



冗余表达的检测。

一个完整的、多项式时间的SSA算法

雷卡-R. 帕伊^{B)}

印度喀拉拉邦卡利卡特国家技术学院，卡利卡特，印度
rekhamapai@nitc.ac.in

摘要。检测程序中基于数值的冗余表达式是一个经过深入研究的问题，目的是为了消除冗余，从而提高程序的运行时间的有效性。这个问题需要检测程序中的等价表达。在这里，我们提出了一种迭代数据流分析算法，以检测SSA中的等价表达，从而达到检测冗余的目的。这种静态分析的核心挑战是确定一个 "连接" 操作，以检测连接点的所有等价物，从而在多项式时间内检测到任何后来出现的冗余表达。我们通过引入 *数值 φ -函数* 的概念来实现这一目标。我们声称该算法是完整的，只需要多项式时间。我们在LLVM中实现了该算法，并展示了其性能。


关键词等价检测--全局数值编号--冗余检测--数值 φ -函数

1 简介

消除程序中基于数值的冗余表达式是一项重要的代码优化工作，目的是提高程序的运行时间效率。这里的基本问题是检测程序中的等价表达。对程序中所有等价关系的检测是不可判定的，因此我们只关注Herbrand等价关系的检测[8]，这是传统的做法。如果两个表达式有相同的运算符，并且相应的操作数是Herbrand等价的，那么它们就是*Herbrand等价的*。

通过给每个表达式分配*数值*来检测等价性。如果检测到两个表达式是等价的，则为它们分配值号 v_i [3]。全局数值编号（GVN）是指为表达式分配数值以检测整个程序中的等价性的问题。文献中提出的GVN算法是完整的，也是有效的。如果一个GVN算法能够检测到所有的Herbrand等价物，并且所有相关的总冗余都被检测到，那么这个算法就是 "完整的"。

目前的GVN算法要么是完整的[5]，要么只需要多项式时间[2-4,6,8-10]，但在检测冗余的情况下，两者都不是。正如在一个

 Springer International Publishing Switzerland 2015
X.Feng and S. Park (Eds.): APLAS 2015, LNCS 9458, pp.49-65, 2015.
doi: 10.1007/978-3-319-26529-2_4-

在数据流分析中，GVN的核心挑战是设计一个 "连接" 操作来检测连接点上的等价关系。尽管在连接点上检测所有的等价物使GVN算法变得完整，但它扩大了等价表达式的分区大小，从而使算法变得不科学[5]。

为了使GVN算法成为多项式，一个解决方案是只检测那些在某一点上的等价物 e' ，这些等价物以后可能会在某个表达式（比如说 e ）出现的地方使用。在这里，我们从一个不同的角度来看待这个解决方案。我们建议，在程序的某一点 p 上给定一个表达式 e ，检测 e 是否等同于在 e 的路径上出现的一些表达式 e' ，而不是检测以后可能使用的等价关系。为此，我们使用静态单一赋值（SSA）形式的 φ -函数的语义，并引入了价值 φ -函数的新概念，这是一组等同的 φ -函数。然后，我们提出了一种迭代数据流分析算法，以检测SSA形式的程序的等价性，该算法是完整的，只需要多项式时间。我们随后证明了该算法的合理性和完整性。

我们在LLVM中实现了所提出的算法以及Kildall[5]和Gulwani[4]的算法，以证实我们对完整性和efficiency的主张。使用这三种算法对SPEC2006程序进行了分析，实验结果表明，所提出的算法是完整的，因为它检测的冗余数量与Kildall的完整算法相同。与广泛接受的Gulwani的多项式时间算法相比，所提出的算法也是有效的，因为它分析SPEC2006程序所需的时间更短。

本文的其余部分组织如下：在第2节中，我们分析了两种经典的GVN算法，以清楚地了解全局值编号的问题。本文中使用的术语在第3节中给出。3.第3节描述了 φ 函数和新算法。4.第5节对该算法进行了正式定义，第6节对我们的算法与Kildall[5]和Gulwani[4]的算法进行了实验比较。在第7节中，我们回顾了文献中的一些算法。第8节总结了这项工作。

2 激励

在这一节中，我们分析了Kildall[5]和Gulwani[4]的经典作品，以了解等价表达的检测问题。Kildall的算法是完整的，Gulwani的算法只需要多项式时间。

2.1 基尔德尔的算法

Kildall的迭代数据流分析算法在程序的每个点上检测等价关系。这些等价表示为将表达式划分为等价类，称为表达式池。该算法在其转移功能中使用了一个强大的概念，即 "结构化"。当一个新的等价类在与程序中的表达式 e 相对应的表达式池中创建时，该算法通过结构化的方式对分区进行分割

并在新的类中加入与 e 等价的所有表达式（Herbrand）。这确保了对所有冗余表达式的检测，这意味着该算法是完整的。但是这将导致等价类的大小呈指数级增长。如[5]中的“实施说明”部分所述，使用值数可以避免这个问题。Kildall使用*价值表达式*，这是一组等价表达式的紧凑表示[5,9]，使等价类的大小呈线性。但如[4]所示，由于连接操作的定义，表达式池的大小（以等价类的数量表示）呈指数级增长的问题仍然存在。这是因为应用于 n 个输入池的连接操作可能导致表达池的大小与输入池的大小成指数关系[4]。

2.2 古尔瓦尼的算法

这种算法的工作原理类似于Kildall的算法，其等价信息表示为一个有向图，即*强等价DAG*（SED）。SED提供了一个分区中等价类的紧凑表示。该算法检测所有大小最多为 s 的表达式之间的等价关系，其中 s 是程序表达式的大小。这减少了通过连接操作计算的分区中的等价类的数量（与Kildall的算法相比），这使得它只需要多项式时间。Gulwani定义的连接操作（见第3.5节，连接算法，[4]中的第3-5行）只有当类至少有一个共同的变量时才会相交。这导致了在检测一些等价物时的缺失，而这些等价物在检测冗余时是有用的[9]。

