OVERVIEW

大家好，我叫寿晨宸，今天由我来做 OVERVIEW 的回课。 第一节 OVERVIEW 的大班课上，老师们高屋建瓴地提了五个问题，指出了我们在学习这门课程时应该注意和思考的重点。接下来我就围绕这几个问题和相应的例子，谈谈我自已的延伸思考。当然，由于本人才疏学浅，所拓展的内容多局限于第二章的范畴内。请各位同学见谅。

整型的溢出问题

> 例1.1：
>
> $$x^2 >= 0$$
>
> 永远成立吗?

课上讲到，如果x是整型，则命题不成立，因为整型的表达范围有限，两个整型相乘会导致**溢出**。那么什么行为会导致溢出呢？或者说，如何判断溢出呢?

> 下文皆以**补码**为例

- **加法**：
  - 首先，数 x，y 需要都处于补码的表示范围之内
  - 如 x，y 异号，则 s=x+y 永远不会溢出
  - 如 x，y 同号，当且仅当 s=0 或 s 相对于 x，y 变号时，s=x+y 溢出。
- **减法**

  与整数减法的定义类比，补码减法的实现方式是符合直觉的，即

  ```
  x-y==x+(-y);
  ```

其中

```
(-y)==~y+1;
```

所以说，在判断 x-y 是否溢出时，首先要判断 `~y+1`是否溢出，然后就可以照抄上文判断加法是否溢出的方法了。

- **乘法**

$$s = x \times y$$

- 当 y 较小（譬如 y<4）时，我们可以将 s 看作是多个 x 的累加，再以判断加法是否溢出的方式去判断 s 是否溢出。
- 然而，当 y 较大时，这种办法毕竟太过于愚蠢了。所以书上的练习题 2.35 给出一种简洁的方法（只不过很遗憾不能在lab中使用），具体实现如下，证明方式不再赘述

```
int tmult_ok(int x,int y)
{
    int p=x*y;
    /* x=0 或 (x*y)/x==y */
    return !x || p/x == y;
}
```

无符号数，补码和整数的运算法则

> 例1.2：是否满足结合律
>
> $$(x+y)+z = x+(y+z)$$

是否满足交换律、结合律、分配律？是否有可逆元和单位元？这些问题错综复杂，但归根结底，考察的是集合以及定义在集合上运算的性质，即代数结构。 说得通俗点，理想中的整数集是环，那么无符号数和补码又是什么？群，环，还是域？

> 简单地介绍一下群环域的基本概念。
>
> - 群表示一个拥有满足封闭性、满足结合律、有单位元、有逆元的二元运算的代数结构。

> - 环是一种代数结构, 指带有相容的加法与乘法运算的集合。加法满足结合律、交换律、有单位元 (称为环的零元, 记为 0)、有逆元 (称为相反元)。 乘法满足结合律, 且有单位元 (称为环的单位元或幺元, 记为 1). 我们一般将 a·b 简记为 ab。乘法对加法满足分配律。
> - 域是一种代数结构, 它是具备加、减、乘、除四则运算的集合, 并满足交换律、结合律、分配律。即在环的基础上，对除法封闭。

如果将整型上的加法定义为整数集上的加法，那么整型甚至无法满足成为群的条件，因为它对于加法来说显然是不封闭的。所以实际上，整型是模 k 同余关系的商集，其中，

$$k = 2^w$$

无符号数上的加法定义为：

$x+_w^uy=(x+y)\hspace{0.5em}mod\hspace{0.5em}2^w$

而补码上的加法定义为：

$x+_w^ty=U2T_w((x+y)\hspace{0.5em}mod\hspace{0.5em}2^w)$

在此定义下，整型满足以下性质：

- 交换律
- 结合律
- 分配律

为什么呢？首先，无符号数上的加法显然是满足这些性质的；其次，因为映射 U2T 是双射，而补码上的加法是无符号数上的加法和映射 U2T 的复合，所以补码上的加法与无符号数上的加法一样拥有这些性质。

同时，存在零元 0，使

$$x + 0 = 0 + x = x$$

对于无符号数而言，每一个元素都存在唯一的逆元

$$(-x) = 2^w - x$$

使

$(-x)+x=(2^w-x+x)\hspace{0.5em}mod\hspace{0.5em} 2^w=0$

对于补码而言，

```
0x80000000+0x80000000==0
```

即 Tmin 的逆元是其本身，而其余数的逆元也唯一，为 `(-x)=~x+1`。也就是说，整型的每一个元素都存在唯一的逆元。

综上所述，整型与其上的加法构成的数集为阿贝尔群。所以说尽管计算机中数据类型所能占用的资源有限，但整型仍然拥有相当良好的性质，甚至可以说是最良好的性质了。因为整型必然为有限集，有限集想成为域，其模的数 k 只能是素数，也就是说，k 只能等于 2，整型只有在成为布尔型时，才能成为数域，而这无疑是不现实的。

