Усовершенствованный алгоритм распространения констант с использованием GSA-представления

П.М. Довгалюк
(Pavel.Dovgaluk@nov.astrosoft.ru)

Аннотация. В данной статье представлены усовершенствования метода распространения констант, использующего GSA-представление (Gated Single Assignment), позволяющие алгоритму находить большее количество констант, чем исходный алгоритм.

1. Введение

Распространение констант – хорошо известная проблема глобального анализа потока данных. Цель распространения констант состоит в обнаружении величин, которые являются постоянными при любом возможном пути выполнения программы, и в распространении этих величин так далеко по тексту программы, как только это возможно. Выражения, чьи операнды являются константами, могут быть вычислены на этапе компиляции. Поэтому использование алгоритмов распространения констант позволяет компилятору выдавать более компактный и быстрый код.

Хотя в общем случае проблема распространения констант является неразрешимой, существует множество проявлений этой проблемы, для которых существуют эффективные алгоритмы.

Технологии распространения констант позволяют решить следующие задачи:

- Выражения, вычисляемые на этапе компиляции не нужно вычислять в процессе выполнения программы;
- Код, который никогда не выполняется, может быть удален, а недостижимый код может появляться в тех случаях, когда значение предиката в условном выражении неизменно и известно на этапе компиляции;
- увеличение эффективности встраивания процедур.

Распространение констант совместно со встраиванием процедур (когда многие параметры процедур являются константами) позволяет избежать разрастания кода, которое часто является результатом встраивания процедур в места из вызовов.

2. Алгоритм распространения констант, использующий GSA-представление

Алгоритм распространения констант, описанный в [8], использует SSA-представление (Static Single Assignment) программы. В SSA-форму программа трансформируется таким образом, что только одно присваивание может достигнуть точки использования. Так как программы имеют ветвления и точки объединения нескольких путей, то в точках объединения необходимо добавить специальную форму присваивания, названную ϕ-функцией. ϕ-функция имеет форму $V \leftarrow \phi(R, S, \ldots)$, где V, R, S, … – переменные. Количество операндов такой функции равняется количеству предшественников данного узла. Эти предшественники перечисляются в некотором определенном порядке, и j-й операнд ϕ-функции ассоциируется с j-м предшественником узла. Если путь выполнения программы лежит через j-го предка узла, то переменная V получает значение, связанное с j-м аргументом. Каждое исполнение ϕ-функции использует только один из аргументов, но который именно – зависит от того, из которого узла получил управление данный узел.

В работе [8] алгоритмы распространения констант разделены на те, которые находят простые и условные константы. Алгоритм Sparse Conditional Constant Propagation находит все простые, а также некоторые условные константы. Время работы такого алгоритма пропорционально размеру SSA-графа, и каждое SSA-ребро обрабатывается максимум два раза.

Когда необходимо классифицировать переменную в точке слияния, мы используем функцию meet для аргументов ее ϕ-функции. Но если в процессе своего выполнения программа всегда идет только по одной ветке, было бы лучше использовать то значение переменной, которое порождается именно в ней. В методе, описанном в предыдущем разделе, если значение предиката будет постоянным, то выполняемое ребро будет добавлено к рабочему списку. Поэтому в точке слияния выражения будут вычисляться с использованием информации о входящих выполняемых ребрах, что может приводить к неоднократному вычислению одних и тех же выражений.

В методе, использующем GSA, если символ использует значение из точки слияния, то вычисляется значение предиката и определяется путь, из которого берется значение. При использовании ϕ-функции мы не можем определить, по какому пути шло выполнение программы. Но если снабдить ϕ-функцию предикатом, то если его значение является константой, можно выбрать нужный аргумент ϕ-функции, а если нет, то самое лучшее, что можно получить с помощью данного метода – это взять функцию meet от ϕ-аргументов.

Чтобы выполнить данную задачу, необходимо расширить SSA-представление до GSA (Gated Single Assignment), введенное в работе [3], которое позволяет вычислять условные выражения, базирующиеся на предикатах. В данном представлении ϕ-функции заменяются на μ- и γ-функции. Большая часть ϕ-функций, расположенных в заголовках циклов переименовывается в μ-функции, тогда как

остальные преобразуются в γ-функции. γ-функция имеет вид: v = γ(P, v1, v2), что означает, что v принимает значение v1, если P=true, и v2, если P=false. В такой форме γ-функция представляет собой конструкцию if-then-else, но она может быть расширена для работы с более сложными условными конструкциями. Алгоритм для преобразования ϕ-функций в μ- и γ-функции представлен в работе [6].