2.3 我们的观点

GVN的核心问题是定义一个连接操作来检测连接点上的等价关系。检测一个点上的所有等价表达式会使算法成为指数级的。为了克服这个问题，一个解决方案是在一个点上只检测那些用于检测后来出现的冗余表达的等价表达。据我们所知，目前还没有任何方法可以精确预测这种冗余表达是否会出现。在这里，我们建议从一个完全不同的角度来看待这个问题。我们不是在以后使用的连接点检测等价的表达，而是将这种等价的检测推迟到一个表达实际出现的点。

3 术语

程序表示法。SSA中的程序被表示为控制流图（CFG）[1]，它有一个空的进入和退出块。其他块包含形式为 $x = e$ 的赋值语句，其中 e 是一个表达式。我们假设一个块最多可以有两个前例，一个恰好有两个前例的块被称为*连接块*。一个块的输入和输出点分别称为该块的*输入点*和*输出点*。

表达式。一个表达式可以是一个常数，一个变量，或者是 $x \oplus y$ 的形式，其中 x 和 y 是常数或变量， \oplus 是一个通用的二元运算符。一个表达式也可以是 $\phi_k(x, y)$ 的形式，其中 x 和 y 是变量， k

是它出现在其中的连接块。这样的表达式是 ϕ -函数。当连接块从上下文中明确时，我们可以省略下标 k 。在我们绘制的CFG中， ϕ -函数出现在连接块中。但为了清楚起见，我们假设 ϕ -函数被转化为复制语句¹，并被附加到连接块的适当的前者。

等价性。如果两个表达式 e_1 和 e_2 具有相同的操作符，并且相应的操作数是Herbrand等价的，则表示 $e_1 \equiv e_2$ 。如果路径 P 中的两个表达式 e_1 和 e_2 在该路径中是Herbrand等价的，则表示为 $e \equiv_{1P} e_2$ ，。

值表达式。一个价值表达式 $v_i \oplus v_j$ 表示两个等价类之间的操作，其中 v_i 和 v_j 是两个等价类的价值编号。 $v_i \oplus v_j = \{x \oplus y; x \in C_i, \text{ 等价类的价值编号为 } v_i, y \in C_j, \text{ 等价类的价值数 } v_j\}$ 。一个价值表达式是一个代表等价表达式集合的ive expression。一个表达式的价值表达通过用操作数替换操作数来构建 $x \oplus y$ 的概念。

4 基本概念

我们的目标是为冗余检测开发一个完整的、多项式时间的算法。冗余的原因是程序中表达式的等价性，因此冗余的检测可以说是CFG中每一点上表达式等价类的计算问题。这个问题可以正式表述为：给定某点的表达式 e ，检测在通往 p 的每条路径中是否有表达式 e' ，使得 e' 和 e 在该路径中是等同的。在这里，为了达到这个目的，引入了价值 ϕ -函数的概念。在这一节中，我们首先解释价值 ϕ -函数，然后提出我们的方法来检测冗余。

4.1 价值 ϕ -函数

考虑一下图1中的代码段。根据所采取的路径，表达式 $x_3 + 1$ 等同于 $x_1 + 1$ 或 $x_2 + 1$ 。换句话说，根据所走的路径，变量 w_3 等同于变量 y_1 和 z_2 中的一个。也就是说， w_3 可以被看作等同于“不同变量的合并”-- y_1 和 z_2 --在连接点，表示为 $\phi(y_1, z_2)$ 。这种“不同变量的合并”可以被看作是文献中 ϕ 函数的扩展形式²。我们用这种扩展的 ϕ 函数概念或“不同变量的合并”来表示

¹复制语句是一个形式为 $x = y$ 的赋值语句，其中 y 是一个变量。

²在文献中，一个 ϕ 函数将其操作数限制为同一非SSA变量的不同下标版本，例如 $\phi(x1, x2)$ 。

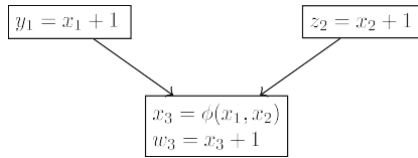


图1.带有分支的程序

在这种情况下表达等价关系。与价值表达的概念类似，我们将**价值 φ -函数**的概念定义为一组等价 **φ -函数**的抽象。

分区。在一个点上的分区代表了在通往该点的路径中存在的等价关系。分区中的一个等价类有一个值数和像变量、常数和值表达式这样的元素。它也被注解为有一个值

必要时，可以使用 **φ -函数**。例如，在一个分区 $\{----|v_r, x_i, y_i |v_s, z_i, v_r + 1|v_m, x_n : \varphi_k(v_i, v_j) |----\}$ 中，在一个点 p ，价值数为 v_r 的类代表变量 x_i 和 y_i 之间的等同性。价值数为 v_s 的类代表

由价值表达式 $v_r + 1$ 代表的表达式之间是等价的，即 $x_i + 1$ 和 $y_i + 1$ 。这些表达式也等同于变量 z_i 。从具有价值数 v_m 的类中，我们可以推断出，在通过左边连接块 k 到点 p 的路径中，变量 x_n 等同于具有价值数 v_i 的表达式。另外， x_n 等同于具有价值数 v_j 在到

p 通过右边缘到连接块。请注意，值 **φ -函数** $\varphi_k(v_i, v_j)$ 作为该类的最后一个元素出现，并用":"符号与其他元素分开，以表明值 **φ -函数**是该类的注释。

