prove that this isn't true for multiplication, consider the interval `2,4` (width = 1). Multiplied by `7,10` (width = 1.5) we get `14,40` (width = 13). Now, if we multiply it by another interval with width 1.5, say `2,5` we get `4,20` (width = 8). Therefore, the width of the multiplication doesn't depend only on the widths of the multiplicands.

The sample example disproves the lemma for division as well: `2,4` divided by `7,10` is `0.2, 0.57` (width = 0.185), while divided by `2,5` is `0.4, 2` (width = 0.8).

Exercise 2.10

```
(defun div-interval (x y)
  (if (and
        (>= (upper-bound y) 0)
        (<= (lower-bound y) 0))
    (error "Denominator spans zero")
    (mul-interval
      x
      (make-interval  (/ 1.0 (upper-bound y))
                      (/ 1.0 (lower-bound y))))))
```

Exercise 2.11

To understand the "cryptic comment", let's first think about the original implementation of `mul-interval`. Here is is:

```
(defun mul-interval (x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (upper-bound y)))
        (p4 (* (upper-bound x) (lower-bound y))))
    (make-interval
      (min p1 p2 p3 p4)
      (max p1 p2 p3 p4))))
```

Why are so many computations needed, instead of something simple like the implementation of `add-interval` ? Because of negative bounds. Had we had an assurance that both bounds would be positive, `mul-interval` would be as simple as `add-interval`. But since some of the bounds can be negative, we should take many cases into consideration.

Now, Ben's suggestion is absolutely correct. If we figure out which bounds are positive and which are negative we can save some multiplications. For example, if both bounds of both intervals are positive, we only need two. For each interval, there are 3 cases: (1) both bounds negative, (2) both positive, (3) one negative and one positive. Since we multiply two intervals, we have 9 cases in total. Behold, this is not pretty:

```
(defun mul-interval (x y)
  (let ((upper-x (upper-bound x))
        (lower-x (lower-bound x))
        (upper-y (upper-bound y))
        (lower-y (lower-bound y)))
    (cond ( (and  (>= upper-x 0) (>= lower-x 0)     ; x++, y++
                  (>= upper-y 0) (>= lower-y 0))
              (make-interval (* lower-x lower-y) (* upper-x upper-y)))
          ( (and  (>= upper-x 0) (>= lower-x 0)     ; x++, y+-
                  (>= upper-y 0) (< lower-y 0))
              (make-interval (* upper-x lower-y) (* upper-x upper-y)))
          ( (and  (>= upper-x 0) (>= lower-x 0)     ; x++ y--
                  (< upper-y 0) (< lower-y 0))
              (make-interval (* upper-x lower-y) (* lower-x upper-y)))
          ( (and  (>= upper-x 0) (< lower-x 0)     ; x+- y++
                  (>= upper-y 0) (>= lower-y 0))
              (make-interval (* upper-y lower-x) (* upper-y upper-x)))
          ( (and  (>= upper-x 0) (< lower-x 0)     ; x+- y+-
                  (>= upper-y 0) (< lower-y 0))
              ; Here we must do all the possible multiplications,
              ; because we can't know what the worst case is
              ;
              (let ((p1 (* lower-x lower-y))
                    (p2 (* lower-x upper-y))
                    (p3 (* upper-x upper-y))
                    (p4 (* upper-x lower-y)))
                (make-interval
                  (min p1 p2 p3 p4)
                  (max p1 p2 p3 p4))))
          ( (and  (>= upper-x 0) (< lower-x 0)     ; x+- y--
                  (< upper-y 0) (< lower-y 0))
              (make-interval (* upper-x lower-y) (* lower-x lower-y)))
          ( (and  (< upper-x 0) (< lower-x 0)     ; x-- y++
                  (>= upper-y 0) (>= lower-y 0))
              (make-interval (* lower-x upper-y) (* lower-y upper-x)))
          ( (and  (< upper-x 0) (< lower-x 0)     ; x-- y+-
                  (>= upper-y 0) (< lower-y 0))
              (make-interval (* lower-x upper-y) (* lower-y lower-x)))
          ( (and  (< upper-x 0) (< lower-x 0)     ; x-- y--
                  (< upper-y 0) (< lower-y 0))
              (make-interval (* upper-x upper-y) (* lower-y lower-x)))

          (t (error "can't be")))))
```

