一种高效的完全值编号算法

聂久焘,程 旭,王克义
(北京大学信息科学技术学院,北京 100871)

摘 要: 值编号是一种重要的静态分析技术,广泛应用于优化编译器和程序验证工具.实际应用中的各种值编号算法在检测等值关系上都存在各种局限性.功能更加强大的能够检测全部 Herbrand 等值关系的完全值编号算法工作效率都十分低下而无法实用.我们发现采用静态单赋值形式能够大幅提高完全值编号算法的性能.本文基于 Herbrand 等值关系给出了静态单赋值形式的程序中值编号的一般定义,建立了值编号和 Herbrand 等值关系的对应关系.基于该定义,判断两个表达式之间的 Herbrand 等值关系等价于判断该两个表达式的值编号是否相同.之后给出了用于计算这种值编号的新的完全值编号算法.我们在 GCC 中实现了该算法并利用别名信息使其能够检测访存语句间的等值关系.基于新算法的部分冗余优化比 GCC 中原有算法消除了更多的动态冗余计算.

关键词: 值编号;编译优化;Herbrand 等值关系;冗余消除
中图分类号: TP314 **文献标识码:** A **文章编号:** 0372-2112 (2010) 02-0416-06

An Efficient Complete Global Value Numbering Algorithm

NIE Jiu-tao, CHENG Xu, WANG Ke-yi
(*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*)

Abstract: Global value numbering (GVN) is an important static analysis technique both for optimizing compilers and for program verification tools. Practically used GVN algorithms all have some restrictions on detecting equivalences. The traditional more powerful complete GVN algorithms that can discover all Herbrand equivalences are all inefficient and cannot be used in practice. We find that the static single assignment (SSA) form can be adopted to improve the efficiency greately. In this paper, we define a kind of value number of expressions of programs in SSA form based on Herbrand semantics. With this definition, determining whether two expressions are Herbrand equivalent are equivalent to determining whether their value numbers are same. Then, we give a new complete GVN algorithm that can compute such kind of value numbers. We also implement the new algorithm in GCC and extend it to be able to handle memory operations. Based on the new GVN algorithm, the partial redundancy elimination (PRE) optimization eliminates more redundant computations.

Key words: complete global value numbering; compiler optimization; Herbrande quivalence; redundancy elimination

1 引言

全局值编号(Global Value Numbering,GVN)是一种重要的静态程序分析技术.它用来检测程序中表达式间的等值关系,具有广泛的应用价值.优化编译器使用等值信息检测并消除语义冗余的计算和确定分支语句跳转目标并消除无用的分支语句.程序验证工具使用等值信息验证程序中的断言.变换验证工具使用等值信息验证程序变换的正确性.

由于检测程序表达式间一般的等值关系是不可判定的,大部分 GVN 算法都将问题做了简化,通常假设条件语句的结果在编译期间是不确定的,并且对所有的运算符都不考虑其特殊语义,即忽略它们可能满足的特殊运算法则,将不同结构的表达式看作不同的表达式.满足这些限制条件的表达式间的等值关系被称作 Herbrand 等值关系[1].能够检测到程序中全部 Herbrand 等值关系的 GVN 算法被称为完全 GVN 算法.

到目前为止,实际应用的 GVN 算法都是非完全的.应用最广泛的 GVN 算法是基于表达式散列的算法[2~4],已经被许多工业级编译器所采用,包括 GCC、Open64 和 LLVM 等.如今实际应用的基于散列的 GVN 算法通常都工作在静态单赋值(SSA)形式的程序上.它们在遍历控制流图的支配关系树的过程中使用一个散列表来保存第一次遇到的结构不同的表达式.当再次遇到结构相同的表达式时则将它们标记为相同的值编号,表示它们属于同一个等价类.由于这类算法只需要遍历

收稿日期:2008-09-26;修回日期:2009-08-12
基金项目:国家 863 高技术研究发展计划(No.2006AA010202)