4.2 建议的方法

利用价值 **φ -函数**的概念，我们提出了一种迭代数据流分析算法，以计算程序中每个点的等价物。该算法的两个主要组成部分是**连接操作**和**转移函数**。

联接操作。连接操作检测的是通往连接点的所有路径中共同存在的等价关系。因为在SSA中，一个变量只有一个定义，所以在一个点 p 上成立的等价关系，支配着3个连接点 j ，在连接点上成立。这些等价物在连接点上是通过简单的类的交叉来检测的。然而，检测一些在分支中产生的共同等价物需要额外的处理，我们对后者进行说明。为了清楚起见，我们将检测变量之间的等价关系的情况与带操作符的表达式之间的等价关系分开。

³如果从入口点到 p 的所有路径 l ，那么CFG中的点 p 就支配着点 p^l 。

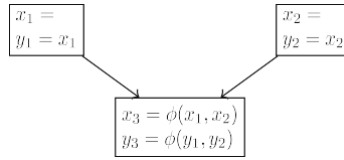


图2.检测变量的等价性

变量的等价性。考虑一下图2中的代码段。在通往连接点的左侧路径中，在左侧分支中定义的变量 x_1 和 y_1 是等价的。同样地，在右边分支中定义的 x_2 和 y_2 在右边路径中也是等价的。通过使用连接块中出现的 ϕ 函数，我们可以发现， x_1 在左路径中与 x_3 等价， x_2 在右路径中与 x_3 等价。同样地， y_1 在左边路径中与 y_3 等价， y_2 在右边路径中与 y_3 等价。根据等价关系的反证性，我们得出结论： x_3 和 y_3 在连接点是等价的。

换句话说，我们的连接操作合并了在不同分支中定义的两个变量（对应于同一个非SSA变量）-- x_1 与 x_2 ， y_1 与 y_2 ，在连接点分别得到 x_3 和 y_3 。然后我们得出结论，变量 x_3 和 y_3 在连接点是等价的；此外，它们与 ϕ -函数 $\phi(x_1, x_2)$ 和 $\phi(y_1, y_2)$ 是等价的。这种等价的检测是通过将 ϕ -函数转换为复制语句来完成的，然后将其追加到连接块的适当的前例。

带操作符的表达式的等价性。现在考虑图3中的代码段。出现在不同分支中的表达式 $x_1 + 1$ 和 $x_2 + 1$ 被合并⁴，在连接点得到 $x_3 + 1$ 。通过这种"合并"，我们检测到 $x_1 + 1$ （和 $x_2 + 1$ ）与 $x_3 + 1$ 的等价性，如果它出现在从连接点开始的某个路径的某一点。然而，如果 $x_3 + 1$ （或一些与之等价的表达式）没有出现，这种合并是可以避免的。

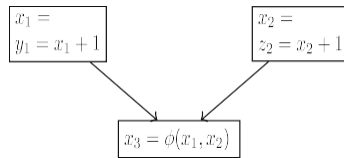


图3.检测带运算符的表达式的等价性

⁴表达式的合并可以被看作是变量合并的一个扩展概念。"表达式的合并" $ei_1 + ei_2$ 和 $ej_1 + ej_2$ 是表达式 $ei + ej$ ，这样 ei 是 ei_1 和 ej_1 的合并。同样地， ej 是 ei_2 和 ej_2 的合并。

正如第2节所讨论的, 问题是是否要在连接点合并不同的表达式以检测等价性。在这里, 我们采取了一个完全不同的应用, 连接操作并不合并连接点上的表达式。相反, 合并操作被推迟到 $x_3 + 1$ (或一些与之等价的表达式) 的出现。这将在解释转移函数的概念时讨论。

例子。考虑在分区上应用连接的情况 $P_1 = \{v_1, x_1, x_3 | v_2, y_1, y_3, v_1 + 1 | v_3, z_1, z_3\}$ 和 $P_2 = \{v_4, x_2, x_3 | v_5, y_2, y_3 | v_6, z_2, z_3, v_4 + 1\}$ 。⁴在 P_1 和 P_2 中具有价值数 v_1 的类中, 只有一个共同的变量

x_3 , 并且它出现在所产生的分区 P_3 的一个类中。由于 P_1 和 P_2 中的两个类分别有不同的值数 v_1 和 v_4 , 我们可以推断出 x_3 实际上是一个变量的合并。因此, 产生的类被注释为价值 φ -函数 $\varphi(v_1, v_4)$ 。该类被分配了一个新的数值, 例如 v_7 。结果类是 $|v_7, x_3 : \varphi(v_1, v_4)|$ 。

现在考虑在 P_1 中的值数 v_2 和在 P_2 中的值数 v_6 的类。有在这些类中没有明显的共同等价物; 但是我们可以从分区中推断出, P_1 中的二ff异值表达式 $v_1 + 1$ 和 P_2 中的 $v_4 + 1$ 实际上代表 $x_3 + 1$ 的共同等价物, 它是二ff异表达式 $5x_1 + 1$ 和 $x_2 + 1$ 的合并。但是如上所述, 不同的表达式 (或者准确地说不同的值表达式) 现在没有被合并, 因此在产生的分区 P_3 中没有创建新的类。

采用类似的策略来检测其他类对中的共同等价物, 从 P_1 和 P_2 中各选一个。结果分区 P_3 是 $\{v_7, x_3$ 。

$\varphi(v_1, v_4) | v_8, y_3 : \varphi(v_2, v_5) | v_9, z_3 : \varphi(v_3, v_6)\}$ 。

