代码翻译中 PERFORM 和 GOTO 语句复合结构的变换*

武成岗[1+], 张兆庆[1], 乔如良[1], 冯晓兵[1], 高 琳[1], 石学林[1], 蒋弘山[2], 崔慧敏[2]

[1](中国科学院 计算技术研究所,北京 100080)

[2](清华大学 计算机科学与技术系,北京 100084)

Converting the Compound Control Structure of PERFORM and GOTO Statements in Code Translation

WU Cheng-Gang[1+], ZHANG Zhao-Qing[1], QIAO Ru-Liang[1], FENG Xiao-Bing[1], GAO Lin[1], SHI Xue-Lin[1], JIANG Hong-Shan[2], CUI Hui-Min[2]

[1](Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100080, China)

[2](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: Phn: +86-10-62562702, Fax: +86-10-62564342, E-mail: wucg@ict.ac.cn, http://www.ict.ac.cn

Received 2003-08-26; Accepted 2003-11-11

Wu CG, Zhang ZQ, Qiao RL, Feng XB, Gao L, Shi XL, Jiang HS, Cui HM. Converting the compound control structure of PERFORM and GOTO statements in code translation. *Journal of Software*, 2004,15(4):475~486.
http://www.jos.org.cn/1000-9825/15/475.htm

Abstract: COBOL, a traditional language, has been presented for more than 50 years. There are at least 100 billion lines of legacy codes written in COBOL up to now. An effective way to maintain these legacy codes is to translate them into modern languages, such as Java. While translating, it is a key-step to eliminate 'GOTO' and 'PERFORM' and their compound control structures in COBOL programs. A method which uses 'switch' and 'while' statements is proposed in this paper instead of 'GOTO' and 'PERFORM' and their compound control structures. It preserves the readability because the target Java program has the similar control structures. The code size of the target program expands only 2 times in average. This method is applied in the 'C2J translation system'. It is proved sound and effective since 4 million lines of real COBOL program have been translated and its target program has passed the test.

Key words: legacy code; COBOL; JAVA; translation; PERFORM; GOTO; control-flow

摘 要: 传统语言 COBOL 从诞生至今已有近 50 年历史,现存约有 1 000 亿行代码是用 COBOL 编写的.维护这些遗产代码的一个有效方法是将其翻译成现代语言,例如 Java.其中将 COBOL 语言中 GOTO 和 PERFORM 语

* Supported by the National Natural Science Foundation of China under Grant No.60103006 (国家自然科学基金)

作者简介: **武成岗**(1969－),男,河南郑州人,博士,主要研究领域为高级编译技术,二进制翻译;**张兆庆**(1938－),女,研究员,博士生导师,主要研究领域为高级编译技术;**乔如良**(1937－),男,教授,主要研究领域为高级编译技术;**冯晓兵**(1969－),男,副研究员,主要研究领域为高级编译技术;**高琳**(1977－),女,博士生,主要研究领域为高级编译技术;**石学林**(1977－),男,博士生,主要研究领域为高级编译技术;**蒋弘山**(1974－),男,博士生,主要研究领域为高级编译技术和生物信息;**崔慧敏**(1979－),女,硕士,主要研究领域为高级编译技术.

句及其复合控制结构消除是翻译过程中的一个关键步骤.提出一种利用 switch,while 语句来消除 GOTO 和 PERFORM 复合控制结构的方法,实现了程序的等价变换.该方法不改变程序的控制结构,保持其可读性,并将代码膨胀率控制到 2 倍左右.该方法已在所开发的"C2J 翻译系统"中进行应用,通过了 400 万行实际商用程序的测试,结果证明,该方法是正确、有效的.

关键词: 遗产代码;COBOL;JAVA;翻译;PERFORM;GOTO;控制流

中图法分类号: TP314 **文献标识码**: A

传统语言(如 COBOL,FORTRAN 等)从诞生到现在已有近 50 年的历史.现存的大量商用与科学计算程序是采用这些传统语言编写的.目前在银行、证券和保险业中运行的程序,有许多诞生于那个时期.据不完全统计,大约 1 000 亿行代码是用传统语言 COBOL 编写的[1].由于现代语言以其优越性替代了传统语言,很少有软件从业人员去学习和使用传统语言,因此用这些语言编写的程序就成为了遗产代码.由此导致了一系列问题的出现.第 1 个问题是遗产代码的维护[2],目前仅有极少数了解传统语言的程序员去维护大量的遗产代码,无疑是一个巨大的负担,并且随着新语言的继续产生,将有更多的编程语言被替代,由这些语言编写的程序也就转变成新的遗产代码.随着时间的推移,遗产代码的种类和数量会不断地增加,维护遗产代码的负担将会越来越大,最后导致无法承受.文献[3]提出维护遗产代码的一个好方法是将遗产代码翻译成现代语言.第 2 个问题是遗产代码的存在影响了软件的互操作性.采用现代语言编写的新软件和同一行业的旧软件由于采用不同的编程语言,再加上新软件的编程者大多对传统语言不熟悉,从而导致新旧软件之间的链接和交互成为一个新的技术屏障.第 3 个问题是不利于代码的跨平台迁移.计算机网络的出现使计算机不再是一台独立的机器,而是整个信息网络的组成部分.各种不同类型、不同平台的计算机组成了复杂的系统,如何使该系统充分发挥其作用,是目前的网格技术研究的一项课题.一套软件能否跨平台工作决定了其是否具有更强的生命力.为了使代码顺利迁移,于是产生了如 Java,Net 这样的平台无关的程序设计语言.采用这类代码编写的程序,可以基本不受约束地在各个平台上迁移.因此,寻求一种将遗产代码转化为现代语言的技术,是实现代码迁移、使旧代码焕发新活力的一项很有意义的尝试.