程序一次,它们的执行效率通常都很高.然而,当程序中存在循环和分支结构时,该类算法可能会遗漏许多类型的等值关系.关于这个问题,文献[4,4.1 节]做了较详细的分析.为了解决这一问题,Alpern,Wegman 和 Zadeck(AWZ)提出了一种划分细化(partition refinement)算法,同样基于 SSA 形式.该算法首先假设所有具有相同根运算符的表达式都是相等的,并将它们放在同一个划分块中.然后,在每一个划分块中根据表达式的操作数进行进一步的划分,将操作数属于相同划分块的表达式划分到同一个细化的划分块中.如此迭代进行,直到所有的划分块都不能在被进一步划分为止.由于该算法不依赖于语句之间的控制流顺序,它们能够检测到控制流汇合处的 phi 节点之间的等值关系,比基于散列的算法有所改进.Simpson 等人融合了基于散列的算法和划分细化算法的优点,提出了一种基于 SSA 图上的强连通团(Strongly Connected Components,SCC)的 GVN 算法[4,第四章].该算法以 Tarjan 的 SCC 检测算法为基础,对输入的 SSA 图中检测到的每个 SCC 使用类似划分细化的迭代算法为每个 SSA 图节点赋予值编号.由于 Tarjan 的 SCC 检测算法按照后深度优先顺序检测每一个 SCC,当迭代计算一个 SCC 中的值编号时,该 SCC 中的节点所引用的节点一定已经被赋予了值编号.因此,该算法只需在每个 SCC 上而不是整个程序上使用迭代算法,所以它同样具有较高的执行效率.然而,所有前面介绍的算法都有共同的一个缺陷:它们不能检测 phi 节点和普通计算语句之间的等值关系.原因是它们都将 phi 节点看做不可解释的普通运算符,即假设一个 phi 节点 $\varphi(a,b)$不可能与一个普通表达式 $c+d$ 相等.

比如考察图 1 中的程序,我们可以观察到所有标记

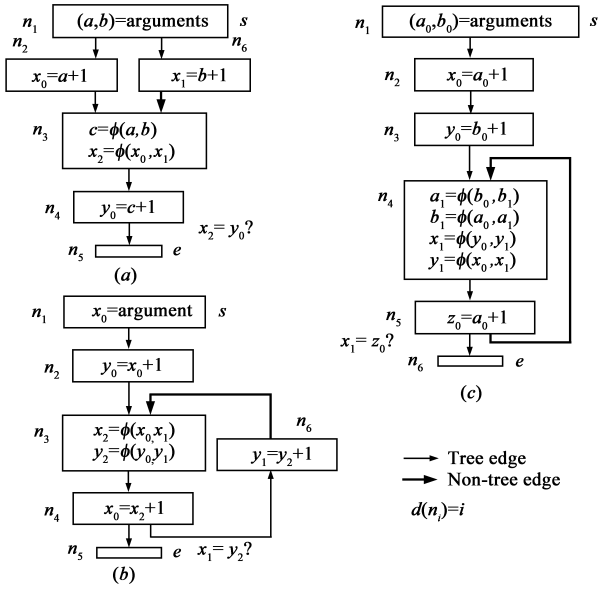

图1　GVN示例代码

“?”的等式在程序所有控制流上都是成立的.然而,所有前面介绍的算法都不能检测到这些等值关系.原因是这些等式的一边是 phi 节点的结果,而另一边则是普通表达式的计算结果.Rüthing,Knoop 和 Steffen(RKS)在 AWZ-算法的基础上增加了对 phi 节点的改写规则来改善这个问题[1].通过对某些模式的 phi 节点结构进行改写,改进的算法能够处理图 1 中(a)和(b)两种情况.但是对于(c),该算法仍无法处理.原因是程序节点 n_4 中的两对 phi 节点的相互依赖形成了回路,无法用改写规则进行变换.交替运行 AWZ 算法和重写规则也使得该算法效率低下而无法实用.实际上该算法即使对没有循环的程序也是非完全的[5].

完全 GVN 算法比所有非完全算法检测等值关系的能力都要更加强大.但是,尽管各种传统的完全 GVN 算法已经有了较长的历史,它们之中还没有一个能够被实际应用.这是因为现有的完全 GVN 算法运行效率都十分低下.最早的完全 GVN 算法在程序的每条语句前后都附加一个表达式的划分块集合,处于同一集合的表达式具有相同的值编号.算法对程序进行抽象执行,对每个划分块进行细化,最终得到完全的 Herbrand 等值关系.文献[6]中的算法使用一种称为 structured partition DAG(SPDAG)的数据结构来压缩表示划分块集合,大大减小了划分块集合本身的大小.但是,由于该算法仍然需要在每个语句的前后都保存一个 SPDAG,它的运行效率仍然非常低下.

