Lab3 - 大数运算实验

1. 实验要求

实现n位二进制大整数的加法运算。输入a, b和输出s都是二进制位的串。要求算法的时间复杂度满足$work = O(n), span = O(logn)$。

2. 实验思路

2.1 加运算求解思路

1. 检测0+0=0的情况，此时需返回空串而非单元素序列。
2. 通过map2将两个bit序列不进位相加，转化为carry序列：由于产生STOP与GEN均代表该位的bit和为0，产生PROP代表bit和为1，保存这个表示不进位和的中间序列。
3. 根据中间序列的carry情况，判断每一位是否需进位：如果为GEN则该位一定由进位导致下一位+1，为STOP则一定不进位，为PROP时情况与上一位相同。易知此操作具有结合律。
4. 保存3中所述进位序列（直接由GEN与STOP保存），通过scan实现错位保存（每个位置的scan结果由其之前序列的情况决定），即得到进位序列。
5. 通过map2将进位序列与中间序列合并，显然当某位进位序列为ZERO时结果为中间序列该位所代表的bit和，为GEN时取反。
6. 根据scan的最后一个返回值判断最高位是否进位，并返回最终结果。

2.2 减运算求解思路

1. 将两数长度补为一致（高位补0）,并补上y的符号位0。
2. 用tabulate将减数y按位翻转（0-1互换），所得结果加1得到含符号位的-y。
3. -y与x相加，舍去高于x位数的符号位，得到非负结果。
4. 判断所得结果是否为0，若是则返回空串，否则直接返回该结果。

2.3 乘运算求解思路

1. 处理乘数长度为0的情况，注意此时需返回空串而非单元素序列。
2. 处理乘数长度为1的情况，若为0则返回空串，为1则无需计算直接返回另一乘数。
3. 其他情况：
    1. 先将乘数分别平均切成比例相同的两段，令$x = (p * 2^m + q), y = (r * 2^m + s)$
    2. 计算如下:

$$x * y = (p * 2^m + q) * (r * 2^m + s) = pr * 2^{2m} + (pq + rs) * 2^m + qs$$

$$\because pq + rs = (p + q) * (r + s) - pr - qs$$

$$\therefore x * y = pr * 2^{2m} + ((p + q) * (r + s) - pr - qs) * 2^m + qs \tag{1}$$

    3. 并行递归，分别计算出$(p + q) * (r + s), p * r, q * s$
    4. 由(1)式，可根据三个递归结果计算出$x * y$

3. 回答问题

3.1 提供加法计算的代码和注释

- *Task 4.1 (35%)*. Implement the addition function `++ : bignum * bignum -> bignum` in the functor `MkBigNumAdd` in `MkBigNumAdd.sml`. For full credit, on input with m and n bits, yoursolution must have $\mathcal{O}(m + n)$ work and $\mathcal{O}(\lg{(m + n)})$ span. Our solution has under 40 lines with comments.

