基于活跃变量分析的流图语言的部分求值器*

陆卫东　金成植

（吉林大学计算机科学系　长春　130061）

摘要　本文描述了流图语言的自应用型静态部分求值器，它由活跃变量分析、抽象分析、标记和例化 4 部分组成. 在活跃变量分析基础上再作抽象分析，比以往的抽象分析获得的抽象解释更精确，也更利于产生较高质量的剩余程序. 转移压缩在例化中直接进行.

关键词　活跃变量分析，抽象分析，部分求值.

部分求值作为一种特殊的程序转换方法和程序优化技术，它在软件生成自动化，尤其是在目标代码的自动生成、编译器的自动生成和编译器的生成器的自动生成方面有着重要的作用.[1,2]近年来，大量的研究工作集中在对函数式语言的部分求值器构造的方法上，并且取得了许多成果[3,4]，而对过程式语言的部分求值器的研究却举步惟艰. 其主要的原因是函数式语言具有良好的数学性，从给定的函数及部分已知的函数参数构造对剩余参数等价的剩余函数比较直接；但过程式语言以赋值语句作为最基本的成份，破坏了数学性.

本文的主要工作是对 Jones 在文献[5]中描述的流图语言静态(Offline)部分求值器的改进和具体化. 流图语言作为基本的过程式语言，其程序由一组基本块组成，每个基本块由一组赋值语句序列和一个控制转移语句组成，保证只有一个入口和一个出口. 对文献[5]的改进主要表现在 2 个方面：一方面是抽象分析算法. Jones 是将被分析程序 P 中出现的所有变量收集在一起，组成一个变量组. 抽象分析是获取这一变量组的收敛(Congruent)的抽象解释(Div)，并且把这个统一的抽象解释作为对程序 P 中包含的各基本块上变量的抽象解释；这样的抽象解释过于保守，它存在着 2 个不足：一是在部分求值时，如果程序 P 中一个变量 V，它在某些基本块 $B_{D1}\cdots B_{Dn}$ 是未知的(V 的抽象值为 D)，但经赋值语句的作用，它在另外一些基本块 $B_{S1}\cdots B_{Sm}$ 总是已知的(抽象值为 S)，按 Jones 的抽象分析方法，一旦变量 V 在某一基本块上被抽象解释为 D，则 V 在 P 中任何基本块都被解释为 D. 这样就导致在基本块 $B_{S1}\cdots B_{Sm}$ 中由于 V 已知而可执行的操作无法执行. 二是由于基本块的例化版本的标号由原有基本块的标号和静态环境组成，对各基本块上的变量采用统一的抽象解释，有可能对某一块而言，静态环境中包含多余的对变量的解释，会导致例化产生冗余的代码. 为克服以上 2 个不足，本文通过对程序 P 的活跃变量分析，估计出 P 中每一基本块入口处的活跃变

---

* 作者陆卫东，1966 年生，博士，主要研究领域为部分求值，新型语言的语义及实现，软件自动化. 金成植，1935 年生，教授，主要研究领域为部分求值，新型语言，软件自动化.

本文通讯联系人：陆卫东，长春 130061，吉林大学计算机科学系

本文 1996-01-08 收到修改稿

量集;再利用抽象分析获得各基本块入口处每一活跃变量的抽象解释,以此使抽象分析更精确、例化产生的代码质量更高.另一方面是改进例化算法.在剩余程序中含有大量的 *goto* 语句,因此必须对某些转移语句进行压缩使某些基本块合并.*Jones* 是在例化后,由后处理部分来处理转移压缩的.本文则采用边例化边压缩的策略,省略了后处理,避免了再次扫描.本文构造的部分求值器的结构如图 1.