我们在文献[7]中提出了一种基于 SSA 的高效的完全 GVN 算法,但该算法及其正确性的证明依赖于一种完全的非 SSA 算法,没有给出在 SSA 形式的程序中相应的理论基础.本文工作基于 Herbrand 等值关系直接给出了 SSA 形式的程序中值编号的一般定义,建立了值编号和 Herbrand 等值关系的对应关系.基于该定义,判断两个表达式之间的 Herbrand 等值关系等价于判断该两个表达式的值编号是否相同.因此只需要计算得到符合该定义的每个表达式的值编号即可得到所有表达式之间的 Herbrand 等值关系.以往文献中出现的各种值编号并不满足该性质.也就是说,基于以往的各种值编号定义,当两个表达式的值编号不同时,并不能确定它们不是 Herbrand 等值的,即无法保证完全性.我们重新设计了新的能够计算符合该定义的全局值编号算法.除此之外,我们还在 GCC 中实现了基于该算法部分冗余消除优化,并给出了新算法消除冗余计算的改进效果数据.

2　Herbrand 等值关系

我们用 V,O 和 T 分别表示变量集合、运算符集合和由变量和运算符构造出来的所有表达式集合.为了

简化概念,本文中假设所有运算符都是二元的,并且将常数也看作在开始节中点被赋值的变量.在 Herbrand 语义模型中,每个变量保存一个值,而值则由 T 中的一个表达式来表示.一个 Herbrand 状态被定义为一个由变量集合 V 到表达式集合 T 的函数 $\sigma: V\to T$. $\Sigma=\{\sigma\mid\sigma: V\to T\}$ 为由所有 Herbrand 状态组成的集合. σ_0 表示初始状态,即 V 上的恒等函数 $\forall x\in V.\ \sigma_0(x)=x$.一个表达式 $t\in T$ 在一个 Herbrand 状态 $\sigma\in\Sigma$ 下的 Herbrand 值被递归定以为一个函数 $\mathscr{H}: T\to(\Sigma\to T)$:

$$\mathscr{H}(t)(\sigma)=\begin{cases}\sigma(x), & (t\equiv x\in V)\\ o(\mathscr{H}(t_1)(\sigma),\mathscr{H}(t_2)(\sigma)), & (t\equiv o(t_1,t_2))\end{cases}$$

本文中的模型和算法都基于静态单赋值形式的程序.在一个静态单赋值形式的程序中,所有变量都有唯一的定值语句,并且所有对变量的使用都被该变量的定值语句所支配,即从程序的入口到达对该变量的使用的所有执行路径都一定经过该变量的定值语句.我们将静态单赋值形式的程序表示为一个有向的流图 $G=(N,E,s,e)$.其中 N、E、s 和 e 分别表示程序节点集合、边集合、唯一的开始节点和唯一的结束节点.每个 $N-\{e\}$ 中的节点可以表示如下三类不同的语句(其中 $x,y,z,x_i,x_{ij}\in V$ 表示任意静态单赋值的变量, $o\in O$ 表示任意二元运算符):

(1)赋值语句: $x=t$,其中 $t\equiv y$ 或 $t\equiv o(y,z)$.

(2)由在同一合并节点(假设有 $m>1$ 条入边)处的所有 *phi* 节点组成的 *phi* 节点集合(可能为空集): $\{x_1=\phi(x_{11},\cdots,x_{1m}),\cdots,x_k=\phi(x_{k1},\cdots,x_{km})\}$.(之所以将在同一合并节点处的所有 phi 节点的集合看作同一个程序节点是因为这些 phi 节点被定义为一个并行赋值语句 $(x_1,\cdots,x_k)=(\phi(x_{11},\cdots,x_{1m}),\cdots,\phi(x_{k1},\cdots,x_{km}))$.

(3)产生全局副作用或依赖全局状态的语句,比如 load、store、函数调用、跳转语句以及用来初始化参数的开始节点 s.通常我们用 $x=f(\cdots)$ 或 $f(\cdots)$ 表示这类程序节点.

我们用 N_a, N_ϕ 和 N_s 分别表示赋值语句集合、合并节点集合和全局语句集合.每个程序节点 $n\in N$ 的语义被定义为一个状态转换函数 $\theta_n:[1,\mid pred(n)\mid]\to(\Sigma\to\Sigma)$:

$$\theta_n(i)(\sigma)=\begin{cases}\sigma[\mathscr{H}(\sigma)(t)/x], & \text{if } n\in N_a\\ \sigma[\sigma(x_{1i}),\cdots,\sigma(x_{ki})/x_1,\cdots,x_k], & \text{if } n\in N_\varphi\\ \sigma, & \text{otherwise}\end{cases}$$

其中 $\mid pred(n)\mid$ 表示节点 n 的前驱节点数, $\sigma[t_1,\cdots,t_l/x_1,\cdots,x_l]$ 表示一个将 σ 中变量 x_i 的值替换为 t_i ($i\in[1,l]$)的新的 Herbrand 状态.状态转换函数的定义可以被扩展到程序 G 上的任意有限路径 $p=(n_1,\cdots,n_q)$:

$$\Theta_p(i)(\sigma)=\begin{cases}\theta_{n_1}(i)(\sigma), & \text{if } q=1\\ \Theta_{(n_2,\cdots,n_q)}(j)(\theta_{n_1}(i)(\sigma)), & \text{if } q>1\end{cases}$$

其中假设 n_1 为 n_2 的第 j 个前驱.

