内部资料

美国计算机科学家施瓦茨夫人

（Frances E. Allen）

学术报告记录

一九七三年八月

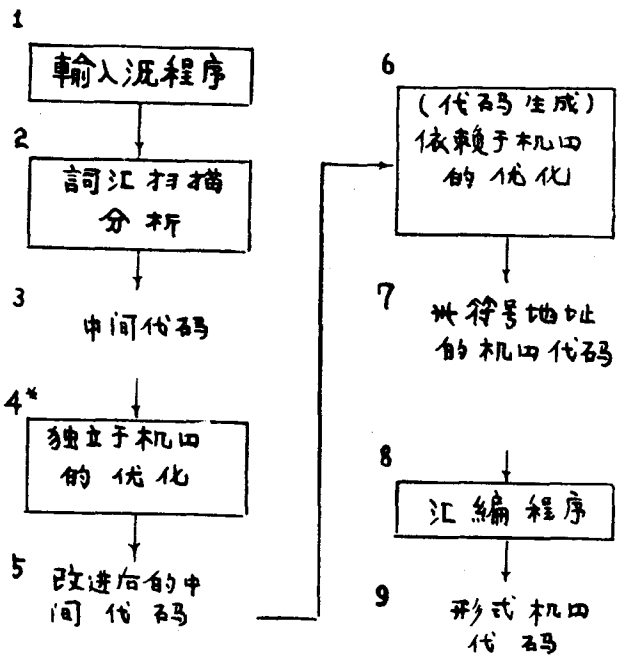
施瓦茨夫人（Frances, E. Allen）报告（一）

时间：一九七三年七月七日上午八时——十一时半；

一九七三年七月九日上午八时——十一时半。

这两次讲优化问题。先讲优化需要的基础。举些例子说明优化过程。然后再详细谈谈优化及优化算法。

一个有优化的编译系统如下图：



重点讲打*的部分，还介绍一下中间编码的最好形式。依赖机器的优化就不多谈了。汇编程序典型的是用两遍加工，很容易，可变成一遍加工。

中间语言的内部形式亦可用符号的，但较化钱。

独立于机器的优化，依赖于输入源语言的类型，而依赖于机器的优化，主要和机器有关。这部分主要是寄存器分配，所以和寄存器的多少有关，而有的机器不用寄存器而用栈，则优化就不同。

二、例子：下面用例子来说明图 1 中的 3 、4 、5 。

设源程序为：

```
REAL  A(10, 10), B(10, 10)
FOR  I=1, 10  DO
BEGIN  A(I, 2*J)=B(I, 2*J)
END
```

对此例如进行简单的翻译，则得下面的代码：

```
      I= 1
label: t1=2*J
      t2=10*I+t1
      Load B(t2) into R1
      t3=2*J
      t4=10*I+t3
      Store R1 into A(t4)
      I=I+1
      IF I<10 go to label
```

一般都这样翻译。（上面忽略了 -些地址常数，因为这是不重要的）。

对此例可作不少优化：

1． 多余子表达式的消除（或公共子表达式消除）

看上例，其中的2*J不需再计算，可消除，而在下面t_3处代之以t_1。t_4表达式亦可消除用t_2代t_4，这就是消除多余子表达式。

再看2＊J在循环中不变。优化主要想运行速度高，所以想把循环内的计算往外提。即代码外提（Code motion）。

2. 代码外提

有两种：向上提和向下提。但向下提2＊J就必须把Store R_1 into $A(t_4)$先往下提，这是很困难的。而不变子表达式向上提较容易。所以通常是考虑不变子表达式向上提，即代码外提。

上例在代码外提后，得下列代码：

```
        I = 1
        t1 = 2＊J
label:  t2 = 10＊I + t1
        Load B(t2) into R1
        Store R1 into A(t2)
        I = I + 1
        IF I⊿10 go to label
```

经过这两步优化，程序已大大改进了。

3. 强度削减

分析上例：t_2每循环一次改变10。如代替乘法以加法，上例改为

```
        I = 1
        t1 = 2＊J
        t2 = 10＊I + t1
label:  Load B(t2) into R1
        Store R1 into A(t2)
        I = I + 1
        t2 = t2 + 10
        IF I⊿10 go to label
```