目前,许多研究机构和公司都开展了这方面的研究,如圣彼得堡大学[4],Software Mining[5],Corporola[6],LegacyJ Corporation[7],Micro-Processor Service[8]等等.由于 COBOL 语言诞生较早,其遗产代码的数量大,且与现代语言的语法语义差别较大,因此选择它作为一个典型的传统语言.而 Java 则是一种典型的现代语言,它除了具备其他现代语言的特点之外,还为跨平台提供支持,并且摒弃了影响程序结构化的语句.与其他现代语言相比,Java 与传统语言的差距最大.因此,代码转化多选择 COBOL 和 Java 作为典型的研究对象.

现代语言为追求程序结构化而不提倡使用 GOTO 语句,甚至一些新的语言(如 Java)已不再提供该语句.但那些诞生于计算机发展早期的遗产代码中大量使用 GOTO 语句,甚至在 COBOL 语言的代码中,除 GOTO 语句外还大量使用了 PERFORM 语句.因此在将这些遗产代码翻译成现代语言时,必须将这些控制流语句通过等价变换进行消除.

目前的文献和一些研究机构推出的方法可以分为两类:一类是通过对遗产代码的控制流进行分析,将 GOTO 转化成循环,再用现代语言的 while 语句来实现这些循环[9].该方法虽能有效地消除 GOTO 和 PERFORM 语句,但改变了程序的结构,变换后的代码可读性差.另外,为了从复杂的 GOTO 和 PERFORM 结构中转化出相应的循环,必要时需进行代码复制,从而引起了代码的膨胀,经测试该类方法引起的平均膨胀率为 1~2 个数量级.像 Java 这样的语言对一个函数内的代码总量存有限制,过于膨胀的代码可能会导致一个函数过大而无法执行.

另一类方法是采用函数递归来模拟 GOTO 和 PERFORM 语句[5].该方法的一个明显特点是利用递归实现循环,而对于循环次数较多的结构,可能会导致过深的递归从而使堆栈无法承受,造成溢出,引起程序的执行意外中止.另外,该类方法也破坏了程序的结构,变换后的程序可读性差.

由于 GOTO 语句影响了程序的结构化[10],多年来一直有人在研究如何消除 GOTO 语句,并提出了许多方法,其中较具代表性的有文献[9,11~13]等.但这些方法并不太适合 COBOL 语言中 PERFORM 和 GOTO 语句形成的复合结构,为此,本文在文献[12]的基础上,提出一种利用 switch 和 while 语句来同时消除 GOTO 和 PERFORM

复合控制结构的方法.该方法不改变程序的结构,保持了程序可读性,并较好地控制了代码膨胀.目前该方法已在我们开发的“C2J 翻译系统”中进行应用,并通过了 400 万行实际应用程序的测试.

本文第 1 节提出 GOTO 和 PERFORM 复合控制结构的消除算法.第 2 节介绍相关工作.第 3 节给出实验结果.第 4 节进行了总结.

1 算法设计

1.1 COBOL中的GOTO和PERFORM语句

在将传统语言翻译成现代语言时,必须解决的一个问题就是控制流语句的变换.特别对于像 Java 这样的目标语言来讲,GOTO 语句的消除成为一个必不可少的环节.在介绍算法之前,需要先了解一下传统语言 COBOL 中的两种已被现代语言摒弃的控制流语句.

其中 GOTO 语句的作用与 C 语言中的 goto 语句相同,就是将控制转入过程中的另一个位置.

PERFORM 语句是具有两种功能的语句,一方面它相当于一个循环语句,在某种循环条件下执行循环对象,另一方面它具有调用功能,调用某个节或几个节执行,其基本语法结构如图 1 所示.

⟨perform⟩::=⟨perform_until⟩|⟨perform_varying⟩|⟨perform_times⟩.

⟨perform_times⟩::=“PERFORM”[⟨procedure_name⟩[“THROUGH”|“THRU”⟨procedure_name⟩]]

[⟨identifier_cint⟩“TIMES”] [⟨statement_list⟩“END-PERFORM”].

⟨identifier_cint⟩::=⟨identifier⟩|⟨cint⟩.

⟨perform_until⟩::=“PERFORM”[⟨procedure_name⟩[“THROUGH”|“THRU”⟨procedure_name⟩]]

[[“WITH”]“TEST”⟨before_after⟩] “UNTIL”⟨condition⟩[⟨statement_list⟩“END-PERFORM”].

⟨perform_varying⟩::=“PERFORM”[⟨procedure_name⟩[“THROUGH”|“THRU”⟨procedure_name⟩]]

[[“WITH”]“TEST”⟨before_after⟩]

“VARYING”⟨identifier⟩“FROM”⟨identifier_literal⟩“BY”⟨identifier_literal⟩

“UNTIL”⟨condition⟩{⟨outer_loop⟩}[⟨statement_list⟩“END-PERFORM”].

Fig.1 The grammar of the ‘perform’ statement

图 1 Perform 的语法结构

分析这个貌似复杂的结构,我们不难发现,无论是 perform_until,perform_varying 还是 perform_times,其区别主要在于对循环条件的控制.这一点我们可以轻松地用 Java 语言中的 while 语句来实现.

PERFORM 语句在向 Java 转换时,其最大的问题不再是循环的转换,而是 PERFORM 对被执行对象的调用方式,以及该方式与 GOTO 语句结合起来所形成的复合控制结构.为了能够清晰地说明问题,我们抹去循环,单纯地来分析其调用方式.