我们用符号 $\mathscr{P}[n_1,n_2]$ 表示在 G 上由程序节点 n_1 到程序节点 n_2 的所有有限路径组成的集合. $\mathscr{P}=\bigcup_{n\in N}\mathscr{P}[s,n]$ 表示 G 上以开始节点 s 为起点的所有路径集合.符号";"表示路径的连接运算.现在,SSA 形式的程序上的 Herbrand 等值关系定义如下:

定义 1(Herbrand 等值关系) 对于任意路径集合 $P\subseteq\mathscr{P}$,任意表达式 $t_1,t_2\in T$ 和任意程序节点 $n\in N$: t_1 和 t_2 在节点 n 处为局部 P-Herbrand 等值的当且仅当对于所有控制流图路径 $p\in\mathscr{P}[s,n]\cap P$, $\mathscr{H}(t_1)(\Theta_p(1)(\sigma_0))=\mathscr{H}(t_2)(\Theta_p(1)(\sigma_0))$.当 $P=\mathscr{P}$,我们省略"$P-$".

比如对于图 1(a)中的程序片段,令路径集合 $P_1=\{p_1=(n_1,n_2,n_3,n_4)\}$, $P_2=\{p_2=(n_1,n_6,n_3,n_4)\}$, $P_3=\{p_1=(n_1,n_2,n_3,n_4),p_2=(n_1,n_6,n_3,n_4)\}$,我们考察表达式 x_2, y_0, $a+1$, $b+1$ 和 $c+1$ 之间关于这三个路径集合的 Herbrand 等值关系.按照每条语句的 Herbrand 语义,这五个表达式在沿路径 p_1 执行得到的抽象值分别为 $a+1$, $a+1$, $a+1$, $b+1$ 和 $a+1$.相应地,沿路径 p_2 执行的抽象值分别为 $b+1$, $b+1$, $a+1$, $b+1$ 和 $b+1$.根据 P-Herbrand 等值关系的定义,表达式 x_2, y_0, $a+1$ 和 $c+1$ 为 P_1-Herbrand 等值的;表达式 x_2, y_0, $b+1$ 和 $c+1$ 为 P_2-Herbrand 等值的;表达式 x_2, y_0 和 $c+1$ 为 P_3-Herbrand 等值的.我们看到,路径集合中的路径越多,对等值关系的限制条件也就越强.如果路径集合包含所有可能的控制流路径,则相应的等值关系在程序运行时将总是会被满足.我们算法的核心思想是:从一个容易得到的不完全的路径集合上的等值关系开始,通过不断合并不同路径集合的等值关系得到具有更多路径限制条件的细化的分析结果,直到所有可能的路径都被包含进去.这里定义的 P-Herbrand 等值关系为我们的算法提供了理论基础.下一节将介绍如何使用值编号来高效地表示 P-Herbrand 等值关系.

3 值编号的定义

给定一个路径集合 $P\subseteq\mathscr{P}$,对于任意程序节点 n_1 和 $n_2\in N$,我们定义 $n_1\,P-\mathrm{dom}\,n_2$ 当且仅当所有路径 $p\in\mathscr{P}[s,n_2]\cap P$ 都包含 n_1. $n_1\,P-\mathrm{sdom}\,n_2$ 当且仅当 $n_1\,P-\mathrm{dom}\,n_2\wedge\neg(n_2\,P-\mathrm{dom}\,n_1)$.这里符号"dom"表示支配(dominate),"sdom"表示严格支配(strictly dominate).对于一个表达式 $t\in T$,我们称 t 在节点 $n\in N$ 处为 P-available 当且仅当所有 t 中出现的变量 x 都满足 $Def(x)\,P-\mathrm{dom}\,n$,其中 $Def(x)$ 表示 x 的定义节点.