В данном алгоритме используется представление программы в виде базовых блоков, состоящих из кортежей. Базовые блоки и кортежи внутри них связаны между собой и вместе образуют граф потока данных. Кортежи имеют следующие атрибуты: op, left, right, ssa_link, lattice, где op – код операции, left и right – два операнда, ssa_link – связь, порождаемая алгоритмом преобразования программы в SSA-форму. Поля left, right, ssa_link представляют собой указатели на соответствующие кортежи. Каждая операция (кортеж) также имеет ассоциированное значение – lattice, принадлежащее множеству значений из решетки и представляющее собой результат выполнения операции, который может быть получен на этапе компиляции.

3. Исходный алгоритм

```
∀ t ∈ tuples
   lattice(t) = T
   unvisited(t) = true

Visit all basic blocks B in the program
   Visit all tuples t within B
      if unvisited(t) then propagate(t)

propagate (tuple t)
   unvisited(t) = false
   if ssa_link(t) ≠ 0 then
      if unvisited(ssa_link(t))
         then propagate(ssa_link(t))
      lattice(t) = lattice(t) Π lattice(ssa_link(t))
   endif
   if unvisited(left(t)) then propagate(left(t))
   if unvisited(right(t)) then propagate(right(t))
   case on type (t)
      constant C: lattice(t) = C
      arithmetic operation:
         if all operands have constant lattice value
            then lattice(t) = arithmetic result of lattice
                 values of operands
            else lattice(t) = ⊥
         endif
      store: lattice(t) = lattice(RHS)
      ϕ-function: lattice(t) = Π of ϕ-arguments of t
      γ-function:
         if lattice(predicate) = C then
            lattice(t) = lattice value of γ-argument
                 corresponding to C
            else lattice(t) = Π of all γ-arguments of t
         endif
      μ-function: lattice(t) = ⊥
      η-function: lattice(t) = lattice(η-argument)
      default: lattice(t) = ⊥
   end case
end propagate
```


4. Недостатки и предлагаемые изменения

В работе [7] показано, что арифметические и логические выражения, включающие γ-функции, можно преобразовывать в несколько вложенных γ-функций и арифметических выражений, что позволяет получать константные выражения даже в тех случаях, когда аргументы исходного арифметического или логического выражения не являются константами. Это видно по следующему примеру:

```
if P then
   a1 = 2
   b1 = 1
else
   a2 = 4
   b2 = 2
endif
a3 = γ(P, a1, a2)
b3 = γ(P, b2, b3)
if a3 > b3 then
   ...
endif
```

В данном случае переменные a3 и b3 будут всегда равны независимо от значения предиката P. Алгоритм, предложенный в [5], не сможет определить отношение значений переменных в данной ситуации.

Предлагаемые изменения алгоритма привели бы предикат a3 > b3 к виду: γ(P, a1 > b1, a2 > b2) = γ(P, true, true) = true.

Предлагаемые изменения алгоритма касаются обработки арифметических и логических выражений и состоят в следующем (E1, E2 – аргументы арифметического (логического) оператора):

1. E1 и E2 не содержат γ-функций, либо обе содержат γ-функции с разными предикатами. Используется обычное правило для вычисления арифмети-ческих и логических выражений.
2. Один из аргументов (E1 для определенности) содержит γ-функцию. Выполняется следующее преобразование:
   γ(P, V1, V2) op E2 = γ(P, V1 op E2, V2 op E2).

3. E1 и E2 содержат в себе γ-функции с одинаковыми предикатами. Выполняется следующее преобразование:
   γ(P, V1, V2) op γ(P, V3, V4) = γ(P, V1 op V3, V2 op V4).

5. Новый алгоритм

```
∀ t ∈ tuples
   lattice(t) = T
   unvisited(t) = true

Visit all basic blocks B in the program
   Visit all tuples t within B
      propagate(t)

propagate (tuple t)
   if not unvisited(t) return
   unvisited(t) = false
label restart:
   if ssa_link(t) ≠ 0 then
      ssa_link(t)
      lattice(t) = lattice(t) Π lattice(ssa_link(t))
   endif
   if type(t) ≠ arithmetic operation
      propagate(left(t))
      propagate(right(t))
   endif
   case on type (t)
      constant C: lattice(t) = C
      arithmetic or logic operation:
         if type(left(t)) = γ-function
            or (type(right(t)) = γ-function
         then
            if type(left(t)) = γ-function
                 and (type(right(t)) = γ-function
                    or type(ssa_link(right(t))) = γ-function)
                 and (predicate(left(t)) = predicate(right(t))
                    or type(ssa_link(left(t))) = γ-function)
            then
                 join_common_predicate(t)
                 goto restart
            else
               join_gamma_with_operand(t)
               goto restart
            endif
         else
            propagate(left(t))
            propagate(right(t))
         endif
```



```
         if all operands have constant lattice value
            then lattice(t) = arithmetic result of lattice
                 values of operands
            else lattice(t) = ⊥
         endif
      store: lattice(t) = lattice(RHS)
      ϕ-function: lattice(t) = lattice(left(t)) Π
lattice(right(t))      γ-function:
         propagate(predicate(t))
         if lattice(predicate(t)) = C then
            lattice(t) = lattice value of γ-argument
                 corresponding to C
         else
            lattice(t) = lattice(left(t)) Π lattice(right(t))
         endif
      μ-function: lattice(t) = ⊥
      η-function: lattice(t) = lattice(η-argument)
      default: lattice(t) = ⊥
   end case
end propagate
```