在如图 2 所示的简单例子中,段 L1 内有一条 PERFORM 语句,该语句带有一个 Through 子句,其语义是当条件 condition-1 成立时,从执行段 L1~L3(包括 L3 所指示的段),执行到 L4 之前,返回该 PERFORM 语句.在段 L2 中有两条 GOTO 语句,分别跳转到段 L0 和段 L1.第 1 个跳转语句是向前跳转,实际上构成了一个循环控制流.而第 2 个则是向后跳转,如果前面的“**PERFORM L1 thru L3**”语句在执行 L2 段时,条件 condition-3 满足,则控制转向 L4.但如果控制不再转回,则顺序执行到“**exit-program**”程序结束,不再返回到前面的 PERFORM 语句.如果在 L4 到“**exit-program**”之间有跳转语句将控制返回到 L3 或 L3 之前,当程序执行到段 L3 的结尾时,控制还可以重新返回前面的 PERFORM 语句.

```
L0.
    …
L1.
    …
     if condition-1 then perform L1 then L3
    …
L2.
     if condition-2 then goto L0
    …
     if condition-3 then goto L4
    …
L3.
    …
     if condition-3 then goto L3
    …
L4.
    …
     perform L3 until var EQUAL 0
Ln.
    …
     exit-program
```

Fig.2 The example constructed by the ‘goto’ and ‘perform’ statement

图 2 由 goto 和 perform 语句构成的控制流例子

可以看出,PERFORM 语句对每个段的执行非常类似于现代语言中对函数的调用,但不同的是,段不像函数那样独立,而是一个松散的结构,它允许 GOTO 语句在段之间跳转,这样势必使段不具有其他语言中函数的封闭性.当 GOTO 语句跳出了 PERFORM 所执行的对象,控制流将继续向下进行,直到程序结束或满足 PERFORM 的返回条件为止.因此,并不能简单地将每个段构造成一个过程,通过函数调用来模拟 COBOL 中的 PERFORM 和 GOTO 语句.

上述例子比较简单,其实还可能出现更为复杂的情况.比如一个 PERFORM 语句所执行的若干个段中可能还包含其他的 PERFORM 语句,这些 PERFORM 语句理论上将是允许不受任何限制地执行其他任意范围的若干个段.从本质上来讲,上述问题的产生源自于 COBOL 语言中段的非结构化.

这一问题成为 COBOL 向 Java 程序变换过程中一个极为棘手的问题,控制结构变换是否合理决定了语言转换的成败.

Software Mining[5]则是利用函数调用方式来实现,其方法我们将在相关工作中加以介绍.

1.2 变换算法设计

由于 Java 没有 GOTO 以及类似语义的语句,不能够直接用 GOTO 语句来仿真 COBOL 中的 PERFORM 和 GOTO 语句,但幸运的是,Java 与其他现代语言中都提供了 switch 语句,利用它们可以实现控制流的转移.

具体算法如下:

(1) 将 COBOL 程序的一个“过程”转换成一个同名的 Java 类,该类包含一个入口函数,我们将该函数命名为 _entry,该函数的参数列表对应于原 COBOL“过程”的参数列表.

(2) _entry 中包含一条复合语句,其作用是对另一个函数 sub 进行调用,并捕捉 sub 在执行结束语句时抛出的例外.

(3) sub 函数用于实现 COBOL“过程”的过程体.该函数的参数列表中除了包含与_entry 函数具有同名的函数列表以外,还在前面增加了两个参数 entrance 和 exit.entrance 作为 PERFORM 语句执行的入口.exit 作为

PERFORM 语句执行对象的出口.我们可以认为,原来的整个程序相当于从第 1 个段进入,从最后一个段退出.因此,在_entry 中,sub 的实参为 0 和 last+1.

至此,_entry 的形式如下:

```
public void _entry(vector paraList){
 try{
   sub(0,last+1,paraList);
 }catch (Exception e){ }
}//end of method _entry
```

(4) 接下来对 sub 的函数体进行设计.

① 段名的处理

对段名进行编号,从 COBOL 程序的第 1 个段开始,依次从 0 向下进行编号直到最后一条语句.为了便于程序一致性的维护,要求在程序结束前增加一个编号.

将整个程序置于一个循环中,其基本形式如下:

```
void sub(int entrance, int exit, vector paraList)
   parameterdispose(paraList); //参数处理
    label=0;
    end:
    while (true){
          switch(label){
                  case 0:
                        第 1 个 Segment 对应的代码
                  case 1:
                        第 2 个 Segment 对应的代码
                        …
                  case last+1:
                        throw new Exception();
                  }
   }
}
```

其中每条 case 语句依次对应于 COBOL 程序中的一个段标号,其后面的代码对应于一个段的内容.

语句“**throw new Exception()**”对应于 COBOL 中的“**exit-program**”,用于结束一个过程.另外,为了程序的一致性,在结束语句前增加了一条 Case 语句,其序号为 last+1.

② GOTO 语句的处理

遇到 GOTO 语句时,按照文献[12]的思想可以将其翻译成如下形式:

{**label=dest**; **continue end**;} //这里 dest 是跳转目标所对应的编号

③ PERFORM 语句的处理

当遇到 PERFORM 语句时,将其翻译成如下形式:

sub(**en**, **ex**, **paraList**); //其中 en 为入口,ex 为出口对应的段的编号

为了让 sub 执行完 ex 所对应的段后返回 PERFORM 语句,我们需要在每个段尾增加一条语句:**if (exit ==segNumber) return**; //其中 segNumber 是本段的编号.

另外,PERFORM 语句还可能带有循环条件,循环条件的实现非常简单,只需要用 while 和 for 语句来完成即可.