集合 $P-\mathscr{H}(t,n)=\{t'\in T\mid ?\ t'$ 和 t 在 n 处为 P-Herbrand 等值的$\}$为包含所有与表达式 t 在程序节点 n 处 P-Herbrand 等值的表达式集合.我们假设每个变量 $x\in V$ 都有一个唯一的整数标识 $id(x)$.现在我们定义表达式集合 $P-\mathscr{H}(t,n)$上的一个偏序关系 $P-\preccurlyeq$如下:对于任意表达式 t_1 和 $t_2\in P-\mathscr{H}(t,n)$,$t_1P-\preccurlyeq t_2$ 当且仅当它们满足下面的条件.

$$\begin{aligned}&t_1\equiv o(t_1',t_1'')\wedge t_2\equiv x\in V\\ \vee\ &t_1\equiv x\in V\wedge t_2\equiv y\in V\wedge(Def(x)P-sdomDef(y)\\ \vee\ &\neg(Def(y)P-sdomDef(x))\wedge id(x)\leqslant id(y))\\ \vee\ &t_1\equiv o(t_1',t_1'')\wedge t_2\equiv o(t_2',t_2'')\wedge t_1'P-\preccurlyeq t_2'\wedge t_1''P-\preccurlyeq t_2''\end{aligned}$$

非形式化地来说,对于任意两个表达式,包含运算符的表达式先于单个变量表达式;对于两个变量表达式,如果一个变量的定值语句支配另一个变量的定值语句,则前者先于后者,否则它们的先后关系由它们唯一的整数标识之间的大小关系确定;对于两个包含相同运算符的表达式,它们的先后顺序由它们相应操作数间的顺序决定.同样,对于这些概念,当 $P=\mathscr{P}$时我们省略前面的"$P-$".如果集合 S 在偏序关系 R 下的最小元素存在,我们用符号 $\min(R,S)$来表示.对于任意表达式 t_1 和 $t_2\in P-\mathscr{H}(t,n)$,我们总是能够构造一个表达式 $t_3\in P-\mathscr{H}(t,n)$满足 $t_3\ P-\preccurlyeq t_1$ 并且 $t_3\ P-\preccurlyeq t_2$.如果 $\mathscr{P}[s,n]\cap P\neq\varnothing$,则 $P-\mathscr{H}(t,n)$为有限集合,所以 $\min(P-\preccurlyeq,P-\mathscr{H}(t,n))$存在.否则,$P-\mathscr{H}(t,n)=T$ 为无限集合.对于 $P-\mathscr{H}(t,n)$为无限集合和空集合这两种特殊情况,我们定义 $\min(P-\preccurlyeq,T)=\top$ 和 $\min(P-\preccurlyeq,\phi))=\top$,且 $\forall t\in\top.\ o(\top,t)=o(t,\top)=o(\top,\top)=\top$.现在我们定义表达式的值编号如下:

定义 2(值编号)　一个表达式 $t\in T$ 关于一个路径集合 $P\subseteq\mathscr{P}$的值编号为一个递归定义的表达式:

$$P-\mathscr{V}(t)=\begin{cases}\min(P-\preccurlyeq,P-\mathscr{H}(x,Def(x))),&\text{if } t\equiv x\in V\\ o(P-\mathscr{V}(t_1),P-\mathscr{V}(t_2)),&\text{if } t\equiv o(t_1,t_2)\end{cases}$$

同样,当 $P=\mathscr{P}$时省略"$P-$".

直观地说,一个包含运算符的表达式的值编号是由其根运算符和它的子表达式的值编号构造出来的;一个变量的值编号是在该变量定值节点处与之等值的表达式集合中关于偏序关系 $P-\preccurlyeq$的最小的表达式.比如,在图 1(a)中表达式 c 的值编号为 c 自己,因为在其定值节点 n_3 处没有其它表达式与它相等.表达式 x_2 的值编号为 $c+1$,因为 $c+1$ 与 x_2 在 n_3 处 Herbrand 相等,并且 $c+1\leqslant x_2$.同样,表达式 y_0 的值编号也为 $c+1$.这样定义的值编号满足一个重要的性质:两个表达式 t_1 和 t_2 在程序节点 n 处为 Herbrand 等值的当且仅当它们的值编号相等,即 $V(t_1)=V(t_2)$.由值编号的定义可以发现,只要得到每个变量的值编号,则所有复杂表达式的值编号就都可以很容易被构造出来,从而也就得到了所有的 Herbrand 等值关系.下一节将介绍为每个变量计算满足该定义的值编号的算法.