- **Answer 4.1**

```
fun x ++ y =
case (length x, length y)
  (* notice that given 0 ++ 0, we need to return empty() instead of singleton(ZERO) *)
  of (0, 0) => empty()
   | (_, _) =>
    let
      (* add two bignum without carry to generate the mid(raw) seq *)
      (* W = S = O(1) *)
      fun bitaddtocarry (x : bit, y : bit) : carry =
        case (x, y)
          of (ZERO, ZERO) => STOP
          | (ONE, ONE) => GEN
          | ((ONE, ZERO) | (ZERO, ONE)) => PROP

      val diff = length x - length y

      (* in this seq, STOP/GEN means ZERO, PROP means ONE *)
      (* call bitaddtocarry max{m, n} times. W = O(m+n), S = O(1) *)
      val rawseq = if diff > 0 then map2 bitaddtocarry x (append(y, tabulate (fn _ =>
              else if diff < 0 then map2 bitaddtocarry y (append(x, tabulate (fn _ =>
              else map2 bitaddtocarry x y

      (* If the last is STOP/GEN, it must gives ZERO/ONE for the next one.
        * But if it's PROP, the carry status will remain as the previous one is.
        * W = S = O(1)
        *)
      fun bitcarry (c1, c2) : carry =
        case c2
          of PROP => c1
          | (GEN | STOP) => c2

      (* in this seq, GEN/STOP means whether to add a ONE or not, determined by previc
      (* since length = max{m, n} = O(m+n), scan costs W = O(m+n), S = O(lg(m+n)) *)
      val (carryseq, most) = scan bitcarry STOP rawseq

      (*
        * rawseq saves the infomation of 0/1
        * carryseq saves the infomation of whether to carry
        * use scan to transplace forwards for one bit.
        * STOP means this bit is what saved in the rawseq.
        * GEN means this bit need to be flipped from what is in the rawseq.
        *)
      (* W = S = O(1) *)
      fun bitgrow (c1 : carry, c2 : carry) : bit =
        case (c1, c2)
          of (c1, GEN) => if c1 = PROP then ZERO else ONE
           | (c1, STOP) => if c1 = PROP then ONE else ZERO

      (* map2 has such cost: W = O(m+n) * O(1) = O(m+n), S = O(1) *)
      val resseq = map2 bitgrow rawseq carryseq
    in
      (* check whether to carry ONE to the most place. W = O(m+n), S = O(1) *)
      if most = GEN then append(resseq, singleton(ONE)) else resseq
      (* final costs are:
        * Work = O(m+n) + O(1) = O(m+n)
        * Span = O(lg(m+n)) + O(1) = O(lg(m+n))
        *)
    end
```

3.2 提供减法计算的代码和注释

- *Task 4.2 (15%)*. Implement the subtraction function `-- : bignum * bignum -> bignum` in the functor MkBigNumSubtract in MkBigNumSubtract.sml, where $x -- y$ computes the number obtained by subtracting y

from x. We will assume that $x \geq y$; that is, the resulting number will alwaysbe non-negative. You should also assume for this problem that $++$ has been implemented correctly. Forfull credit, if x has n bits, your solution must have $\mathcal{O}(n)$ work and $\mathcal{O}(\lg(n))$ span. Our solution has fewerthan 20 lines with comments.

- **Answer 4.2**

```
fun x -- y =
let
  (* A simple function to reverse ONE and ZERO, with W = S = O(1) *)
  fun reversebit (b : bit) : bit = if b = ONE then ZERO else ONE
  (* save the difference of lengths, W = S = O(1) *)
  val diff = length x - length y
  (* flip y, with setting it to the same length as x. ZERO on sign bit means positive,
  (* append n elements altogether, W = O(n), S = O(1) *)
  val reversed = append (tabulate (fn i => reversebit (nth y i)) (length y), tabulate
  (* by adding a ONE, we get the negative y, with W = W_++ = O(n+1), S = S_++ = O(lg(r
  val negated = reversed ++ singleton(ONE)
in
  (* add x and negative y. throw the sign bit *)
  (* notice the assumption `x > y`, so that we won't worry about getting 0 as a result
  (* add between length n and length n+1, with W = O(2n+1), S = O(lg(2n+1)) *)
  tabulate (fn i => nth (x ++ negated) i) (length x)
  (* finally we get:
    * Work = O(n) + O(n+1) + O(2n+1) + O(1) = O(n)
    * Span = O(lg(2n+1)) + O(lg(n+1)) + O(1) = O(lgn)
    *)
end
```

3.3 提供乘法计算的代码和注释

- *Task 4.3 (30%)*. Implement the function `** : bignum * bignum -> bignum` in MkBigNumMultiply.sml. For full credit, if the larger number has n bits, your solution must satisfy $W_{**}(n) = 3W_{**}(\frac{n}{2}) + \mathcal{O}(n)$ and have $\mathcal{O}(\lg^2 n)$ span. You should use the following function in the Primitives structure: `val par3 : (unit -> 'a) * (unit -> 'b) * (unit -> 'c) -> 'a * 'b * 'c` to indicate three-way parallelism in your implementation of $**$. You should assume for this problem that $++$ and $--$ have been implemented correctly, and meet their work and span requirements. Our solutionhas 40 lines with comments.