对于前面的例子,我们可以用如图 3 所示的 Java 程序模拟.

至此,变换算法结束.

```
public void _entry(vector paraList){
 try{
  sub(0,last+1,paraList);
 }catch (Exception e){ }
}//end of method _entry
private void sub(int entrance, int exit, vector paraList){
 parameterdispose(paraList);                    //参数处理
 int label=entrance;
 end:
 while(true){
   switch (label) {
     case 0: ...
          if (exit==0) return;
     case 1:...
       if (condition-1)sub(1,3,paraList);        //if condition-1 then perform L1 then L3
          ...
          if (exit==1) return;
     case 2:...
       if (condition-2) {label=0;continue end;}  //if condition-2 then goto L0
          ...
       if (condition-3) {label=4;continue end;}  //if condition-3 then goto L1
          ...
          if (exit==2) return;
     case 3:...
       if (condition-3) {label=3;continue end;}  //if condition-3 then goto L3
          ...
          if (exit==3) return;
     case 4:...
          while(var!=0) sub(3,3,paraList);
          if (exit==4) return;
     case last+1:
      throw new Exception();
        }
   }
 }
```

Fig.3 Corresponding Java program of the example in Fig.2

图 3 图 2 例子所对应的 Java 程序

1.3 算法的正确性证明及性能评估

该算法实现了将 COBOL 语言中的两种特殊控制流语句进行消除.下面我们来分析一下该算法的正确性并对其进行性能评估.

1.3.1 算法的正确性证明

按照图灵机的理论[14],一个确定的程序,如果每次执行前的初始状态相同,并且在执行过程中所遇到的输入

相同,则结果也相同.也就是说,在输入确定的情况下,程序执行过程中所经历的动作序列也是确定的.

从本质上讲,代码转换的目的是通过转换得到一个目标程序,该目标程序与源程序在任何相同的初始状态和任何相同的输入情况下,产生同样的结果.

为了便于证明,我们给出如下几个定义:

定义 1. 程序在某一时刻的状态.

对于高级语言来讲,程序在某一时刻的状态可定义为 $k=(p,s,v,o)$:

(1) p 为当前执行的语句的位置(即该语句对应的指令序列的第 1 条指令的内存地址);

(2) s 为栈的状态,栈中记录两方面的内容:调用关系和数据(参数、局部变量);

(3) o 为计算机及外设的状态;

(4) v 为所有当前有效变量的值.

COBOL 源程序和 Java 目标程序从启动开始到执行结束,各自都会发生状态的变化,根据程序变换的目的,我们定义以下映射规则.

定义 2. 状态映射规则.

设 k 和 k'分别是 COBOL 源程序和 Java 目标程序在某一时刻的执行状态,如果满足下列条件,就称 k 和 k'满足映射规则,记作 $k\rightarrow k'$.

(1) p'值是 p 指示的 COBOL 语句所对应的 Java 语句的位置(记作 $p\rightarrow p'$);

(2) s 为 COBOL 栈的状态,s'为 Java 栈的状态.s'与 s 记录的调用关系一致,且 s'实现了 COBOL 参数的正确传递(记作 $s\rightarrow s'$);

(3) v 为 COBOL 中的变量状态,v'是 Java 程序中与之相对应的变量的状态.v 和 v'对应变量的值相同(记作 $v\rightarrow v'$);

(4) o 为 COBOL 中的机器状态,o'是 Java 程序中与之相对应的机器的状态.o 和 o'的对应值相同(记作 $o\rightarrow o'$).

定义 3. 执行动作的映射规则.

设 a 为 COBOL 源程序的一条语句,a'是经过翻译后对应于 a 的一个执行单元(可能是一条语句,也可能是一段代码),f 和 f'分别是二者的执行动作.如果满足下列条件:

设 k_i 和 k_i'分别是 a 和 a'执行前的状态,如果 $k_i\rightarrow k_i'$,$k_{i+1}=f(k_i)$,$k'_{i+1}=f'(k_i')$,且 $k_{i+1}\rightarrow k'_{i+1}$,我们就称 f 和 f'满足映射规则,记作 $f\rightarrow f'$.并且如果 $f\rightarrow f'$,则称 a 和 a'也满足映射规则,记作 $a\rightarrow a'$.

定义 4. 源程序向目标程序的转换是成功的.

根据程序翻译的目的,我们可以认为,如果源程序和目标程序在相同的输入下,初始状态满足 $k_s\rightarrow k_s'$,且每执行一步的状态都满足 $k_i\rightarrow k_i'$,则目标程序就实现了源程序的功能,因此,可以称源程序向目标程序的转换是成功的.

推论.

如果源程序和目标程序在相同的输入下,初始状态满足 $k_s\rightarrow k_s'$,且源程序和目标程序每条语句都满足 $f\rightarrow f'$,则源程序向目标程序的转换是成功的(这一点可以采用数学归纳法证明,证明过程比较简单,不列入正文,可参考附录).

本文旨在讨论控制流等价变换问题,因此这里假定其他对控制流无影响的语句均能实现等价变换(在我们所开发的 C2J 翻译系统中,已经实现了这些语句的正确变换,但这不是本文重点讨论的内容).设 a'是转换后的执行单元,f 和 f'是 a 和 a'的执行动作,我们有:$a\in\Sigma-\{$GOTO,PERFORM,evalue,if$\}$,且 $f\rightarrow f'$(其中 Σ 是 COBOL 所有语句的集合).

在代码翻译时,我们对段的结构和控制流语句进行了变换.因此,根据上述推论和假设,我们只需对如下 3 点进行证明,就可以说明我们对程序的转换是正确的.