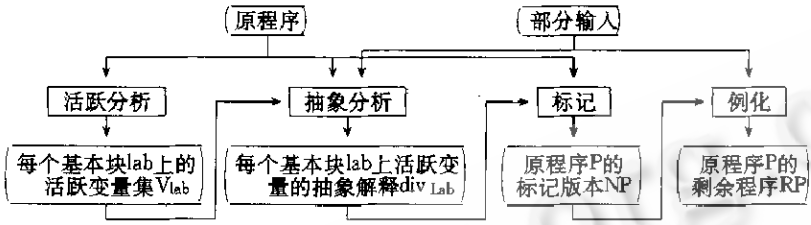

图1

1 流图语言 FCL

流图语言 FCL 是一个小的过程式语言,其程序由一组基本块组成,每个基本块包括一个赋值语句序列和一个控制语句.为保证基本块具有单入口和单出口的特性,限定控制语句必须是基本块的最后一条语句.控制语句有条件转移语句 if、无条件转移语句 goto 和终结语句 return 3 种.为使部分求值可自应用,把程序 P 和数据 d 定义在一个统一的数据域 *Date* 上,我们将 *Date* 定义为由 S-表达式组成的域.流图语言 *FCL* 的语法如下:

```
⟨Prog⟩       ::=read(⟨Var⟩,...,⟨Var⟩);⟨BasicBlock⟩+
⟨BasicBlock⟩ ::=⟨Label⟩:⟨Assignment⟩*⟨Jump⟩
⟨Assignment⟩ ::=⟨Var⟩:=⟨Exp⟩;
⟨Jump⟩       ::=goto ⟨Label⟩|return⟨Exp⟩
                |if⟨Exp⟩goto⟨Label⟩else⟨Label⟩
⟨Exp⟩        ::=⟨Constant⟩|⟨Var⟩|⟨op⟩⟨Exp⟩...⟨Exp⟩
⟨Constant⟩   ::=quote ⟨Val⟩
```

为使以下各部分算法表述得清晰明了,对 FCL 进行扩充,允许使用 while,repeat,for 和 case 语句,这些扩充的语句转换成 FCL 的形式是很直接的.

2 活跃变量分析

活跃变量分析是利用全局数据流分析的方法,收集程序中每一基本块入口处的活跃变量.如果将流图程序中的每一基本块看作一个结点,则根据流图程序的控制流,每一流图程序对应一个有向图.其中没有后继结点的结点称为终止结点,由包含终止语句的基本块对应;没有前驱结点的结点称为开始结点,由标号为 init 的基本块对应.一个变量 X 在某一块的入口处称为活跃的,如果存在着一条从该块标记的标号开始到终止结点的路径,在这条路径上有对 X 的使用,且该 X 在此路径中使用前未曾被定值过.

图 2 描述了活跃变量分析的算法,算法是用扩展的流图语言写的.子过程 ini_livemap 初始化各基本块入口处的活跃变量集,初始值为空集;live_exp 确定哪些变量在表达式中出现;find 寻找在 Livemap 中由 Lab 指称的变量集合.

```
read(Prog);Blocks:=get_blocks(Prog);Livemap:=ini_livemap(Blocks);/* 初始化 */
repeat      /* 迭代求 Livemap */
  Old_livemap:=Livemap;
```

```
for 程序中的每一基本块 Bb in Blocks do
  Lab:=get_Lab(Bb); Jump:=get_jump(Bb); Assp:=get_assp(Bb);
  case Jump of /* 根据块的控制语句判断其后继以求该块出口处的活跃变量集 */
    if E goto Lab1else Lab2:
              Lives:=find(Livemap,Lab1)∪find(Livemap,Lab2)∪live_exp(E);
    goto Lab: Lives=find(Livemap,Lab);
    return E: Lives:=live_Exp(E);
  endcase;
  RAssp:=revease(Assp); /* 将赋值语句序列倒置,因为活跃分析是从后往前的 */
  while RAssp 不为空 do /* 从出口从后往前地分析,求其入口处的活跃变量集 */
    Ass:=first_command(RAssp); RAssp:=rest_command(RAssp);
    Var:=get_ass_left(Ass); Exp:=get_ass_right(Ass);
    Lives:=(Lives-kill(Var))∪live_exp(Exp);
  enddo;
  Livemap:=update_livemap(Livemap,Lab1,Lives); /* 更新 livemap */
endfor;
until Livemap=Oldlivemap;
return(Livemap);
```

图2

3　抽象分析和标记

程序的执行是在实值环境下进行的.当程序给定部分已知输入时,虽然实值环境不完全,但程序中那些只依赖于静态值的操作在部分求值时是可执行的.部分求值的目的是在部分求值阶段利用部分已知输入预先执行掉程序中的可执行部分,而将不可执行部分保留下来组成剩余程序,以此提高程序的效率.判断程序中的某一操作在部分求值时可不可执行,并不取决于该操作的操作数是什么样的值,而是取决于操作数是已知的还是未知的这样的抽象值.已知的用抽象值 S 来表示,未知的用抽象值 D 来表示.抽象分析是在抽象环境下利用活跃信息对程序进行抽象分析以获取程序中每一基本块入口处活跃变量的抽象值.