- **Answer 4.3**

```
fun x ** y =
case (length x, length y)
  (* notice that we need to return empty() when result = 0. *)
  of (0, _) => empty ()
   | (_, 0) => empty ()
   | (1, _) => if nth x 0 = ZERO then empty() else y
   | (_, 1) => if nth y 0 = ZERO then empty() else x
   | (_, _) =>
    let
      (* judge which one is larger, then setting them to the same length. S = O(1). *)
      val diff = length x - length y
      val (larger, smaller) =
        if diff > 0 then (x, append(y, tabulate (fn _ => ZERO) diff))
        else if diff < 0 then (y, append(x, tabulate (fn _ => ZERO) (~diff)))
        else (x, y)

      (* showt them into p, q, r, s, with larger = (p*2^m + q), smaller = (r*2^m+s). S
      val (q, p) =
        case showt larger
          of EMPTY => (empty (), empty())
           | ELT bit0 => if bit0 = ZERO then (singleton ZERO, empty()) else (singletor
           | NODE (l, r) => (l, r)
```

```
    val (s, r) =
      case showt smaller
        of EMPTY => (empty (), empty())
         | ELT bit0 => if bit0 = ZERO then (singleton ZERO, empty()) else (singletor
         | NODE (l, r) => (l, r)

    (*
     * @res_a = (p+q) * (r+s)
     * @res_b = p*r
     * @res_c = q*s
     * calculate them in parallel costs W = 3W(n/2), S = S(n/2).
     * and we have: pq+rs = (@res_a-@res_b-@res_c)
     *)
    val (res_a, res_b, res_c) = par3 (fn _ => (p++q) ** (r++s), fn _ => p**r, fn _ =
  in
    (*  x*y = (p*2^m + q)*(r*2^m+s) = p*r*2^(2m) + (pq+rs)*2^m + qs = @res_b*2^(2m)
     * obviously m = length q, so the following sentence can calculate x*y correctl
     * notice that the square of a number has the length shorter than three times c
     * append with length q costs O(n), add those `shorter than 3*(length n)` thinc
     * S = O(1).
     *)
    (append(tabulate (fn _ => ZERO) (length q * 2), res_b) ++ append(tabulate (fn _
    (*  finally we have:
     * W(n) = 3W(n/2) + O(n).
     * S(n) = S(n/2) + O(lgn), which means S(n) = lgn*O(lgn) = O((lgn)^2)
     *)
  end
```

3.4 迭代计算复杂度分析

- *Task 5.1 (15%)*. Determine the complexity of the following recurrences. Give tight $\Theta - bounds$, and justify your steps to argue that your bound is correct. Recall that $f \in \Theta(g)$ if and only if $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$. You may use any method (brick method, tree method, or substitution) to show that yourbound is correct, except that you must use the substitution method for problem 3.

  1. $T(n) = 3T(\frac{n}{2}) + \Theta(n)$
  2. $T(n) = 2T(\frac{n}{4}) + \Theta(\sqrt{n})$
  3. $T(n) = 4T(\frac{n}{4}) + \Theta(\sqrt{n})$ (Prove by substitution.)

- **Answer 5.1**

  1. $T(n) = \Theta(n^{\log_2 3})$. Using Tree Method.
     1. Obviously the height h can be calculated from $(\frac{1}{2})^h * n = 1$, then we have:

     $$h = \log_2(n) \tag{1}$$

     2. Each floor i has $3^i * \Theta(\frac{n}{2^i})$, and the sum is shown in the following.

     $$T(n) = \sum_{i=0}^{h}(3^i * \Theta(\frac{n}{2^i})) = \sum_{i=0}^{h}(\Theta(n * (\frac{3}{2})^i)) = \Theta(n * \sum_{i=0}^{h}(\frac{3}{2})^i) \tag{2}$$