转移函数。基于在语句 $s : x=e$ 的输入点上存在的等价关系, 语句的转移函数确定了在其输出点上存在的等价关系。这涉及到检测表达式 e 是否与通往它的每条路径中的表达式 e 等价⁵。因此, 转移函数通过更新现有的类或创建一个新的类来计算出点的分区, 表示为 $POUT$ 。(从入点的分区, 表示为 PIN 。)。我们在这里设计的转移函数与文献中的函数类似, 只是它使用价值 φ 函数来检测通往 e 的每条路径中的等价关系。第一步是检查分区 PIN 。是否存在 e 的价值表达。如果没有找到, 转移函数就会继续检查表达 e 是否可以被表达为 "不同表达的合并"。下面用图4中的代码段来说明这一点。

考虑处理最后一条语句 $w_3 = x_3 + 1$ 。由于表达式 $x_3 + 1$ 的值表达式 $v_7 + 1$ 没有出现在 PIN_3 中, 转移函数继续检查 $x_3 + 1$ 是否可以被表达为表达式的合并, 如下所示。

⁵由于 x_3 是变量 x_1 和 x_2 的合并, 表达式 x_3+1 是 x_1+1 和 x_2+1 的合并。

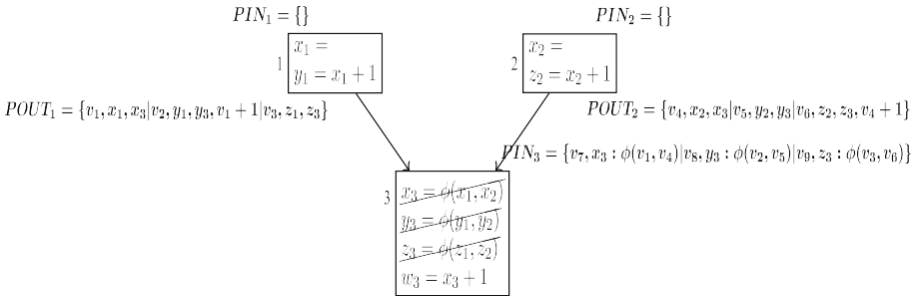


图4. 传递函数的概念

$x_3 + 1 \equiv v_7 + 1$ //使用PIN码计算出的x的值表达式 $v_7 + 1$
 $\equiv \phi(v_1, v_4) + 1$ //在PIN₃中有 v_7 的类被注释为 $\phi(v_1, v_4)$
 $\equiv \phi(v_1 + 1, v_4 + 1)$ //使用价值 ϕ -函数的语义
 $\equiv \phi(v_2, v_6)$ // $v_1 + 1 \in POUT_1$ 有 v_2 , 其中块1是连接块的左前辈, $v_4 + 1 \in POUT_2$ 有 v_6 , 其中块2是右前辈。
 。

因此, 使用价值 ϕ -函数, 我们发现 $x_3 + 1$ 实际上等同于价值数为 v_2 的表达式, 当左边的路径被单独考虑时。同样地, $x_3 + 1$ 等同于值数为 v_6 的表达式, 当右边的路径被单独考虑时。也就是说, $x_3 + 1$ 是一个“表达式的合并”, 这里是 $x_1 + 1$ 和 $x_2 + 1$ 。就变量而言, w_3 是不同变量的合并, 这里是 y_1 和 z_2 。

由于PIN₃中既没有价值表达式 $v_7 + 1$, 也没有价值 ϕ 函数 $\phi(v_2, v_6)$, 转移函数在POUT₃中创建了一个具有新价值编号的新类, 例如 v_{10} , 并将 w_3 和 $v_7 + 1$ 加入其中。该类也被注释为价值 ϕ -函数 $\phi(v_2, v_6)$ 。PIN₃中的类被如实添加到POUT₃。由此产生的POUT₃分区是 $\{v_7, x_3 : \phi(v_1, v_4) | v_8, y_3 : \phi(v_2, v_5) | v_9, z_3 : \phi(v_3, v_6) | v_{10}, w_3 : v_7 + 1 : \phi(v_2, v_6)\}$ 。

冗余的检测。如果在语句 $x = e$ 中存在表达式 e' 与语句的每条路径中的 e 等价, 那么语句中的表达式 e 就是多余的。就变量而言, 这意味着 x 等同于某个变量 y , 而不考虑采取的路径或不同变量的合并。在语句 $x=e$ 的分区POUT中, 如果在POUT中 x 的类别中存在一个变量, 而不是 x , 或者POUT中 x 的类别被注释为 ϕ -函数, 那么表达式 e 就是多余的了。

在图4的示例代码中, 最后一条语句 $w_3 = x_3 + 1$ 中表达式 $x_3 + 1$ 的冗余被检测到, 因为POUT₃中 w_3 的类(在上一小节计算)被注释为一个值 ϕ -函数。

5 算法

在这里，我们正式提出了迭代数据流分析算法，以检测程序中每个点的等价性。该算法的两个主要组成部分是连接操作和转移函数，其定义见下文。检测冗余的算法是微不足道的，这里就不写了。

5.1 加盟

下面给出的 "连接" 算法定义了连接操作。在执行连接操作之前，连接块中的 φ -函数被转换为复制状态，并被附加到连接块的适当的前任上。然后，转移函数被应用于这些副本。

```
Join( $P_1, P_2$ )
)  $P = \{\}$ 
  对于每一对类  $C_i \in P_1$  和  $C_j \in P_2$ 
     $C_k = \text{Intersect}(C_i, C_j)$ 
     $P = P \cup C_k$  // 当  $C_k$  为空时，忽略。
  返回  $P$ 
```

```
Intersect( $C_i, C_j$ )
   $C_k = C_i \cap C_j$  // 设置相交点
  如果  $C_k \neq \{\}$ ，并且  $C_k$  没有值数
    则  $C_k = C_k \cup \{v_k\}$  //  $v_k$  是新的值数
     $C_k = (C_k - \{vpf\}) \cup \{\varphi_b(v_i, v_j)\}$ 
    //  $vpf$  是  $C_k$  中的值  $\varphi$  函数， $v_i \in C_i$ ， $v_j \in C_j$ ， $b$  是连接块。
  返回  $C_k$ 
```