(1) 初始状态正确映射;

(2) 段结构的正确映射;

(3) 若 $a\in\{$GOTO,PERFORM,evalue,if$\}$,有 $f\rightarrow f'$.

• 初始状态正确映射

设 $k_s=(p_s,s_s,o_s,v_s)$是 COBOL 程序起始的状态.其中 p_s为第 1 条语句的位置,s_s为调用栈的初始值,o_s,v_s分别是设备和变量的初始值.

目标程序在启动执行时,执行 sub(0,last+1,paraList).通过执行“参数处理”、“label=0;”、“while 和 switch”进入 case 0,这些语句都不影响 v_s和 o_s对应信息的状态,因此 $v_s\rightarrow v_s'$,$o_s\rightarrow o_s'$.case 0 对应的是 COBOL 的第 1 条语句,因此 $p_s\rightarrow p_s'$.目标程序的栈中压入了进入 sub 时的返回地址.从表面上看,似乎比源程序增加了一级调用.但是我们在程序的最后增加了一条“**throw new Exception**();”语句,当程序结束时,该语句抛出一个例外,由**_entry**()中的 **catch** 子句捕捉,控制自然地回到_entry(),将栈顶所有的内容退出.由此可见,这一级压栈在程序结束时不会影响栈的正常映射.另外,paraList 由 **parameterdispose** 进行处理,将它分配到相应的参数变量中,实现了参数的正确传递,故 $s_s\rightarrow s_s'$成立.因此有 $k_s\rightarrow k_s'$.

• 程序结构的正确映射

COBOL 和 Java 在程序结构上的区别在于,COBOL 的段是一种松散的结构,段和段之间没有明显的约束.在不作为 PERFORM 执行对象的情况下,一个段的最后一条语句执行结束后,控制流将进入下一个段执行.如果该段是程序中最后一个段,则程序结束.

其实,COBOL 程序段的结尾相当于有一个映射 f_{segend}.关于这一映射我们分两种情况考虑:

- 该段不作为 PERFORM 的执行对象.
- 该段作为 PERFORM 的执行对象.

我们先讨论第 1 种情况,第 2 种情况将在 PERFORM 语句的映射中加以讨论.

按照 COBOL 语义,如果当前段不是最后一个段 last,则 $f_{\text{segend}}(k_{\text{segend}})=k_{\text{segnext}}$.其中 k_{segend}=(段尾,s,v,o),k_{segnext}=(下一个段第 1 条语句,s,v,o).

如果 k_{segend}是最后一个段 last,则 $f_{\text{segend}}(k_{\text{segend}})=k_{\text{finish}}$.$k_{\text{finish}}$为程序结束时的状态.

在变换后的目标程序中,段和段之间增加了“**if (exit==本段的编号) return**;”_entry 启动 sub 执行时,是通过“**sub(0,last+1,paralist)**”来进行的.此时 exit 的值为 last+1.令 $k_{\text{segend}}\rightarrow k'_{\text{segend}}$.

如果 k'_{segend}不是段 last,则在执行“**if (exit==本段的编号) return**;”时,该语句条件不成立,控制转入下一个 case 的第 1 条语句,栈、变量和机器状态都不变,因此下一步状态为 k'_{segnext}=(下一个段第 1 条语句,s',v',o'),即 $f'_{\text{segend}}(k'_{\text{segend}})=k'_{\text{segnext}}$,且 $k_{\text{segnext}}\rightarrow k'_{\text{segnext}}$.

如果 k'_{segend}是段 last,则在该段执行结束后,进入 case last+1,然后执行语句“**throw new Exception**();”,抛出一个例外,由 entry 捕捉到该例外,结束整个程序的执行,因此 $f'_{\text{segend}}(k'_{\text{segend}})=k'_{\text{finish}}$.$k'_{\text{finish}}$为程序结束时的状态,可以看出 $k_{\text{finish}}\rightarrow k'_{\text{finish}}$.

因此,可以得出在第 1 种情况下:$f_{\text{segend}}\rightarrow f'_{\text{segend}}$,目标代码实现了相同的段结构.

• 控制流语句的正确映射

COBOL 包含的控制流语句包括 GOTO,PERFORM,if,evalue.evalue 语句是条件分支语句,该语句可以等价变换为 if 语句.Java 提供了 if 语句,可以用 if 直接实现相应的变换,因此这里仅讨论 GOTO 和 PERFORM 语句.

COBOL 中的 GOTO 语句其作用是将控制转向 GOTO 语句的目标,即 $f_{\text{GOTO}}(k_{\text{GOTO}})=k_{\text{dest}}$,其中 k_{GOTO}=(GOTO 位置,s,v,o),k_{dest}=(GOTO 目标,s,v,o).

在目标程序中,设 k'_{GOTO}=(GOTO 位置,s',v',o')为执行 GOTO 语句时的状态.GOTO 语句变换后的形式为“**label=目标语句的位置**;**continue end**;”.按照 Java 程序的语义,label 被赋值为“目标语句的位置”,程序进入下一次循环,由 switch 将控制转到目标语句所对应的 case.因此,控制转到目标语句的位置,虽然增加了一次循环,但栈的内容没有发生变化,并且该类语句不影响 v 和 o.因此有 $f'_{\text{GOTO}}(k'_{\text{GOTO}})=k'_{\text{dest}}$,其中 k'_{dest}=(GOTO 对应语句的目标,s',v',o').如果 $k_{\text{GOTO}}\rightarrow k'_{\text{GOTO}}$,则有 $k_{\text{dest}}\rightarrow k'_{\text{dest}}$,因此 $f_{\text{GOTO}}\rightarrow f'_{\text{GOTO}}$.