这里用加法代替了乘法，这就叫強度削减。这步较复杂，下次再介绍一个作強度消减的通用方法。这在“Programming optimization”一文中已谈到。

这样作是为了适应变址寄存器的用法，在循环头上给初值，末尾给增量。

4. 常数传递

例如上例经过強度消减后的$t_2 = 10 * I + t_1$中的I实际上可用1来代替。

5. 检查代替

至此，对I的检查，实际上可用对t_2的检查来代替。

6. 无用代码的消除

经过上述各步的优化，最终变成：

$t_1 = 2 * J$

$t_2 = 10 + t_1$

$t_3 = 100 + t_1$

label: Load $B(t_2)$ into R1

Store R_1 into $A(t_2)$

$t_2 = t_2 + 10$

IF $t_2 \angle t_3$ go to label

下面我们来比较一下：

设一次乘法需3个周期(Cycle)，一次加法需一个周期，一次检查需2个周期，一次存或取需一个周期。

未优化的内循环		优化后的内循环	
4 个 “*”	12个周期	1 取，1 存	2 个周期
3 个 “+”	3	1 个“+”	1
1 取，1 存	2	1 次检查	2
1 次检查	2		
共用19个周期		共用 5 个周期	

优化后运行速度快四倍。

以上是不依赖于机器的优化。

然后生成代码。优化后代码也许会长一些，也不一定，这和变址寄存器的分配有关。另外，也有的机器把增量和检查合成一个结构实现。

三、中间语言

Ⅰ）中间语言的形式：

表达式如

$$A=B+C+D+E$$

其分析方式是任选的。如果是左结合方式分析，即

$$A=((B+C)+D)+E$$

(FORTRAN不是这样分析的，PL/1是这样分析的)。

也可这样分析：

$$A=(B+C)+(D+E)$$

从优化角度来看，这种分析好。因为分成两个独立的子表达式，外提可以独立的进行。

表达式的内部形式分两部分：文本——字典(TEXT——DICTIONARY)。

其中字典，即相应的符号表形式很多，可有：

1） 三地址文本形式

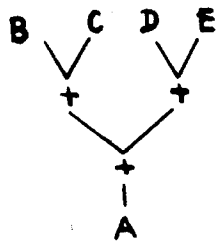
如 $A=B+C+D+E$的三地址文本为：

$T_1=B+C$

$T_2=D+E$

$T_3=T_1+T_2$

$A=T_3$



2） 树形形式

和1)的差别就在于不用临时变量，且关系联系紧密。

3） 波兰表示

$=+++BCDEA$

对优化来说，希望用1)因便于表达式外提，而2)关系太紧密，3)就更差了。

以后我们就采用1)。

临时变量的产生有很多文章谈到，这里仅简单介绍一下。这些问题是在分析时处理的。关于那种方法好，文献中没谈过。但这问题对优化很重要，所以要介绍一下。

1） 临时变量个数不限：

在分析时检查是否要临时变量，当产生的值在后面要用时，就存放在临时变量中。

2） 语句间重叠使用临时变量。

大部分编译程序用此法。

3） 对形式相等（“Formal Identity”）的子表达式给以同一临时变量。

这里要用杂凑码（Hashing）方法，此法用得很多，很重要。

“形式相等”的概念解释一下：例如

$t_5 = B + C$

$B =$

$t_5 = B + C$

这里的两个B+C就是“形式相等”的。

对形式上一样的子表达式给一个临时变量，不管其中是否有赋值。用Hashing方法就可达此目的。这样做，非常有利于多余子表达式的消除。如下例，用此法分配：

（1）$A = B + C + D + E$

$t_1 = B + C$

$t_2 = D + E$

$t_3 = t_1 + t_2$

$A = t_3$

（2）$X = D + E + B + C$

$t_1 = B + C$

$t_2 = D + E$

$t_3 = t_1 + t_2$

$X = t_3$

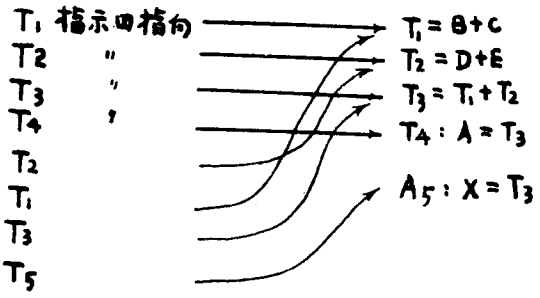
（这里本应为$t_2 + t_1$，利用交换律改成$t_1 + t_2$，然后通过Hashing码发现和前面有一样的，从而得$t_3 = t_1 + t_2$）