定理1. 如果 $e_1 \equiv e_2$ 在一个点 p ，并且点 p 支配着加入点 j ，那么 $e_1 \equiv e_2$ 在 j 处 *iff*，Join 算法检测它们的等同性。

定理2. 如果变量 $x=y$ 在每条路径上加入点 j ，那么 $x=y$ 在 j 处 *iff*。连接算法检测它们的等价性。

5.2 传递功能

给定语句 s 的 *in* 的分区 PIN_s ，语句 s 的转移函数计算其 *out* 点的分区 $POUT_s$ ，其定义如下。转移函数使用函数 `valueExpr`，它接受一个表达式 e 并返回 e 的值表达式，如果 e 的形式是 $x \oplus y$ ，否则返回 e 本身。函数 `valuePhiFunc` 接受数值表达式和一个分区以及如果值表达式所代表的表达式为 φ -函数，则返回值表达式

⁶块的传递函数是块中每个语句的传递函数的组成[1]。

是一个表达式的合并。否则它返回NULL。这个函数假设每个区块外的分区对它来说是可访问的。这个函数的概念在下面给出，详细的算法在附录中。

```
transferFunction( $x = e, PIN_s$ )
     $POUT_s = PIN_s$ 
    如果  $x$  在一个  $C$  类  $i$ , 在  $POUT$ 
        中s
        那么  $C_i = C_i - \{x\}$ 
     $ve = \text{valueExpr}(e)$ 
     $vpf = \text{valuePhiFunc}(ve, PIN_s)$  // 可以是NULL
    如果  $ve$  或  $vpf$  在  $C$  类  $i$ , 在  $POUT_s$  // 在NULL时忽略  $vpf$ 
        那么  $C_i = C_i \cup \{x, ve\}$  // 集合联合
    else  $POUT_s = POUT_s \cup \{v_n, x, ve : vpf\}$  //  $v_n$  是新的值数
    返回  $POUT_s$ 
```

```
valuePhiFunc( $ve, P$ )
    如果  $ve$  的形式是  $\phi_k(v_{i1}, v_{j1}) \oplus \phi_k(v_{i2}, v_{j2})$ 
        那么  $v_i = \text{getVN}(POUT_{k_1}, v_{i1} \oplus v_{i2})$ 
        如果  $v_i == \text{NULL}$ 
            那么  $v_i = \text{valuePhiFunc}(v_{i1} \oplus v_{i2}, POUT_{k_1})$ 
         $v_j = \text{getVN}(POUT_{k_r}, v_{j1} \oplus v_{j2})$ 
        如果  $v_j == \text{NULL}$ 
            那么  $v_j = \text{valuePhiFunc}(v_{j1} \oplus v_{j2}, POUT_{k_r})$ 
    返回  $\phi_k(v_i, v_j)$  //  $v_i, v_j$  是非NULL的。
```

定理3. 假设 $x = e$ 是程序中某点 p 的语句，在通往 p 的每条路径中的 p_i ，存在表达式 e_i ，使得 p_i 's 中至少有一个不支配 p 。那么表达式 e 有一个价值 ϕ -函数，如 valuePhiFunc 算法所计算的，iff 表达式 e_i 和 e 在各自的路径中是等价的。

定理4. 假设 $x = e$ 是程序中某一点 p 的语句，在通往 p 的每条路径中都存在表达式 e_i 。表达式 e_i 和 e 在各自的路径中是等价的，iff transferFunction 算法可以检测到这些等价。

5.3 迭代算法

下面给出的算法 $\text{detectEquivalences}$ 分析了程序（以 CFG G 表示），并计算了程序中每个点的等价表达的分区。迭代分析方法改编自 [1]。该算法用分区 T （顶部）初始化每条语句（第一条语句除外）的输出点。 T 是一个特殊的分区，其属性为 $\text{Join}(P, T) =$

$P = \text{Join}(T, P)$ 。该算法在每个点上迭代计算分区，直到检测到的等价物没有变化（从上一次迭代开始）。

detectEquivalences(G)