负数的位表示方式

由例1.2，我们可以延伸出关于整型设计的问题，那么我们不妨接着来探讨一下整型中负数的位表示方式。 理论上来讲，只要负数的位表示和其值之间的映射为双射，那么使用什么位表示方式都是无所谓的。那么为什么要将补码作为负数的表出方式呢？如果将 -8 表示为 10001000 不是更符合直觉吗？ 让我们来举一个例子，16+(-8)=?，16=00010000 使用 10001000 来表示 -8 的话，我们有 16+(-8)=00010000+10001000=10011000=-24，这显然是不对的，也就是说，在这种位表示方式下，机器需要两套不同的加法实现方式——同号数之间的和异号数之间的。 如果使用补码，也就是 11111000 来表示 -8 的话，有 16+(-8)=00010000+11111000=00001000(假设该数只有8位)=8 这是正确的，也就是说对于补码而言，异号数和同号数的加法实现是统一的。

整型乘法与移位操作

> 例2.2:为何 fun2 的效率更高？(我对课上的例子稍做了改动，让我们暂且悬置存储器的问题)
>
> ```
> void fun1(int *x,int *y)
> {
>   *x+=*y;
>   *x+=*y;
>   *x+=*y;
>   *x+=*y;
> }
> void fun2(int *x,int *y)
> {
>   *x+=4*(*y);
> }
> ```

课上给出的答案是，fun1 比 fun2 做了更多次数的存储器引用，但是有没有一种可能，乘法实现方式的差异也是影响因素？ 经过改动后，fun1 和 fun2 之间的差异就更加明显了，其中 fun1 共做了4次加法运算，而 fun2 做了1次乘法运算和1次加法运算，而这次乘法运算，可以视作一次移位操作。也就是说，fun1 比 fun2 多花了一倍的时间在运算上。 然而，事情真的那么简单吗？计算机是怎么判断要用哪些移位操作来实现乘法的呢，总不可能是打表吧？它实现这种替代的开销会不会反而比计算的开销大？加上这种开销，用移位替代乘法真的是一种省时的做法吗？ 众所周知，任何数都可以通过拆成2的幂次的和，也就是说，任何数与常数的乘法都可以用移位和加减法替代，譬如 `x*9=(x<<3)+x`，`x*14=(x<<4)-(x<<1)`。 那么，怎么拆更快呢？ 事实上，并不需要做任何逻辑判断，甚至不需要拆数，只要做简单的竖式乘法就行了。 我们将一般的竖式乘法称作 Long multiplication，特殊地，将二进制的竖式乘法称作 Shift-and-add algorithm。使用该算法的情况下，需要 O(n^2) 个单位运算，其中n为二进制数的位数。（什么鬼，根本不快啊！） 举个例子，`8*9=8*(2^3+2^0)=8*1+(8<<1)*0+(8<<2)*0+(8<<3)*0=72`

```
      1000
      1001
      ————
      1000     ->1*8
     0000      ->0*(8<<1)
    0000       ->0*(8<<2)
   1000        ->1*(8<<3)
   ———————
   1001000
```

我们在竖式运算中不知不觉地做了拆分，只用移位和相加就实现了乘法。

但是，说到底，这种算法是远远称不上快的。

- 如 Karatsuba 算法，时间复杂度为

$$O(n^{log_2 3}) \approx O(n^{1.58})$$

- 如 Toom–Cook-k 算法，时间复杂度为

$$O(n^{\frac{log(2k-1)}{log(k)}})$$

- 如 Schönhage-Strassen 算法，时间复杂度为

$$O(nlog(n)log(log(n)))$$

- 还有 Joris van der Hoeven 和 David Harvey，他们宣称发现了已知最快的乘法算法，时间复杂度为

$$O(nlog(n))$$

那为什么不使用这些算法去取代 Shift-and-add 算法实现二进制乘法器呢?

我暂时没有找到明确的答案，以下都是我个人未经检验的揣测。

其一是因为以上这些算法的逻辑过于复杂，在用电路实现算法时反倒不如传统的二进制乘法器简单；其二是因为在数据位数较小时，传统的算法与快速乘法算法之间并没有显著的效率差距，甚至传统算法要更优一筹。其三，硬件进步，算力已经强大到无需纠结乘法算法的优劣，直接沿着技术传统走下去就好了。

当然，最后也是私以为最有可能的一条理由，那就是这些炫目的算法已经实现于我们计算机的底层逻辑中了，只是我孤陋寡闻，不曾听闻罢了。

以上。（谢谢大家）