COBOL 语言中的 PERFORM 语句具有两方面的作用,一是实现循环,二是执行程序中的一个或多个段.

循环的作用体现在 PERFORM 后面的循环子句,如 times,until,varying 等,这些子句的作用就是判断循环条件,根据循环条件来决定是否结束循环.我们知道,循环条件均可用 while 语句来实现,即“循环条件的初始化”、

“循环条件在循环体中的变化”以及“对循环结束条件的判别”.这 3 点的实现在程序设计课本中均有介绍,本文不再赘述.下面仅讨论 PERFORM 对段的调用的实现.

PERFORM 对段的调用方式均可以用统一的形式表示:“**PERFORM en thru ex**”.其中 en 为入口,ex 为出口.因此当执行该语句时,由当前状态 k_{PERFORM} 到 k_{next},即 $f_{\text{PERFORM}}(k_{\text{PERFORM}})=k_{\text{next}}.k_{\text{next}}$=(目标地址 en,$s$,$v$,$o$),其中 s 为压入 PERFORM 的下一条语句的地址的栈的状态.当执行 ex 段的段尾时,程序返回.即 $f_{\text{segend}}(k_{\text{segend}})=k_{\text{ret}}$,其中 k_{ret}=(栈中的返回地址,s 将栈中的返回地址退栈,v,o).

与 PERFORM 相对应的 Java 程序为“**sub(en,ex,paraList)**”.设当前状态为 k'_{PERFORM},且 $k_{\text{PERFORM}}\rightarrow k'_{\text{PERFORM}}$.该语句的作用是将下一条语句的地址入栈,并且将控制转移到 case en 所对应的语句,即段 en 对应的第 1 条语句.因此有 $f'_{\text{PERFORM}}(k'_{\text{PERFORM}})=k'_{\text{next}}$,其中 k'_{next}=(目标地址 en,s 中添加一个 PERFORM 后的下一条语句的位置,v',o').可以看出,$k_{\text{next}}\rightarrow k'_{\text{next}}$,因此有 $f_{\text{PERFORM}}\rightarrow f'_{\text{PERFORM}}$.

当执行到 ex 段尾时,遇到 **if(exit==ex) return**;,设此时的状态为 k'_{segend},且 $k_{\text{segend}}\rightarrow k'_{\text{segend}}$.此时,exit==ex 成立.由该语句作用可知,$f'_{\text{segend}}(k'_{\text{segend}})=k'_{\text{ret}}$,其中 k'_{ret}=(栈中的返回地址,s 将栈中的返回地址退栈,v',o').可以看出,$k_{\text{ret}}\rightarrow k'_{\text{ret}}$,因此 $f_{\text{segend}}\rightarrow f'_{\text{segend}}$,即段作为被执行对象的情况下,实现了段的结构映射.

通过上述证明可知,本算法完全实现了段结构以及 PERFORM,GOTO 和其他控制流语句.

按照数学归纳法原理,若初始状态对应关系相同,每条指令的对应关系也相同,则运行结果也相同.

由推论可知,上述变换方法实现了控制结构的等价变换.

证明完毕. □

需要说明的是,由于 PERFORM 和 GOTO 完全是按照 COBOL 的执行机理进行变换的,该机理完全保证了无论 PERFORM 和 GOTO 语句如何组合,转换后的程序都可以完全实现其功能.

为了有一个关于复合控制结构的等价变换的感性认识,我们分析如图 2 所示的例子.

图 2 中段 L1 内的 PERFORM 语句,在图 2 中有一个对应的语句“**if (condition-1) sub(1,3,paraList)**;”.若条件 condition-1 成立,从执行段 L1~L3 对应的这段 Java 代码执行结束后,“**if (exit==3) return**;”语句将控制返回.在图 2 中段 L2 内的两条 GOTO 语句,图 2 中也有相应的两条复合语句对应,同样实现了相应的跳转.第 1 个跳转语句是向前跳转,同样构成了一个循环控制流.而第 2 个则是向后跳转,如果前面的“**sub(1,3,paraList)**”语句在执行 case 2 段时,条件 condition-3 满足,则控制转向 case 4.如果控制不再转回,则顺序执行到“**throw new Exception**();”抛出一个例外.该例外被_entry 中的 catch 所捕捉,从而程序结束,不再返回到前面的“**sub(1,3,paraList)**”语句.如果在 case 4 到“**exit-program**”之间有跳转语句将控制返回到 case 3 或 case 3 之前,则当程序执行到 case 3 的结尾时,控制还可以重新返回前面的 PERFORM 语句.

可以看出,图 3 中 Java 代码的控制流程与图 2 中的例子完全一致.

1.3.2 开销评估

本方法对 PERFORM 语句的实现实质上是通过递归调用来实现的.这样就会增加一次压栈操作,将 sub 的活动记录压入堆栈中.从空间开销上讲,每执行一次 PERFORM 需要压栈一次,实际上是向栈中压入一个活动记录.为了减小活动记录,我们将除了 entrance 和 exit 以外的参数放置于一个 Vector 中,用指向该对象的指针作为参数,从而将 sub 的参数数量限制成 3 个.活动记录的大小则成了一个较小的常数.

在 COBOL 程序执行 PERFORM 语句时,同样需要保留返回地址,进行压栈操作.因此,本方法与 COBOL 的空间开销之比实际上是一个较小的常数.GOTO 语句是通过 switch 语句实现的,因此没有增加空间开销.