4　计算值编号的算法

图 2 为计算所有变量值编号的算法.算法结束后,表达式数组 v_1 包含了所有变量的值编号,即 v_1 满足 $\forall x\in V.\ v_1[x]=V(x)$.下面列出了算法中用到的一些符号和数据结构:

```
1  GVN()
2  begin
3      for x ∈ V do v1[x] := ⊤
4      W2 := {⟨n,1⟩ | n ≡ x = f(...) ∈ Ns}
5      repeat
6          v2 := v1 ;              // Copy array v1 to v2
7          W1 := W2; W2 := ∅
8          while W1 ≠ ∅ do
9              ⟨n,i⟩ := min(⊴, W1); W1 := W1 − {⟨n,i⟩}
10             AbstractExec(n, i)
11             for x ∈ V such that v1[x] is changed
               during calling AbstractExec(n, i) do
12                 for ⟨n',i⟩ such that n' ∈ Na ∪ Nφ uses
                   x through its i-th incoming edge do
13                     if i = 1 then W1 := W1 ∪ {⟨n',i⟩}
14                     else W2 := W2 ∪ {⟨n',i⟩}
15     until W2 = ∅
16 end
17 AbstractExec(n, i)
18 begin
19     if n ≡ x = o(y,z) ∈ Na then
       v1[x] := o(v1[y], v1[z])
20     else if n ∈ Nφ then
21         (vl, vr) := if i = 1 then (v2, v1) else (v1, v2)
22         for x = φ(x1, ..., xi, ..., xm) ∈ n do
23             v1[x] :=Intersect(vl[x], vr[xi], n, i)
24     else if n ≡ x = f(...) ∈ Ns then v1[x] := x
25 end
26 Intersect(tl, tr, n, i)
27 begin
28     if tl ≡ x ∈ V and vr[x] = tr then return x
29     else if tr ≡ x ∈ V and vl[x] = tl then return x
30     else if tl ≡ o(tl1, tl2) and tr ≡ o(tr1, tr2) then
31         t1 := Intersect(tl1, tr1, n, i)
32         t2 := Intersect(tl2, tr2, n, i)
33         if t1 ≠ ⊥ and t2 ≠ ⊥ then return o(t1, t2)
34     A := {x | x = φ(x1, ..., xi, ..., xm) ∈ n ∧ vl[x] =
       tl ∧ vr[xi] = tr}
35     return min(≼, A);              // min(≼, ∅) = ⊥
36 end
```

图2　全局值编号算法

●v_1,v_2,v_l 和 v_r 为表达式数组,用于在算法运行过程中保存每个变量的部分路径值编号 $P-\upsilon(P\subseteq\mathscr{P})$.

●DFT 为 G 的一棵以 s 为根节点的深度优先搜索(depth-first-search,DFS)树.

●$d(n)$为节点 $n\in N$ 在深度优先搜索过程中第一次到达的次序,即 n 是第 $d(n)$个到达的节点.在图 1 中,$d(n_i)=i$.

●W_1 和 W_2 为包含序偶$\langle n,i\rangle$的集合,其中 $n\in N$

是将要被抽象执行的程序节点,i 是一个正整数,指出抽象执行过程将从 n 的哪条入边到达 n.为了描述方便,我们将节点所有入边中唯一的一条 DFT 边编号为 1,其他所有边编号为大于 1 的值.这样我们只需要根据序偶$\langle n,i\rangle$中的 i 值是否为 1 就能够判断抽象执行是否从一条 DFT 边进入节点 n 了.

●$\triangle$是由所有序偶$\langle n,i\rangle$组成的序偶集合上的一个全序关系.任意两个序偶$\langle n_1,i_1\rangle \triangle \langle n_2,i_2\rangle$当且仅当 $d(n_1)<d(n_2)\vee d(n_1)=d(n_2)\wedge i_1\leqslant i_2$.这个顺序用于保证算法总是先处理 DFT 的祖先节点,并且对于同一节点的多条入边则总是先处理 DFT 上的边.

由于不同的值编号之间有大量相同的子结构,即大量共享子表达式,在实现中,我们可以用一个有向无环图来紧凑地表示所有的值编号.而且,我们可以使用一个列表来保存值编号的更新信息,而不需要每次都复制整个数组.现在,回到图 1 中的程序,将我们的算法应用在(a)中程序片段的详细步骤如下:

(1)初始化:$v_1=\{\top,\cdots,\top\}$, $W_2=\{\langle n_1,1\rangle\}$.