Exercise 2.12

```
(defun make-center-percent (c p)
  (let ((w (abs (* p (/ c 100)))))
    (make-center-width c w)))

(defun percent (i)
  (* 100 (/ (width i) (abs (center i)))))
```

Exercise 2.13

Let's assume that the error of `X` is `dx`. Then:

```
Z + dz = (X + dx) * (Y + dy)
= X*dy + Y*dx + XY + dy*dx
=>
Z = XY
dz = X*dy + Y*dx + dy*dx
```

No surprises so far. But recall the assumption that `dx << X` and `dy << Y`. So we can drop the `dy*dx` term:

```
Z = XY
dz = X*dy + Y*dx
```

In percentages:

```
dz/Z = (X*dy + Y*dx) / XY
=>
dz/Z = dy/Y + dx/X
```

This is the tolerance (in percentages) of the result in terms of the tolerances of the factors.

Exercise 2.14

First, showing that `par1` and `par2` indeed result different results:

```
(defun par1 (r1 r2)
  (div-interval
    (mul-interval r1 r2)
    (add-interval r1 r2)))

(defun par2 (r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval
      one
      (add-interval (div-interval one r1)
                    (div-interval one r2)))))

(defvar aa (make-center-width 5 0.1))
(defvar bb (make-center-width 10 0.1))

(print (par1 aa bb))
=> (3.191447 . 3.4804058)
(print (par2 aa bb))
=> (3.2777026 . 3.3888159)
```

The reason why this is so isn't immediately obvious, but it becomes quite clear once we consider the two alternative formulas for parallel resistors carefully. Let's take this formula:

```
1 / (1/R1 + 1/R2)
```

And see how we reach the other formula from it. If we take the denominator, we can multiply `1/R1` by `R2/R2` without changing the formula, because `R2/R2` is 1 (A). Then we can multiply `1/R2` by `R1/R1` using the same reasoning. So we get:

```
1 / (R2/R1*R2 + R1/R1*R2) =>
1 / ((R2 + R1)/R1*R2) =>
R1*R2 / (R1 + R2)
```

Viola! We've reached the second formula. So indeed they are algebraically equivalent. However, for intervals this transformation doesn't work. Look above at the statement marked by (A). It says that `R2/R2 = 1`. But is it, when we talk about intervals ?

```
(defvar aa (make-center-width 5 0.1))
(print (div-interval aa aa))
=>
(0.9607844 . 1.0408163)
```

Of course not. After all, it's just two intervals divided one by another, and all the rules apply. No wonder Lem is getting different answers – he is using two different formulas!

Exercises 2.15 & 2.16

This is a combined answer to both exercises:

Eva Lu Ator is right, and an example can be seen in the computation done in the previous exercise – `par2` produces tighter bounds than `par1`. To understand why this is so, I will try to explain the problems with interval arithmetic (answer to 2.16):

In doing arithmetic, we rely on some laws to hold without giving them much thought. Speaking mathematically, the real numbers are fields. For example, we expect to have an inverse element for addition – for each element A to have an element A' so that A + A' = 0. It is easy to check that this doesn't hold for intervals! An inverse element for multiplication also doesn't exist (this is the problem we saw in exercise 2.14). The distributive law doesn't hold – consider the expression `[1,2] * ([-3,-2] + [3,4])` – it makes a difference whether you do the additions or the multiplication first.

To solve these problems at least for the simple arithmetic package we're developing, I think we need to define the concept of identity for an interval. Operations must be able to identify if two operators are identical, and adjust the results accordingly.

For comments, please send me an email.

Back to top