经过进一步的优化，（2）就可变成：

$X = t_3$。

这是好的办法，但也存在问题。如中间值有变化，就有问题。

所以主要问题是如何保持临时变量是正确的。有一个简单紧凑的方法。不用文本形式，而用字典形式。如上例得：



这就很简单。

现在正从事大机器上的研究工作，所以用1)与3)的结合。规模小时，一般用上形式，如UNIVAC 1107/1108的FORTRAN就是。

Ⅱ）中间语言的级别：

中间语言的级别依赖于源语言。如中间语言是没有寄存器的机器一级指令，此时“取”就有问题。

例如：A = B + C　　写成

$t_1 = B + C$

$A = t_1$

这样写是为了可以处理表达式外提。这种情况t_1用寄存器R_1为好。

另一种形式，则如下：

Load B into R_1

Load C into R_2

Add R_1, R_2 into R_3

Store R_3 into A

这样的形式不利于循环优化和外提，因如寄存器不够，则要加临时变量。

Load B into R_1

Store R_1 into Temp

而在循环内要

Load Temp into R_1

Load C into R_2

⋮

Add R_1,R_2 into R_3

⋮

这样就更不好。所以还是用前一种好。

如用高级语言就不需要这无寄存器的机器语言一级的中间语言了。

如 Real A(10, 10), B(10, 10), C(10, 10)

A = B + C

扩展后为：

```
I = 1, 10
  J = 1, 10
  A(I,J) = B(I,J) + C(I,J)
```

如再有 X = B + C

扩展后就可发现A = B + C, X = B + C是一样的。

在高级语言情况，一般两级就可以了。

四、优化的级别

优化的等级一般有三种：

局部优化（只用有限代码区域的信息）

全局优化（用全过程的信息）

交错过程（Cross-procedure）

局部优化，由于编译速度快，所以还常用。全局优化编译速度

慢。一般编译程序用前两者。第三级做得很少，有关文献也很少。目前本人正从事这方面工作，因时间不多，这方面不准备讲，而且其中问题还不少。

五、局部优化问题

局部优化是从基本块出发的。

一个具有一个入口（即第一条执行指令）及一个出口（最后一条执行指令）的直线的指令序列称为基本块。

基本块前面可有先行块，后面可有后继块。基本块内是直线的，所以不用担心转移问题。

局部优化是用值编号码（Value Numbering）的方法。可参见Cocke and Schwartg的文章"Programming language and their Compilers" Cocke先生作了不少优化基础工作，他有不少文章。

这方法用于多余子表达式的消除。

例：
```
A = J
I = J * K
L = L + I
I = J * K
L = L + I
I = A * K
```

这程序当然是写得不好的。但从统计学观点来看，这种程序还是常出现的。Knuth在统计方面作了不少工作，他认为：

50%的FORTRAN语句有形式A = ……

60%的FORTRAN语句有形式A = B，A = 5之类的。本人认为前一数据是对的，对后一数据感到惊奇。

本人感到有件事常被人们忽视——统计分析，不大清楚程序是如

何用的。这方面唯一的文章是Knuth的文章。但统计分析对优化很重要。本人亦只是程序看多了，有统计的印象。

上例就是一个基本块，分析这个例子，会发现第二个J＊K不需要作了。而最后一个A＊K就是J＊K，所以也不需要作了。

现在来建立一个表格。

	作表达式	变量和表达式	值号码
A＝J		J	1
	是	A	1
I＝J＊K		K	2
		J＊K〔①＊②〕	3
	是	I	3
L＝L＋I		L	4 5 6
	是	L＋I〔④＋③〕	5
I＝J＊K	否		
L＝L＋I	是	L＋I〔⑤＋③〕	6
I＝A＊K	否		