(2)$v_2:=v_1$; $W_1:=W_2$; $W_2=\phi$: $v_1=v_2=\{\top,\cdots,\top\}$, $W_1=\{\langle n_1,1\rangle\}$, $W_2=\phi$

(3)AbstractExec$(n_1,1)$: $v_1[a]=a$, $v_1[b]=b$, $v_1[1]=1$, $W_1=\{\langle n_2,1\rangle,\langle n_3,1\rangle,\langle n_6,1\rangle\}$, $W_2=\{\langle n_3,2\rangle\}$

(4)AbstractExec$(n_2,1)$: $v_1[x_0]=v_1[a]+v_1[1]=a+1$, $W_1=\{\langle n_3,1\rangle,\langle n_6,1\rangle\}$, $W_2=\{\langle n_3,2\rangle\}$

(5)AbstractExec$(n_3,1)$: $v_1[c]=$Intersect$(\top,a,n_3,1)=a$, $v_1[x_2]=$Intersect$(\top+\top,a+1,n_3,1)=$Intersect$(\top,a,n_3,1)+$Intersect$(\top,1,n_3,1)=a+1$, $W_1=\{\langle n_4,1\rangle,\langle n_6,1\rangle\}$, $W_2=\{\langle n_3,2\rangle\}$

(6)AbstractExec$(n_4,1)$: $v_1[y_0]=v_1[c]+v_1[1]=a+1$, $W_1=\{\langle n_6,1\rangle\}$, $W_2=\{\langle n_3,2\rangle\}$

(7)AbstractExec$(n_6,1)$: $v_1[x_1]=v_1[b]+v_1[1]=b+1$, $W_1=\phi$, $W_2=\{\langle n_3,2\rangle\}$

(8)$v_2:=v_1$; $W_1:=W_2$; $W_2=\phi$: $v_1=v_2=\{a:a,b:b,1:1,x_0:a+1,c:a,x_2:a+1,y_0:a+1,x_1:b+1\}$, $W_1=\{\langle n_3,2\rangle\}$, $W_2=\phi$

(9)AbstractExec$(n_3,2)$: $v_1[c]=$Intersect$(a,b,n_3,2)=\min(\leqslant,\{c\})=c$, $v_1[x_2]=$Intersect$(a+1,b+1,n_3,2)=$Intersect$(a,b,n_3,2)+$Intersect$(1,1,n_3,2)=\min(\leqslant,\{c\})+1=c+1$, $W_1=\{\langle n_4,1\rangle\}$, $W_2=\phi$

(10)AbstractExec$(n_4,1)$: $v_1[y_0]=v_1[c]+v_1[1]=c+1$, $W_1=W_2=\phi$

(11)算法终止,我们得到计算结果 $v_1=\{a:a,b:b,1:1,x_0:a+1,c:c,x_2:c+1,y_0:c+1,x_1:b+1\}$.注意,$v_1[x_2]=v_1[y_0]=c+1$,意味着变量 x_2 和 y_0 满足 Herbrand 等值关系.

对于(b)中的程序:在 GVN()过程中的循环第一次执行结束后,$v_1=\{x_0:x_0,y_0:x_0+1,x_2:x_0,y_2:x_0+1,x_1:x_0+1,y_1:(x_0+1)+1\}$, $W_2=\{\langle n_3,2\rangle\}$.该循环第二次执行结束后,$v_1=\{x_0:x_0,y_0:x_0+1,x_2:x_2,y_2:x_2+1,x_1:x_2+1,y_1:(x_2+1)+1\}$, $W_2=\phi$,算法结束.从中我们得知 x_1 和 y_2 为 Herbrand 等值的,因为它们的值编号满足 $v_1[x_1]=v_1[y_2]=x_2+1$.

对于程序(c):第一次循环后,$v_1=\{a_0:a_0,b_0:b_0,x_0:a_0+1,y_0:b_0+1,a_1:b_0,b_1:a_0,x_1:b_0+1,y_1:a_0+1,z_0:b_0+1\}$, $W_2=\{\langle n_4,2\rangle\}$.第二次循环执行结束后,$v_1=\{a_0:a_0,b_0:b_0,x_0:a_0+1,y_0:b_0+1,a_1:a_1,b_1:b_1,x_1:a_1+1,y_1:b_1+1,z_0:a_1+1\}$, $W_2=\{\langle n_4,2\rangle\}$.第三次循环执行结束后,$v_1$ 不再发生变化,所以 $W_2=\phi$,算法结束.由 $v_1[x_1]=v_1[z_0]=a_1+1$ 推出 x_1 与 z_0 为 Herbrand 等值的.