$PIN_1 = \{\}$ 。 // "1" 是程序中的第一条语句。

$POUT_1 = \text{transferFunction}(PIN_1)$

对于程序中除第一条语句以外的每一条语句 s

$POUT_s = T$

当任何 $POUT$ 发生变化时 // 也就是等价物的变化

对于程序中除第一条语句以外的每一条语句 s

如果 s 出现在有两个前例的区块 b 中, 那么 $PIN_s =$

$\text{Join}(POUT_{s^l}, POUT_{s^u})^a$ else $PIN_s =$

$POUT_{s^l}$

$POUT_s = \text{transferFunction}(PIN_s)^b$

$a s^l$ 和 s^u 是各自前身的最后陈述。

$b s^l$ 是 s 之前的语句。

定理1 (健全性和完全性)。 假设 P 是由迭代数据流分析算法计算出的 p 点的分区。两个表达式在 p 处是等价的, iff 算法检测到了它们的等价性。

附录中给出了算法的正确性证明的概要。

5.4 复杂度分析

让程序中存在 n 个表达式。根据Join和transferFunction的定义, 一个分区可以有 $O(n)$ 个类, 每个类的大小为 $O(v)$, 其中 v 是程序中变量和常量的数量。连接操作是分区的类间相交。在支持查找的古老数据结构下, 每个类的相交需要 $O(v)$ 时间。在总共有 n^2 个这样的交集的情况下, 一个连接需要 $O(n^2 \cdot v)$ 时间。如果有 j 个连接点, 一个迭代中所有连接操作的总时间是 $O(n^2 \cdot v \cdot j)$ 。转移函数包括在输入分区中构建和查找价值表达式或价值 φ -函数。一个价值表达式的计算和查找需要 $O(n)$ 时间。计算表达式 $x+y$ 的价值 φ -函数基本上涉及到在一个连接块的左和右的前身的分区中递归地查找价值表达式。如果一个查找表被维护以将价值表达式映射到价值 φ -函数(或者当价值表达式没有价值 φ -函数时为空), 那么价值 φ -函数的计算可以在 $O(n \cdot j)$ 时间内完成。因此, 语句 $x=e$ 的转移函数需要 $O(n \cdot j)$ 时间。在一个有 n 个表达式的程序中, 所有转移函数在一个迭代中所花费的总时间是 $O(n^2 \cdot j)$ 。因此, 所有的连接和转移函数在一次迭代中花费的时间是 $O(n^2 \cdot v \cdot j)$ 。如[4]所示, 在最坏的情况下, 迭代分析需要 n 次迭代, 因此分析所花费的总时间为 $O(n^3 \cdot v \cdot j)$ 。

6 实施和结果

在本节中, 我们将新算法与Kildall[5]和Gulwani[4]的算法进行比较。我们选择了Kildall的算法, 因为它是完整的, 而且是广泛的

我们选择了Gulwani的算法，因为它只需要多项式时间。三个迭代数据流分析算法在程序的每个点上计算等价信息。我们在LLVMv 3.4编译器中实现了这些算法，并将clang作为前端。这些实现考虑了所有的算术操作、转换操作、向量操作和聚合操作，而为了简化实现，忽略了内存和分支操作。该实现使用`llvm::DenseMap`、`llvm::SmallPtrSet`和`llvm::SmallVector`类来定义分区、等价类和值表达。分区的实例与每个指令的输入和输出点相关。实现的输入是LLVM-IR的SSA形式。由于Kildall和Gulwani的算法适用于非SSA形式的程序，我们修改了算法以处理 ϕ -函数。我们使用SPEC2006程序对该算法进行了比较，结果是在运行Ubuntu 12.04的2GHz英特尔至强处理器和8GB内存上获得的。

表1.Gulwani、Kildall和我们的算法所检测到的冗余的数量

CINT2006	Gulwani	基达尔	建议	改善(%)
Mcf	32	36	36	12.5
卫星	130	153	153	17.7
洛克菲勒公司 (Libquantum)	210	259	259	23.3
bzip2	580	691	691	19.1
sjeng	1141	1265	1265	10.9
哼哼	3810	4204	4204	10.3
gobmk	8907	10005	10005	12.3
h264ref	8982	10216	10216	13.7
gcc	19837	23300	23300	17.5
CFP2006	Gulwani	基达尔	建议	改善(%)
米克	775	867	867	11.9
sphinx3	827	919	919	11.1
吕梁市	1085	1169	1169	07.7
朔方	2685	3022	3022	12.6
povray	3319	3623	3623	09.2

表1显示了使用Gulwani、Kildall和新算法在SPEC2006 CINT和CFP C/C++程序中检测到的冗余数量。该表还给出了新算法在检测冗余方面比Gulwani的改进百分比。结果显示，所提出的算法检测到的冗余数量与Kildall的完整算法相同，从而证明了完整性。同时，与Gulwani的算法相比，这两种算法都能检测到更多的冗余，平均改进率为14.2%。

这些图表表明，在实际程序中可能存在一些语句，例如 $z = x \oplus y$ 的形式，使得变量 z 在通往语句的不同路径中等同于不同的变量。Gulwani没有检测到 $x \oplus y$ 的冗余，而Kildall和新的算法都可以捕捉到它。

表2. 分析输入的SPEC2006程序及其大小所需的时间（以秒为单位）（当转换为LLVM-IR SSA形式）。

CINT2006	方案的规模		分析的时间		
	#加入了	#说明	基达尔	Gulwani	建议
Mcf	171	1815	2.5961	0.8520	0.4917
洛克菲勒公司 (Libquantum)	277	5045	7.5244	1.8921	1.1035
卫星	450	6586	13.7687	4.1121	2.0936
bzip2	814	13346	66.4680	9.3012	6.3841
sjeng	1874	18658	119.0993	15.7408	9.5835
哼哼	3279	48387	203.3485	34.6138	30.2571
gobmk	9754	105994	361.5141	49.1068	45.5976
h264ref	6804	116253	358.3743	70.6684	67.6074
编码	45861	605303	750.6864	110.2226	98.3966
CFP2006	#加入了	#说明	基达尔	Gulwani	建议
吕梁市	55	3773	7.6245	3.9202	1.4973
米克	1103	18867	30.4560	8.7964	5.5625
sphinx3	1836	22929	62.6717	22.4852	18.4850
朔方	3206	48513	136.0443	25.3210	21.4148
povray	8349	128305	320.8518	79.0802	74.7350