众所周知,语言之间的转换势必会引起代码的膨胀,我们曾经按照文献[9]所介绍的方法来实现控制流的变换,结果带来了一个数量级的代码膨胀,其执行效率也随之大幅度下降.

采用本方法实现 GOTO 和 PERFORM 的变换,其语句膨胀关系见表 1.

Table 1 The expanding relation of statements

表 1 语句膨胀关系

Kinds of grammar	The needed number of statements
Framework of 'switch' and 'while'	3/file
Segment structure	2/segment
'GOTO' statement	2
'PERFORM' statement	1

我们可以通过下述公式计算语句膨胀率:

$$3+(2\text{count}_{段}+2\text{count}_{GOTO}+\text{count}_{PERFORM})/(\text{count}_{GOTO}+\text{count}_{PERFORM}).$$

通过对被测试文件的分析发现,由于 GOTO 和 PERFORM 变换导致的语句膨胀率一般在 2.5 倍左右,因此本算法的时间开销相对于文献[9]的方法小很多.

2 相关工作

在 COBOL 变换中,任何一个研究机构和软件公司都无法回避对 PERFORM 和 GOTO 的复合结构的变换问题.由于商业运作的需要,仅有几个机构公开了其变换方法,我们可以对其进行分析并加以比较.

文献[9]介绍了一种适用于 COBOL 到 Java 翻译的 GOTO 消除算法.它通过 GOTO 移动和 GOTO 消除两种操作来实现 GOTO 语句的变换.对于位于同一个层次相互间为兄弟关系的 GOTO-LABEL 对,通过 IF 语句或者通过引入 DO…WHILE 语句来消除.如果它们之间不是兄弟关系,则需要通过一系列 GOTO 语句的移动,使得 GOTO 语句和相应的 LABEL 位于同一层次,且两者位于同一组语句序列.这种算法适用于不可规约的控制流,对它进行扩充,将 PERFORM 语句的执行融入其中,可以实现 PERFORM 和 GOTO 复合语句的消除.但由于该算法需要频繁地移动 GOTO,在移动过程中又需要对部分代码进行复制.因此,这种方法虽然能够实现 COBOL 控制流的转换,但其代码膨胀率较大.

Software Mining[5]是一个商用软件,它为了实现 PERFORM 语句的转换,将每个段转化成一个函数.PERFORM 于是变成了对一个或几个函数的调用.但这种方法带来的一个问题是如何实现 GOTO 语句,即如何从一个段跳转到另一个段去执行.当目标段执行结束后,不再返回.由于每个段被转化成函数,并且 Java 没有提供GOTO语句.Software Mining则通过对目标段所对应的函数调用来实现.在此情况下,当目标段执行完毕后,为了继续执行下面的段而不是返回,Software Mining 在此调用中依次对目标段以及它后面的段所对应的函数进行了调用.当最后一个段执行结束后,强制结束整个过程.这种方法的确实现了 PERFORM 和 GOTO 的语义,但是又带来了一个问题.我们知道,一些编程者喜欢采用 GOTO 和 if 语句进行组合来构造一个循环.采用 Software Mining 的方法进行转换,程序变为如图 4 所示的形式.

```
S0.                                   void s0(){i=1;}
    Move 1 to I                       void s1(){
S1.                                     i++;
      …                                 if (i<100) {s1();s2();…;sn();exit(0);}
      Add 1 to I                      }
      If I<100 then goto S1           void s2(){}
S2.                                   …
      …                               void sn(){}
Sn.
    Exit-Program.
     COBOL program                          Corresponding Java program
```

Fig.4 The converting format of 'PERFORM' and 'GOTO' statements in Software Mining

图 4 Software Mining 中 PERFORM 和 GOTO 语句的变换形式

从上述变换中可以看出,由 if 和 GOTO 所构成的循环实际上在变换后由递归来实现了.其循环次数和递归深度相同.这种思想最后会由于过深的递归导致系统崩溃.

3 实验和测试结果

"C2J 翻译系统"是我们为银行业的遗产代码开发的一套软件,其着眼点是翻译一个包含 400 多万行

COBOL 代码的实际系统,本算法是该系统中的一部分.

该系统通过上述 400 多万行代码的测试,证明了本文中所述算法是完全正确的.另外,我们还对本文和文献[9]的算法进行了测试比较(由于 Software Mining 的缺陷使其不能适用于这个庞大的测试对象,故而没有对它进行深入的测试),其结果见表 2.

Table 2 Test result

表 2 测试结果

Method	Expanding rate of statements (source/target)			Expanding rate of file size (source/target)		
	Minimum	Maximum	Average	Minimum	Maximum	Average
COBOL to Java translation system	1.24	3.43	1.95	1.56	5.71	2.72
Method in reference[9]	4.87	58.3	23.5	5.78	62.5	26.7

在“C2J 翻译系统”中,基本上一条 Java 语句可以实现一条 COBOL 语句,一些完成复杂功能的语句(如 compute,inspect,start,read,write,display 等)需要用多条 Java 语句来实现.每条 PERFORM 和 GOTO 语句平均需要用 2.5 条 Java 语句.经过大量统计,目标程序相对于源程序来讲语句平均膨胀率为 1.95,也就是说,翻译后的程序比源程序多执行将近 0.95 倍的语句.

转换后的文件平均膨胀率为 2.72.其原因有二:一是语句数量上产生了一定的膨胀;二是为了使系统中目标程序能够具有与 COBOL 相同的高精度,我们采用了封装映射技术,由于 Java 不允许操作符重载,只能用较长的函数名来实现,因而会使文件字符数量有相应的增加.

上述数据表明,无论是语句数量还是源文件的平均膨胀率都比文献[9]的方法少一个数量级.