     3. With (1) and (2), we can get the result:

     $$T(n) = \Theta(n * \frac{1 * (1 - (\frac{3}{2})^{h-1})}{1 - \frac{3}{2}}) = \Theta(n * 2 * ((\frac{3}{2})^{h-1} - 1))$$

     $$\therefore T(n) = \Theta(n * (\frac{3}{2})^{\log_2 n}) = \Theta(n * \frac{3^{\log_2 n}}{2^{\log_2 n}}) = \Theta(3^{log2 n})$$

     $$\therefore T(n) = \Theta(3^{\frac{log3 n}{log3 2}}) = \Theta(n^{\frac{1}{\log_3 2}}) = \Theta(n^{\log_2 3})$$

  2. $T(n) = \sqrt{n} * \log_4 n$. Using Tree Method.

1. Similar to the last problem, we have the height.

$$h = \log_4 n \tag{3}$$

2. Each floor i has $2^i * \Theta(\sqrt{\frac{n}{4^i}}) = \Theta(\sqrt{n})$, sum up all floors:

$$T(n) = \sum_{i=0}^{h} \Theta(\sqrt{n}) \tag{4}$$

3. With (1) and (2), we can get the result:

$$T(n) = \Theta(\sqrt{n} * \log_4 n)$$

3. $T(n) = \Theta(n)$. Using Substitution method.
   1. we need to prove that given a constant k_1, there exists a variable k_{2_i}, while $n \leq T(n) \leq k_1 n + k_{2_n} \sqrt{n}$, $k_{2_n} \leq \sqrt{n} - 1$
   2. (Base case)When $n = 1$, we have:

$$\exists k_{2_1} \leq 0, 1 \leq T(1) = 1 \leq k_1 * 1 + k_{2_1} * 1 \tag{5}$$

   3. (Induction)Assume whenever $n \leq \frac{k}{4}$, we have:

$$\exists k_{2_n} \leq \sqrt{n} - 1, n \leq W(n) \leq k_1 * n + k_{2_n} * \sqrt{n} \tag{6}$$

   4. Which means:

$$\forall i <= \frac{k}{2}, \exists k_{2_i} \leq \sqrt{i} - 1, i \leq W(i) \leq k_1 * i + k_{2_i} * \sqrt{i}, \tag{7}$$

$$\therefore \exists k_{2_{\frac{k}{4}}} \leq \frac{\sqrt{k}}{2} - 1, \frac{k}{4} \leq W(\frac{k}{4}) \leq k_1 * \frac{k}{4} + k_{2_{\frac{k}{4}}} * \frac{\sqrt{k}}{2}, \tag{8}$$

   5. Then:

$$\because W(k) = 4W(\frac{k}{4}) + \sqrt{k} * \Theta(1), \because (8)$$

$$\therefore 4 * \frac{k}{4} \leq W(k) \leq 4 * k_1 * \frac{k}{4} + 4 * k_{2_{\frac{k}{4}}} * \frac{\sqrt{k}}{2} + \sqrt{k}$$

$$\therefore k \leq W(k) \leq k_1 * k + (\sqrt{k} - 1) * \sqrt{k}$$

$$\therefore \exists k_{2_k} = \sqrt{k} - 1, k \leq W(k) \leq k_1 * k + k_{2_k} * \sqrt{k} \tag{9}$$

   6. With Induction, we proved:

$$\exists k_{2_n} = \sqrt{n} - 1, n \leq W(n) \leq k_1 * n + k_{2_n} * \sqrt{n} \tag{10}$$

$$\because n \leq W(n)$$

$$\therefore n \in \mathcal{O}(W(n)) \tag{11}$$

$$\because W(n) \leq k_1 * n + k_{2_n} * \sqrt{n} \therefore W(n) \in \mathcal{O}(2n - \sqrt{n})$$

$$\therefore W(n) \in \mathcal{O}(n) \tag{12}$$

$$\because (11), (12)$$

$$\therefore W(n) = \Theta(n) \cdots\cdots \square$$