为了显示我们算法的有效性，我们测量了在编译SPEC2006程序时进行分析所需的CPU时间。这些时间是用clang的`-ftime-report`选项测量的。表2给出了实现分析SPEC程序所需的时间（以秒为单位）。该表还给出了连接块和指令的数量，以表明在分区上应用的连接操作和转移函数的数量。表2显示，新算法分析SPEC程序的时间比Gulwani的多项式时间算法要少。我们认为这是因为Gulwani递归地交叉了等价类（至少有一个共同的变量）以检测连接点的等价表达式（见第3.5节，连接算法，[4]中的第3-5行）。然而，所提出的算法只对等价类进行了简单的交叉。只有在需要时，才会通过计算 φ 函数来检测表达式的路径上的等价关系。这两种算法都比Kildall的指数时间算法花费的时间少得多。Kildall的连接操作与Gulwani的类似，只是在Kildall中，连接操作递归地与等价类相交，即使有

这使得它在三种算法中是最不容易的。

表中的结果清楚地表明，拟议的算法是完整的，因为它检测到的冗余数量与Kildall的完整算法相同。同时，与Gulwani的多数时间算法相比，该算法也是有效的，因为它花费的时间更少。由于所提出的算法可以在相对较短的时间内检测到更多的冗余，该算法可用于帮助生成更快代码的冗余消除算法。

7 相关工作

Kildall[5]在GVN方面的开创性工作是使用迭代数据流分析算法在程序的每个点上检测等价关系。该算法使用了等价表达式分区的“结构化”，这使它变得完整。然而，分区结构化扩大了它的规模，因此影响了算法的efficiency。为了提高检测等价物的efficiency，Alpern等人提出了一种算法（称为AWZ算法）[2]，该算法适用于静态单一赋值（SSA）形式的程序，并使用*同构*概念。该算法虽然是efficient，但不如Kildall的算法精确，原因之一是它不解释 φ -函数。Knoop, and Steffen [8]通过使用*归一化规则*，在检测到的等价物的数量方面改进了AWZ。这些规范化规则本质上是对 φ -函数的解释。正如Gulwani[4]所证明的，该算法是efficient的，但不是完整的。Briggs等人[3]的*基于Dominator的数值编号*算法适用于SSA形式。该算法并不完整，因为它对程序中的循环进行了悲观的假设。Simpson[10]提出的*基于SCC的数值计算*算法考虑了运算符的语义来改进AWZ。然而，该算法有与AWZ类似的问题，因为它不解释 φ -函数。VanDrunen[11]和Odaira[7]的SSA中的GVN算法检测并消除了更广泛的部分冗余，而不仅仅是全部冗余。Gulwani和Necula[4]的多名义时间算法据称可以检测特定大小的表达式之间的所有等价关系。然而，使用这种GVN算法无法检测到一些冗余[9]。Nie提出了一个Gulwani算法的SSA版本[6]。一般来说，这些算法要么是完整的，要么只需要多项式时间，但不能两者兼得。

8 总结

程序中等价表达的检测是一种静态分析，旨在消除冗余表达。这里的基本问题是在程序的每个点上检测等价关系，以便在多项式时间内检测出所有的冗余。为此，我们引入了新概念-- φ 函数。然后，我们提出了一个迭代数据流分析算法，该算法使用值 φ -函数来检测等价物。我们表明，该算法是完整的，并且

只需要多项式的时间。此外，我们实现了我们的算法，并将其与文献中两个广泛接受的GVN算法进行了比较。实验结果表明，所提出的算法是完整的和有效的。

鸣谢。我们感谢Vineeth Paleri、Muralikrishnan K、Vinith R和匿名审稿人提出的有见地的意见。

A 附录

A.1 价值评估

这个递归函数计算一个给定值表达式的 φ -函数。该函数假设每个块中的分区对它来说是可用的。这个函数使用equiVE来替换给定值表达式的操作数，只要有可能就用等值的 φ -函数。否则，它将返回原样的值表达式。这里使用的getVN函数从连接块 k 的左边或右边的前身中取出一个分区，它在该分区中搜索输入的值表达式，如果存在，则返回其值编号。如果该分区以前被搜索过，那么该函数返回一个新的数值。这种情况可能出现在程序的循环中。

```
valuePhiFunc( $ve, P$ )
 $v_i = v_j = vpf = \text{NULL}$ 
 $ve' = \text{equiVE}(ve, P)$ 
如果  $ve'$  的形式是  $\varphi_k(v_{i1}, v_{j1}) + \varphi_k(v_{i2}, v_{j2})$ 
    , 那么  $v_i = \text{getVN}(POUT_{k_l}, v_{i1} +$ 
     $v_{i2})$   $v_j = \text{getVN}(POUT_{k_r}, v_{j1} + v_{j2})$ 
    如果  $v_i == \text{NULL}$ 
        那么  $v_i = \text{valuePhiFunc}(v_{i1} + v_{i2}, POUT_{k_l})$ 
    如果  $v_j == \text{NULL}$ 
        那么  $v_j = \text{valuePhiFunc}(v_{j1} + v_{j2}, POUT_{k_r})$ 
否则, 如果  $ve'$  的形式是  $\varphi_k(v_{i1}, v_{j1}) + v_m^a$ 
    那么  $v_i = \text{getVN}(POUT_{k_l}, v_{i1} + v_m)$ 
     $v_j = \text{getVN}(POUT_{k_r}, v_{j1} + v_m)$ 
    if  $v_i == \text{NULL}$ 
        那么  $v_i = \text{valuePhiFunc}(v_{i1} + v_m, POUT_{k_l})$ 
    如果  $v_j == \text{NULL}$ 
        那么  $v_j = \text{valuePhiFunc}(v_{j1} + v_m, POUT_{k_r})$ 
否则, 如果  $ve'$  的形式是  $v_m + \varphi_k(v_{i2}, v_{j2})$ 
    那么  $v_i = \text{getVN}(POUT_{k_l}, v_m + v_{i2})$ 
     $v_j = \text{getVN}(POUT_{k_r}, v_m + v_{j2})$ 
    if  $v_i == \text{NULL}$ 
        那么  $v_i = \text{valuePhiFunc}(v_m + v_{i2}, POUT_{k_l})$ 
    如果  $v_j == \text{NULL}$ 
        那么  $v_j = \text{valuePhiFunc}(v_m + v_{j2}, POUT_{k_r})$ 
如果  $v_i \wedge v_j$  // 两者都是非空的
    那么  $vpf = \varphi_k(v_i, v_j)$ 
返回  $vpf$ 
```