另外需要说明的是,文献[9]中方法过高的代码膨胀率,导致翻译出的代码 90%以上无法通过 Java 的编译,不能够运行.

4 结 语

本文给出了一种可靠的将 GOTO 和 PERFORM 复合语句消除的算法.该算法避免了一些其他算法所产生的改变程序结构、引起代码呈数量级膨胀的问题,而且目标程序保持了源程序的控制结构,具有较好的可读性,便于软件工作人员在目标程序上进行二次开发.

由于本项目所处理的对象是一个应用程序,其执行过程中不要求精确中断,故而文中没有涉及对精确中断的处理.但对于系统程序,对精确中断的处理则成为语言翻译中的另一个关键问题.因此,研究在翻译中的精确中断将是我们以后的一项重要工作.

References:

[1] Frank PC. Legacy integration changing perspective. IEEE Software, 2000,17(2):37~41.

[2] Kontogiannis K, Martin J, Wong K, Gregory R, Muller H, Mylopoulos J. Code migration through transformations: An experience report. In: MacKay SA, Johnson JH, eds. Proc. of the CASCON'98. Toronto: IBM Press, 1998. 1~12.

[3] Harsu M. Re-Engineering legacy software through language conversion. Technical Report, A-2000-8, Department of Computer and Information Sciences, University of Tampere, 2000. 150.

[4] Terekhov AA, Verhell C. The realities of language conversions. IEEE Software, 2000,17(6):111~124.

[5] Softeware Mining. COBOL Translation Toolkit (CORECT) Technical Architecture Overview Document Version 1.2. http://www.acm.co.uk/papers/techpaper.jsp

[6] Corporola Inc. http://www.corporola.com/product/COBOL2Java.html

[7] LegacyJ Corporation. PerCOBOL. http://www.legacyj.com/

[8] COBOL to JAVA source code translator and converter. http://www.mpsinc.com/cob2j.html

[9] Erosa AM. A GOTO-elimination method and its implementation for the McCAT C compiler [MS. Thesis]. McGill University, 1994.

[10] Dijkstra EW. GO TO statement considered harmful. Communications of the ACM, 1968,11(3):147~148.

[11] Ramshaw L. Eliminating go to's while preserving program structure. Journal of the ACM, 1988,35(4):893~920.

[12] Church-Turing Thesis. http://www.cs.williams.edu/~kim/cs361/F02/Church.pdf

[13] Erosa AM, Hendron LJ. Taming control flow: A structured approach to eliminating GOTO statements. In: Henri B, ed. Proc. of the 15th IEEE Int'l Conf. on Computer Languages. Toulouse: IEEE Computer Society Press, 1994. 229~240.

[14] Wegner P, Goldin D. Interaction, computability, and Church's thesis. Technical Report, 00-7, University of Massachusetts at Boston, 2000.

附录. 关于文中推论的证明.

采用数学归纳法证明:

首先,根据推论假设有 $k_s \rightarrow k_s'$;

设程序执行到第 i 步时,$k_i \rightarrow k_i'$.设 $f_i(k_i)=k_{i+1}$ 和 $f_i'(k_i')=k'_{i+1}$,根据推论假设有 $f_i \rightarrow f_i'$,由定义 3 可知,$k_{i+1} \rightarrow k'_{i+1}$.

根据数学归纳法原理可知,推论正确. □

2004 年全国理论计算机科学学术年会

征 文 通 知

由中国计算机学会理论计算机科学专业委员会主办，海军工程大学信息与电气学院承办的“2004 年全国理论计算机科学学术年会”将于 2004 年 10 月在武汉召开。会议录用论文将收录在正式出版的论文集中，欢迎大家积极投稿。现将有关征文要求通知如下：

1．应征论文应未在其他刊物或学术会议上正式发表过。特别欢迎有创见的论文和有应用前景的论文。

2．稿件要求用计算机打印，格式为 38 行×38 字，字体为 5 号宋体。稿件中的图形要求画得工整、清晰、紧凑，尺寸要尽量小；图中字体要求为六号宋体。稿件正文不超过六千字。标题、作者姓名、作者单位、摘要、关键词采用中英文间隔行文。稿件各部分依次为：一、引言；二、...；最后是结束语。附录放在参考文献之后；参考文献限已公开发表的，文中最好不要出现文献序号。参考文献的格式为：

序号 作者·书名·出版社所在地：出版社名，出版年代

序号 作者·论文名·出处，年代·卷号（期号）：起迄页码

务必附上第一作者简历（姓名、性别、出生年月、职称、学位、研究方向等）、通信地址和联系电话。并注明论文所属领域。请提供打印稿和电子稿各一份。来稿一律不退，请自留底稿。

3．征文范围

程序理论（程序逻辑、程序正确性验证、形式开发方法等）；计算理论（算法设计与分析、复杂性理论、可计算性理论等）；语言理论（形式语言理论、自动机理论、形式语义学、计算语言学等）；人工智能（知识工程、机器学习、模式识别、机器人等）；逻辑基础（数理逻辑、多值逻辑、模糊逻辑、模态逻辑、直觉主义逻辑、组合逻辑等）；数据理论（演绎数据库、关系数据库、面向对象数据库等）；计算机数学（符号计算、数学定理证明、计算几何等）；并行算法（分布式并行算法、大规模并行算法、演化算法等）。

4．征文截止日期：2004 年 5 月 1 日

5．论文投寄地址：（430033）武汉 海军工程大学信息与电气学院 张志祥 收

联系电话：027-83443985,83443984（张志祥，贲可荣）

电子信箱：tcs2004@vip.sina.com; hgzzx@163.com