加工A＝J时，J先进表，并给值号码1，然后A进表。因是A＝J，所以A的值号码亦是1。加工完了，则在“作表达式”栏上填上“是”。然后加工I＝J＊K，此时J因表中已有，J就不进表，且其值号码就是1。K进表，值号码是2。然后J＊K进表。并注上是〔①＊②〕，然后给值号码3。最后I进表，其值号码亦是3。至此，在“作表达式”栏填上“是”。然后加工L＝L＋I，L进表，值号码为4，I表中已有，值号码为3，然后L＋I进表，并注上〔④＋③〕，给值号码为5。然后加工等号左端的L，L表中已有，但此时值号码应为5，所以把表中L的值号码4改为5。如此下去。对I＝J＊K，I＝A＊K都是表中已有，所以不需计算。在“作表达式”栏填上“否”。注意，后面的L＝L＋I虽形式和前面的L＝L＋I相同。但前面是④＋③，而后面的⑤＋③，所以实际并不相同。

这样边作边删。查表用Hashing技术。

这样作，可提供那些信息可以保持在寄存器里。值号码相同的变量可分在同一个寄存器里。值号码不同的同一个名字的量必须放在不同的寄存器中。当然还可以进一步压缩。很重要的是那些不同名字的量可以放在同一个寄存器里。

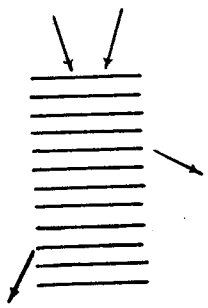
关于基本块的文章很多。但关于这个方法的，只有Cocke and Schwartz的文章。我认为这个方法好。

局部优化就介绍到这里。再引进一个概念——扩充的基本块。

关于这方面，工作作的不多，但发现很有用，上述方法也可用于这种块。

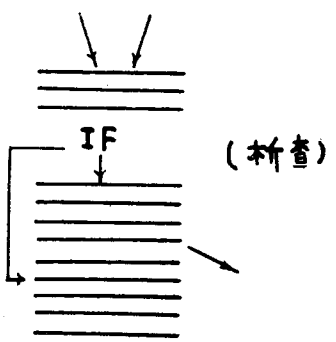
定义：一个指令序列，它有一个入口（即第一条执行指令），且除了第一条指令外，该序列中每条指令有且仅有一条先行指令。

如下图是扩充基本块



它中间可以转出去，但没有进来的。

又如



如用值编码方法，这时就需用栈，顺序要保证正确。

要注意的是许多程序可压缩到几个这样的扩充基本块。因为它可以包含IF类的语句。用扩充基本块可达到较好的优化。

现在美国有倾向不用转移语句。我个人认为完全不用不大可能，不过应尽量不用go to。因为没有go to语句时，优化起来方便。

有的语言有条件循环语句，介绍PL/1时再讲。有了条件循环语句，就可把转移语句去掉。

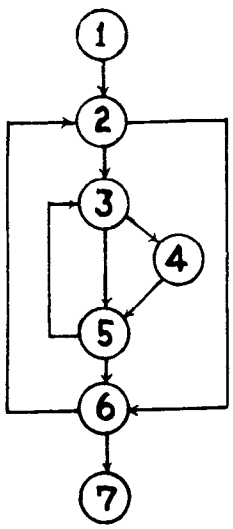
六、全局优化

重要的是如何系统地看一个程序（Choose A "Schematic"）。用图的方法可以有两种：

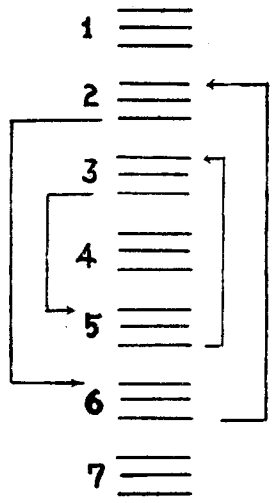
1） 给出一代码序列，标出向前转移和向后转移。如前面举例就是。这种图是一种简单常用的表示法。