5 算法实现和实验结果

我们在 GCC-4.2 中实现了新的 GVN 算法.利用 GCC 中的存储静态单赋值形式(Memory SSA),我们的算法还能够检测存储访问语句之间的等值关系.GVN 算法本身并不做任何代码变换,它只分析程序中的等值关系.为了测试新的 GVN 算法是否能够帮助编译器消除更多的冗余计算,我们实现了一个能够利用新 GVN 算法分析结果的部分冗余消除优化,称之为最优语义

表 1 GVNPRE 和 OSCM 消除动态冗余计算百分比对比

CINT2000	gzip	vpr	gcc	mcf	crafty	parser	gap	bzip2	twolf
GVNPRE	1.05	8.52	2.67	0.15	1.21	0.14	1.61	1.11	1.94
OSCM	9.62	16.41	3.21	14.00	0.56	1.69	2.12	3.56	2.25
Improved	8.57	7.89	0.54	13.85	−0.65	1.55	0.51	2.45	0.31

CFP2000	wupwise	swim	mgrid	applu	mesa	art	equake	facerec	ammp	lucas	fma3d	sixtrack	apsi
GVNPRE	10.96	16.68	46.08	42.88	0.54	8.77	7.75	28.79	0.22	4.74	1.03	1.52	33.30
OSCM	12.01	21.59	53.33	48.44	0.54	10.58	7.78	28.82	0.66	6.53	3.82	3.75	34.15
Improved	1.05	4.91	7.25	5.56	0.00	1.81	0.03	0.03	0.44	1.79	2.79	2.23	0.85

代码移动(Optimal Semantic Code Motion,OSCM).GCC-4.2 中使用的部分冗余消除算法为文献[3]中的 GVNPRE 算法,使用一种集成的基于散列的 GVN 算法分析表达式间的等值关系.我们使用 SPEC CPU2000 中的测试程序统计了两种部分冗余优化所消除的动态冗余计算数量.表 1 为详细统计数据对比.从中我们看到,采用了新的 GVN 算法的部分冗余消除优化比采用基于散列的 GVN 算法的部分冗余消除优化消除了更多的动态冗余计算.这说明完全 GVN 算法不仅在理论上优于非完全的 GVN 算法,在实际应用中也有很高的实用价值.

6　结论

传统的能够检测程序中全部 Herbrand 等值关系的完全 GVN 算法由于运行效率低下而无法被实际应用.本文基于 Herbrand 等值关系给出了静态单赋值形式的程序中值编号的一般定义,建立了值编号和 Herbrand 等值关系的对应关系.基于该定义,判断两个表达式之间的 Herbrand 等值关系等价于判断该两个表达式的值编号是否相同.在此基础上,给出了用于计算这种值编号的新的完全值编号算法.该算法在 GCC 中实现,并利用 GCC 中的存储静态单赋值形式使其能够检测访存语句之间的等值关系.实验表明,基于新算法的部分冗余优化比 GCC 中原有算法消除了更多的动态冗余计算.

参考文献:

[1] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision[A]. SAS[C]. Berlin: Springer, 1999. 232 – 247.

[2] Cliff Click. Global code motion/global value mumbering[A]. PLDI[C]. New York: ACM, 1995. 246 – 257.

[3] Thomas John VanDrunen. Partial redundancy elimination for global value numbering[D]. West Lafayette: Purdue University, 2004.

[4] Loren Taylor Simpson. Value-Driven Redundancy Elimination [D]. Houston: Rice University, 1996.

[5] Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering [A]. SAS [C]. Berlin: Springer, 2004. 212 – 227.

[6] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph: A program representation for optimal program transformations[A]. ESOP[C]. Berlin: Springer, 1990. 389 – 405.

[7] Jiu-Tao Nie and Xu Cheng. An Efficient SSA-Based Algorithm for Complete Global Value Numbering [A]. APLAS [C]. Berlin: Springer, 2007. 319 – 334.

[8] Karthik Gargi. A sparse algorithm for predicated global value numbering[A]. PLDI[C]. New York: ACM, 2002. 45 – 56.

[9] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation[A]. POPL[C]. New York: ACM, 2004. 342 – 352.

作者简介:

聂久焘　男,2002 年 7 月毕业于北京理工大学计算机科学与工程系,获工学学士学位.现为北京大学信息科学技术学院微处理器研发中心博士研究生,研究方向为编译优化技术、计算机体系结构和程序设计语言.
E-mail: njt@mprc.pku.edu.cn

程　旭　男,北京大学信息科学技术学院教授,博士生导师.主要研究方向包括:高性能微处理器、系统芯片、嵌入式系统、指令级并行、软硬件协同设计以及编译优化技术等.

王克义　男,1970 年本科毕业于北京大学,获学士学位,现为北京大学信息科学技术学院教授,博士生导师.目前研究兴趣为高性能微处理器.