图3描述了基于活跃变量分析的抽象分析算法.它的输入部分包括程序 Prog,Prog 中各块入口处的活跃变量集 Livemap 和 Prog 的输入变量的抽象解释 Init_div;输出是程序中各基本块收敛的抽象环境.算法中子过程 init_Blocks 对各块产生初始的抽象环境(Div),在该环境下,块入口处的每一活跃变量的抽象值为 S;create 形成限定抽象环境 Div 在变量集合 Lives 上的抽象环境;max 求2个抽象环境的最小上界($S \leqslant D$).

```
read(Prog,Livemap,Init_div); Abmap:=ini_Blocks(Livemap); /* 初始化 abmap */
Blocks:=get_blocks(Prog); Pend:={(init,Init_div)};
while Pend 不为空 do /* 迭代求 Abmap */
  从 Pend 中取一元素(Lab, Div)并将其从 Pend 中去掉;
  Lives:=find(Livemap,Lab); Division:=create(Div,Lives); /* 限定抽象环境 */
  Old_division:=lookup(Abmap,Lab);/* 寻找块 Lab 的抽象环境 */
  if Division≠Old_division then
    New_division:=max(Old_division,Division); /* 抽象环境的最小上界 */
    Abmap:=update_abmap(Abmap,Lab,New_dividion); Div:=new_division;
    Bb:=find_block(Brocks,Lab); Assp:=get_assp(Bb);
    while Assp 不为空时 do /* 在抽象环境下解释执行赋值语句序列 */
      Ass:=first_command(Assp); Assp:=rest_command(Assp);
      Var:=get_ass_left(Ass); E:=get_ass_right(Ass);
      Ab_val:=ab_eval(E,Div)/* 抽象求值 */; Div:=update_div(Div,Var,Ab_val);
    enddo;
    Jump:=get_jump(Bb);
```

```
    case Jump of /* 将执行本块后生成的新的抽象环境传递给它的后继块 */
      if e goto Lab1 else Lab2: Pend:=Pend∪{(Lab1,Div),(Lab2,Div)};
      goto Lab: Pend:=Pend∪{(Lab,Div)};
      return e: break;
    endcase;
  endif;
enddo;
return(Abmap);
```

图3

标记是根据抽象分析得到的程序中各个基本块入口处活跃变量的抽象解释，对各个基本块中包含的每一操作进行判断，哪些是在部分求值时可执行的，哪些是不可执行的，并将这些属于不同阶段执行的操作用不同的记号标识，标记后的程序属于两级语言(two_level)．标记算法见图4.

```
read(Prog,Abmap);
Blocks:=get_blocks(Prog); Ann_prog:=init_code;/* 产生标记版本程序的初始代码 */
for 每一个基本块 Bb∈Blocks do
  B_Lab:=get_Lab(Bb); Div:=lookup(Abmap,B_Lab);
  Assp:=get_assp(Bb); Ann_Assp:=ini_Assp;/* 初始化赋值语句序列的标记版本代码 */
  while Assp 不为空 do /* 生成标记后的语句序列 */
    Ass:=first_command(Assp); Assp:=rest_command(Assp);
    Var:=get_ass_left(Ass); E:=get_ass_right(Ass);
    E':=ann_exp(Div,E)/* 标记表达式 */; Ab_val:=ab_eval(Div,E);
    Div:=update_div(Div,Var,Ab_val);
    if Ab_val=S then Ann_ass:=gen_s_ass(Var,E');/* 生成静态的赋值语句 */
    else             Ann_ass:=gen_d_ass(Var,E');/* 生成动态的赋值语句 */
    Ann_Assp:=Ann_ass::Ann_Assp; /* Ann_Assp 是 Assp 的倒置 */
  enddo;
  Jump:=get_jump(Bb);
  case Jump of /* 生成块 Lab 的控制语句部分的标记版本 */
    if e goto Lab1 else Lab2:Ab_val:=ab_eval(Div,E);
                             E':=ann_exp(Div,E);/* 标记表达式 E */
                             if Ab_val=S then B_jump:=gen_ifs(E',Lab1, Lab2);
                             else B_jump:=gen_ifd(E',Lab1, Lab2);
                             Div1:=lookup(Abmap,Lab1); Div2:=lookup(Abmap,Lab2);
                             Dyn_var:=select_D(Div1)∪select_D(Div2);
    goto Lab1:B_jump:=gen_gotos(Lab1);
              Div1:=lookup(abmap,Lab1); Dyn_var:=select_D(div1);
    return e; E':=ann_exp(Div,e);B_jump:=gen_return(E');
              Dyn_var:={};
  endcase;
  B_Assp:=ini_Assp;
  while Ann_Assp 或 Dyn_var 不为空 do /* 标记本块需保留的静态赋值语句 */
    Ass:=first_command(Ann_Assp); Ann_Assp:=rest_command(Ann_Assp);
    case Ass of
      v:=^s e: if v∈Dyn_var then B_Assp:=(v:=^sd e) ::B_Ass;Dyn_var:=Dyn_var-{v};
                 else B_Assp:=(v:=^s e)::B_Assp;
      v:=^d e if v∈Dyn_var then B_Assp:=(v:=^d e)::B_Assp; Dyn_var:=Dyn_var-{v};
                 else B_assp:=(v:=^d e)::B_Assp;
      endcase;
    enddo;
    if Ann_Assp 不为空 then B_Assp:=append(reverse(Ann_Assp),B_Assp);
    Ann_block:=gen_block(B_Lab,B_Assp,B_jump);
```

```
        Ann_prog:=Ann_Block::Ann_prog;
    endfor;
    retuen(Ann_prog);
```