^a具有价值数 um 的类不具有价值的 φ -函数，或者具有 $\varphi_r(v_s, v_t)$ ，从而使块 r 支配 k 。

A.2 证明

连接算法的正确性

推理1. 如果 $e_1 \equiv e_2$ 在一个点 p , 并且点 p 支配着加入点 j , 那么 $e_1 \equiv e_2$ 在 j iff, 该算法会检测到它们的等价性。

证明. 让表达式 e_1 和 e_2 在一个点 p 处是等价的, 这样 p 就支配了连接点 j 。由于变量在SSA中只被定义一次, 所以表达式在通往 j 的每条路径上都是等价的。算法Intersect的第1行确保了这种

在连接点检测到共同的等价物。 努

定理2. 如果变量 $x=y$ 在每个加入点 j 的路径中, 那么 $x=y$ 在 j 处iff, 算法会检测到它们的等价性。

证明. 让两个变量 x 和 y 在通往连接点 j 的每条路径中都是等价的。那么通过适当地转换连接块 j 中的 ϕ -函数, 以及通过交叉算法的第1行, 也可以检测到这样的等价关系。

努

假设在每个路径上都有表达式 e_i , 在相应的路径上有表达式 $e_i \equiv e$ 。这些等价物是由transferFunction算法检测出来的
这一点在下面得到了证明。

transferFunction算法的正确性

定理3. 假设 $x = e$ 是程序中某点 p 的语句, 在通往 p 的每条路径中的 p_i , 存在表达式 e_i , 使得 p_i 's中至少有一个不支配 p 。那么表达式 e 有一个价值 ϕ -函数, 如valuePhiFunc算法所计算的, iff表达式 e_i 和 e 在各自的路径中是等价的。

证明. 这可以通过对路径中连接点数量的归纳来证明, 其基本情况与图4相似。

努

定理4. 假设 $x = e$ 是程序中某一点 p 的语句, 在通往 p 的每条路径中都存在表达式 e_i 。表达式 e_i 和 e 在各自的路径iff中是等价的, transferFunction算法可以检测到这些等价。

证明. 让表达式 e_i 出现在 p' 点, 使 p' 支配 p 。那么 e_i 的等价类与它的值表达式将出现在 p' 的分区中 (由算法中的第7行和第8行保证)。由于一个变量在SSA中只被定义一次, 在语句 $x=e$ 的分区中会有一个带有 e 的值表达的类 i 。然后算法中的第6行确保 e 的等价性 i , e 被检测到。

现在考虑这样的情况: 一个表达式 e_i 出现在一个点 p' , 使得 p' 不支配 p 。在这种情况下, 在第5行计算 ϕ 函数的值 (Lemma 3) 和随后在第6行检查它的存

在，确保检测到

e 的等价物 i 和 e 在各自路径中的等价物。

努

迭代式数据流分析算法的正确性

定理1（健全性和完全性）。假设 P 是由迭代数据流分析算法计算出的点 p 的分区。两个表达式在 p 处是等价的，iff算法检测到了它们的等价性。

证明这是由定理1、2和4得出的。

努

检测冗余的算法的正确性

定理2（健全性和完备性）。假设 $s : z = x + y$ 是点 p 的状态ment。表达式 $x + y$ 是冗余的，即算法检测到它的冗余。

证明这由定理1得出。

努

参考文献

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: 编译器。原则、技术和工具，第二版。Addison Wesley, Boston (2006)
2. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: 第15届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集, POPL 1988, 第1-11页。ACM, New York (1988)
3. Briggs, P., Cooper, K., Simpson, L.: Value numbering. 软件。Practice and Experience 27(6), 701-724 (1997)
4. Gulwani, S., Necula, G.C.: 全局值计算的多项式时间算法。在。Giacobazzi, R. (ed.) SAS 2004. LNCS, 第3148卷, 第212-227页。Springer, Heidelberg (2004)
5. Kildall, G.A.: 全局程序优化的统一方法。In: 第一届ACM SIGACT-SIGPLAN程序语言原理研讨会论文集, POPL 1973, 第194-206页。ACM, 纽约 (1973)
6. Nie, J.-T., Cheng, X.: 一种基于SSA的完整全局数值编号的古老算法。In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 319-334. Springer, Heidelberg (2007)
7. Odaira, R., Hiraki, K.: 局部值数冗余消除。In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 409-423. 斯普林格, 海德堡 (2005)
8. O., Knoop, J., Steffen, B.: 检测变量的相等性：结合efficiency与精确性。In: Cortesi, A., Fil'e, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 232-247. Springer, Heidelberg (1999)
9. Saleena, N., Paleri, V.: 冗余检测的全局值编号：一种简单而有效的算法。In: 第29届ACM年度应用计算研讨会论文集, SAC 2014, 第1609-1611页。ACM, 纽约 (2014)
10. Simpson, L.T.: 价值驱动的冗余消除。博士论文, 莱斯大学, 休斯顿, 德克萨斯州, 美国 (1996)。
11. VanDrunen, T., Hosking, A.L.: 基于价值的部分冗余消除。In: Duesterwald, E. (ed.) CC 2004. LNCS, 第2985卷, 第167-184页。Springer, Heidelberg (2004)