2） 控制流分析：

用节点表示一个基本块，用有向线段表示控制线路。如下图。



如用1)则为：



用2)可便于信息更换，如可把④放在⑦下面。而用1)就不容易作到。

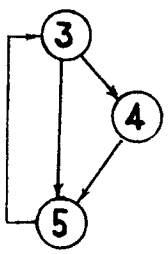
1)、2)是不同的两种看问题方法，各有各的长处。

现在优化的大部分工作用2)。以前一般用1)。UNIVAC上FORTRAN大部分用2)。

首先谈谈为何要用控制流程图：

1) 易于删掉某些序列

从图中可以看出那条路线有检查，那些路不必经，如何进一步来化简。如图1中的



就可进一步化简成一个节点。

2) 易于调换次序

假如③处是检查出错的，当不出错时，就不执行④，但它在图中还占位置。

好的优化是尽量使存贮占用的少。所以一般把不常用的语句不放在主存中，而用控制流程图表示就有利于调换节点次序来作这工作。

3） 易于引进新节点

如在③中有C＝A＊B可以外提。但如外提至②，而⑥中有X＝C时，则⑥中这个X＝C就错了。因此必须在②与③间加上一个节点㉗，用它来放C＝A＊B，且㉗是在循环外的，这样就改进了运算。

用这种方法还可利用数学图论的理论。

全局优化的基础有两个：控制流分析和数据流分析。这两部分在本人七一年写的文章上谈过，发表在IFIPS 71年第一卷上，今天只简单地讲一些，因为还想讲讲公共子表达式的节省、代码外提及強度削减、以及其他总结性的看法。还想谈一些与该文章不一样的地方。另外还想说明一下，控制流分析与数据流分析是公共子表达式节省、代码外提及強度削减的基础。

Ⅰ、一些概念

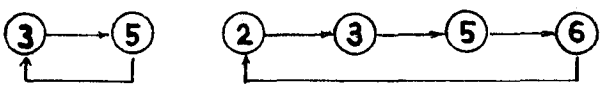
请看图2，说明下述概念。

先行块：②，③，⑤是⑥的先行块。

直接先行块：⑤是⑥的直接先行块。

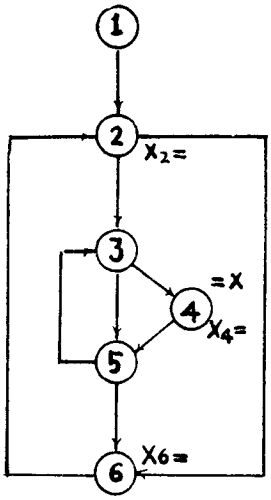
后继块：⑥，②，③是⑤的后继块。

直接后继块：⑥是⑤的直接后继块。

循环：　如 

现在讲定义和使用关系，以及有效（live）的含义。请看图2，②中有X的定义，④中有X的使用，又有X的定义，⑥中有X的定义。我

们用X_2，X_4，X_6分别表示在②，④，⑥中定义的X，从图看，X_2定义可影响④中X的使用；经过④→⑤→③→④，X_4定义可影响④中X的使用；但X_6定义不能达到④，因为必须要经过②。



由此，我们可造出数据流图。在优化时很少使用数据流图，但数据流图的概念是很有用的。

有效(live)的概念不仅表示定义——使用关系，而且表示出走向的路线。这对寄存器分配很有用。

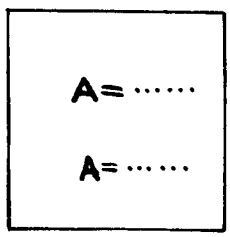
现在考虑定义X_2和④中X的使用：X_2定义在②→③→④是有效的，在③→⑤→③也是有效的。这样，在X_2和④中X的使用就可分配同一个寄存器R_1。

要看到X_4会影响④中X的使用。因为按④→⑤→③→④，X_4是有效的，这样，X_4也分在寄存器R_1。这样，从②→③→④→⑤→③→④都使用寄存器R_1，这样问题就复杂了。

分析出哪些定义是有效的，还不能完全解决寄存器分配问题。

现在讲如何用系统的方法来发现定义是有效的。这里介绍一下七一年文章的基本思想。

对于一个基本块bi，为进行数据流分析，要找出bi中的可用定义(Available definitions)，记为Dbi，以及没被切断的定义(Definitions not killed)，记为$\overline{K}$bi。



可用定义，即基本块bi中最后的定义，如右图中，后一定义A即为可用定义。可用定义