图4

根据本文给出的抽象分析方法,程序中每个基本块上的变量分别有各自的抽象解释,同一变量可能在某些基本块上有抽象值S,而在另一些基本块上有抽象值D;这就导致了在某一基本块中当E的抽象解释值为S而赋值语句$Var:=E$并不一定在例化时被消去.这些静态赋值语句不能被消去当且仅当在该赋值语句所在块的某个后继块B的入口处Var被抽象解释为D,并且该赋值语句是它所在块对Var的最后一次定值.在图4的标记算法中,第1个while循环将所有的静态赋值语句标记为$v:\overset{s}{=}e$,所有的动态赋值语句标记为$v:\overset{d}{=}e$.第2个while循环将所有的需保留的静态赋值语句$v:\overset{s}{=}e$标记为$v:\overset{sd}{=}e$.这类需保留的静态赋值语句与动态赋值语句不同,静态赋值语句的部分求值将改变实值环境,而动态赋值语句不改变.

4 例　化

例化是在不完全的实值环境下,对程序中可执行的操作进行执行,把不可执行的操作保留以组成剩余程序.通过标记过程,程序中可执行的和不可执行的操作已用不同的记号标识.程序中标记为静态赋值语句$v:\overset{s}{=}e$的,例化后将被消去;标记为$v:\overset{sd}{=}e$和$v:\overset{d}{=}e$的赋值语句将以$v:=e'$的形式保留在剩余程序中,其中e'是表达式e的例化版本.为使剩余程序简单明了,本文在例化过程中加入了转移压缩,即根据控制流将多个基本块合并成一个基本块.具体的就是在标记时把所有goto语句和条件静态已知的条件转移语句标记为静态可执行控制语句,例化时将由静态可执行控制语句连接的基本块压缩合并.这样例化后的基本块的控制语句只有条件未知的动态条件转移语句和return语句2种.