Функция join_common_predicate объединяет две γ-функции с одинаковыми предикатами в одну:

```
join_common_predicate (tuple t)
   tuple new_t = new γ-function
   predicate(new_t) = predicate(left(t))

   left(new_t) = new operation node, same as t
   ssa_link(left(new_t)) = 0
   unvisited(left(new_t)) = true
   lattice(left(new_t)) = T
   left(left(new_t)) = left(left(t))
   right(left(new_t)) = right(left(t))

   right(new_t) = new operation node, same as t
   ssa_link(right(new_t)) = 0
   unvisited(right(new_t)) = true
   lattice(right(new_t)) = T
   left(right(new_t)) = left(right(t))
   right(right(new_t)) = right(right(t))

   t = new_t
   ssa_link(t) = 0
   lattice(t) = T
end join_common_predicate
```

Функция join_gamma_with_operand – объединяет γ-функцию и операнд, который не является γ-функцией:

```
join_gamma_with_operand (tuple t)
   tuple g, op
   tuple g_left = new operation node, same as t
   tuple g_right = new operation node, same as t
   if type(left(t)) = γ-function then
      g = left(t)
      op = right(t)
      left(g_left) = left(g)
      left(g_right) = right(g)
      right(g_left) = op
      right(g_right) = op
   else
      op = left(t)
      g = right(t)
      left(g_left) = op
      left(g_right) = op
      right(g_left) = left(g)
      right(g_right) = right(g)
   endif
   left(g) = g_left
   right(g) = g_right
   t = g

   unvisited(g_left) = true
   lattice(g_left) = T
   ssa_link(g_left) = 0

   unvisited(g_right) = true
   lattice(g_right) = T
   ssa_link(g_right) = 0
end join_gamma_with_operand
```

6. Время работы алгоритма

Т.к. в оригинальном алгоритме флаг unvisited для каждого узла не может измениться более одного раза, то это означает, что алгоритм посещает каждый узел единожды, поэтому время его работы может быть оценено как O(N), где N – количество узлов.

Модифицированный алгоритм, как и исходный, посещает каждый узел представления программы единожды. В предельном (невозможном на практике) случае, однако, тип каждого узла может быть преобразован, поэтому время, необходимое для обработки одного узла, может увеличиться. Тем не менее, время работы алгоритма также представляет собой величину порядка O(N).

Таким образом, улучшенный алгоритм обладает большими возможностями по обнаружению констант, чем исходный, лишь незначительно увеличивая время его работы (при том, что асимптотическая оценка времени работы остается той же).

Для сравнения предикатов можно использовать сравнения указателей на них (т.е. предикаты эквивалентны, если они указывают на один и тот же оператор

ветвления) или алгоритм классификации выражений, описанный в [2]. Первый вариант позволяет выполнить сравнения без значительных затрат времени. Второй позволяет найти одинаковые предикаты, которые не принадлежат одному оператору ветвления, но требует дополнительных затрат времени на классификацию предикатов.

Литература

1. А. Ахо, Р. Сети, Д. Ульман. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2001.
2. B. Alpern, M. N. Wegman, F. K. Zadeck. Detecting equality of variables in programs, 1988.
3. R. A. Balance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In Proc. ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation, June 1990.
4. S. Muchnick. Advanced compiler design and implementation, 1997.
5. E. Stoltz, M. Wolfe, M. P. Gerlek. Constant propagation: a fresh, demand-driven look, 1994.
6. E. Stoltz, M. Wolfe, and M. P. Gerlek. Demand-driven constant propagation. Technical Report 93-023, Oregon Graduate Institute, 1993.
7. P. Tu, D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. ACM International Conference on Supercomputing, 1995.
8. M. N. Wegman, F. K. Zadeck. Constant propagation with conditional branches. ACM Trans. on Programming Languages and Systems, 13(2):181-210, July 1991.