例化算法详见图5.它以程序Prog的标记版本Ann_prog,程序Prog的抽象环境Abmap和Prog初始基本块init的不完全实值环境S_env_0作为输入,输出是Prog的剩余程序R_prog.子过程seclet_S收集被抽象环境Div解释为S的变量;limit是限定实值环境S_env在变量集S_var上;eval是在实值环境S_env下求表达式E的值;reduce产生剩余的表达式.

```
read(Ann_prog,Abmap,S_env0); Blocks:=get_blocks(Ann_prog);
div:=Abmap(init); S_var:=select_S(div); S_env:=limit(S_env0,S_var);
Poly:={(init,S_env)}; Marked:={}; R_prog:=ini_code;
while Poly≠{} do
  从Poly取一元素(Lab,S_env)并从Poly中去掉它;
  Marked:=Marked∪{(Lab,S_env)}; RB_Lab:=gen_Lab(Lab,S_env); /* 产生新标号 */
  RB_assp:=ini_assp; Judge:=true;
  repeat /*块压缩:变量Judge根据控制语句是动态还是静态的确定其值以决定是否压缩*/
    Bb:=find_block(Brocks,Lab); Assp:=get_assp(Bb);
    while Assp不为空 do /* 例化赋值语句部分 */
      Ass:=first_command(Assp); Assp:=rest_command(Assp);
      case Ass of
        v:=s e: Val:=eval(e,S_env); S_env:=update(S_env,v,Val);
        v:=sd e: Val:=eval(e,S_env); S_env:=update(S_env,v,Val);
```

```
                RB_assp:=extend(RB_assp,v:=Val);
          v:=d e: R_e:=reduce(e,S_env); RB_assp:=extend(RB_assp,v:=R_e);
        endcase;
      endwhile;
      Jump:=get_jump(bb);
      case Jump of /* 例化动态控制语句;通过块合并消除静态控制语句 */
        ifs e gotos Lab1 elses Lab2:if eval(e,S_env) then Lab:=Lab1 else Lab:=Lab2;
                                    Div:=abmap(Lab); S_var:=select_S(Div);
                                    S_env:=limit(S_env,S_var);
        ifd e goto Lab1 else Lab2:Div1:=abmap(Lab1); S_var1:=select_S(Div1);
                                  Div2:=abmap(Lab2); S_var2:=select_S(Div2);
                                  S_env1:=limit(S_env,S_var1);
                                  S_env2:=limit(S_env,S_var2);
                                  Poly:=Poly∪{(Lab1,S_env1),(Lab2,S_env2)}-Marked;
                                  R_Lab1:=gen_Lab(Lab1,S_env1);
                                  R_Lab2:=gen_Lab(Lab1,S_env2);
                                  R_e:=reduce(e,S_env); Judge:='false;
                                  RB_jump:=gen_if(R-e,R_Lab1,R_Lab2);
        gotos Lab0:Div0:=abmap(Lab0); S_var0:=select_S(Div0);
                   S_env:=limit(S_env,S_var0); Lab:=Lab0;
        return e:Judge:='false; R_e:=reduce(e,S_env);
                RB_jump:=gen_return(R_e);
        endcase;
      until Judge
      RB_block:=gen_block(RB_Lab,RB_assp,RB_jump); /* 产生剩余程序的基本块 */
      R_prog:=RB_block::R_prog; /* 产生剩余程序 */
    enddo;
    return(R_prog);
```

图5

5 结 论

本文利用活跃变量分析构造了流图语言的静态部分求值器,文中详细地描述了活跃变量分析、抽象分析、标记和例化的算法,并以用流图语言所写的图灵机的解释程序为例,观察了各算法运行的结果,从这些结果不难看出与以往的部分求值器相比,利用活跃变量分析后,能使抽象分析更精确,例化产生的剩余程序的质量更高.

本文只研究了较简单的过程式语言——流图语言的部分求值器的构造方法,进一步的工作是对复杂的过程式语言例如C,PASCAL构造合理而高效的部分求值器.

参考文献

1 Jones N D, Sestoft P, Sondergaard H. An experiment in partial evaluation: the generation of a compiler generator. LNCS 202, 1985. 124~140.

2 Futamural Y. Partial evaluation of computation process-an approach to a compiler-compiler. System, Computer, Controls, 1971, 2(5):45~50.

3 Bondorf A. Automatic autoprojection of higher order recursive equations. Science of Computer Programming 17, 1991.

4 Launchbury J. Projections for specialization. In: Bjorner D, Ershov A P, Jones N D eds., Partial Evaluation and Mixed Computation, Amsterdam: North-Holland, 1988. 225~282.

5 Jones N D. Automatic program specialization: a re-examination from basic principles. In: Bjorner D, Ershov A P,

Jones N D eds., Partial Evaluation and Mixed Computation, Amsterdam: North-Holland, 1988. 225～282.

A PARTIAL EVALUATOR FOR FLOW CHART LANGUAGE BASED ON THE LIVENESS ANALYSIS

LU Weidong　JIN Chengzhi

(*Department of Computer Science　Jilin University　Changchun*　130061)

Abstract　This paper describes a partial evaluator for the flow chart language which is based on the liveness analysis. This partial evaluator consists of liveness analysis, binding time analysis, annotation and specialization. With the liveness informations, the binding time analysis can get the abstract interpretation to variables much more precisely than usual. So the quality of the residual program produced by the specialization is improved.

Key words　Liveness, binding time, partial evaluator.