**Анализ потока данных.**

Под анализом потоков данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте. Такая постановка задачи возможна по той причине, что язык программирования и вычислительная среда определяют некоторую общую, "безопасную" семантику конструкций, которая годится "на все случаи жизни". Учет же контекстных условий позволяет делать более конкретные, частные заключения о поведении той или иной конструкции; при этом такие заключения, вообще говоря, перестают быть верными в другом контексте. Например, общая семантика присваивания заключается в вычислении выражения, стоящего в правой части, и присваивании полученного значения в переменную, стоящую в левой части. Однако в случае, когда выражение в правой части не имеет побочных эффектов, а переменная в левой части более нигде не используется, данный оператор становится эквивалентен пустому.

Для того чтобы описать понятие контекста, обратимся к графу потока управления.



 Понятно, что на смысл каждой конструкции может оказывать влияние любая конструкция, из которой в этом графе достижима данная. Отсюда следует, что для правильного учета контекста необходимо учесть влияние всех путей до данной вершины, сначала определив влияние каждого пути, а затем выделив общую часть. Задача осложняется тем, что при наличии контуров множество всех путей в графе управления становится бесконечным.

Далее в этой лекции будет рассмотрен общепринятый итеративный подход, который позволяет получить приближенное решение задач анализа потоков данных, а при определенных условиях это решение становится точным.

**Пример**



Для демонстрации сути задач анализа потоков данных рассмотрим несколько примеров.

На иллюстрации приведен фрагмент программы. Вхождения одного и того же выражения **(v + i) -> b**, обведенные сплошной линией, являются эквивалентными. В то же время вхождение того же самого выражения, обозначенное пунктирной линией, не эквивалентно первым двум, поскольку else -часть условного оператора содержит разрушающее присваивание.

Понятно, что для выяснения эквивалентности данных выражений необходимо перебрать все пути и убедиться, что ни в одном из них значения переменных, входящих в выражения, не меняются.

**Устранение мертвого кода**

В теории компиляторов удалением мёртвого кода называется оптимизация, удаляющая мёртвый код. Мёртвым кодом (так же бесполезным кодом) называют код, исполнение которого не влияет на вывод программы, все результаты вычисления такого кода являются мёртвыми переменными, то есть переменными, значения которых в дальнейшем в программе не используются.

В связи с существующим разночтением термина мёртвый код, важно отметить, что оптимизация удаления мёртвого кода не занимается удалением недостижимого кода. Локализацией и удалением недостижимого кода могут заниматься сборщик мусора или другие оптимизации, например, удаления недостижимого кода.

Удаление бесполезного кода способно ускорить работу программы за счёт уменьшения количества исполняемых в ней операций и уменьшить размер программы или промежуточного представления.

Примеры

Рассмотрим следующий код на языке С:

int foo(void)

 {

 int a = 24;

 int b;

 b = a + 3; */\* Бесполезный код \*/*

 **return** a << 2;

 }

В данном примере операция сложения b = a + 3 является мёртвым кодом, так как переменная b не используется в дальнейших вычислениях и является мёртвой, начиная с этой точки и заканчивая концом процедуры. После удаления этой инструкции получим:

int foo(void)

 {

 int a = 24;

 int b; */\* Мёртвая переменная \*/*

 **return** a << 2;

 }

После удаления операции сложения, переменная b становится мёртвой во всей процедуре. Так как она объявлена локально, то может быть полностью удалена из программы:

int foo(void)

 {

 int a = 24;

 **return** a << 2;

 }

Несмотря на то что вычисление происходит внутри функции, его результат записывается в переменную, находящуюся в области видимости только этой функции; и, если учесть, что функция безусловно возвращает число 96, она может быть упрощена оптимизацией распространение констант так, чтобы её тело состояло только из операции return 96. А затем компилятор может заменить все вызовы этой функции на возвращаемое значение.

Классический алгоритм DCE («Dead») работает на SSA-представлении и состоит из двух обходов: первый обход («Mark») отмечает (маркирует) все заведомо живые операции (операции выхода из процедуры, ввода-вывода, изменяющие глобальные переменные); второй обход («Sweep») начинается с живых операций и идёт вверх по определениям аргументов, помечая все операции на своём пути живыми, заканчивая теми операциями, которые не имеют предшественников в SSA-форме. Максимальная вычислительная сложность такого алгоритма зависит от размера программы как O(n2).

**Понижение силы операций**



Данное преобразование особенно эффективно при применении к вычислениям внутри тел циклов, хотя это не является его необходимым контекстным условием.

Основная идея понижения силы операций - это замена (в частных случаях) использования более дорогих операций более дешевыми (умножения - сложением или сдвигом, возведения в степень - умножением и т.д.) Обычно используются следующие тождества:



Два последних тождества и дают возможность применения преобразования к циклам. На иллюстрации приведен пример такого преобразования. В данном случае *умножение* внутри тела *цикла* было заменено сложением.

**Экономия общих подвыражений**



Экономия общих подвыражений заключается в том, что среди всех вычисляемых на участке экономии выражений выделяются эквивалентные, и затем их вхождения заменяются на вхождение новой переменной, хранящей заранее вычисленное значение этого общего выражения.

Поскольку выделение одних подвыражений закрывает возможность выделить другие, данное преобразование является неоднозначным.

На иллюстрации показаны примеры применения этого преобразования. Во втором случае видна неоднозначность при выделении подвыражений: выражения b+c и c+d не могут быть выделены одновременно. Поскольку подвыражение b+c используется чаще, именно оно и было выделено.

**Чистка циклов**

Чистка циклов вверх



Данное преобразование применяется к сильно связным подграфам графа управления (т.е. подграфам, состоящим из взаимно достижимых вершин). Таким образом, для его проведения необходима фрагментация программы на уровне сильно связных подграфов. Алгоритм выделения сильно связных подграфов будет рассмотрен в "Анализ потока управления".

Если такой подграф выделен, то для него определяется множество входных вершин, то есть таких его вершин, в которые существует путь из вершины start, лежащий целиком за пределами этого подграфа. На иллюстрации входные вершины подграфа S обозначены темными кружками и символами e1, …, ek .

Чистка цикла вверх заключается в том, что вершина, лежащая в сильно связном подграфе, заменяется на несколько копий вне его, каждая из которых непосредственно предшествует входным вершинам. Такое преобразование возможно только в случае, когда оператор, которым помечена данная вершина, использует только данные операторов, недостижимых изнутри выбранного сильно связного подграфа.

Легко видеть, что данное преобразование также повторно по отношению к самому себе.

Чистка циклов вниз



Данное преобразование также легко реализуется в представлении с использованием def-use chains и в каком-то смысле является симметричным предыдущему.

Теперь вершина изнутри сильно связного подграфа заменяется на несколько копий, каждая из которых является непосредственным преемником выходных вершин подграфа. Выходной вершиной называется такая вершина, среди непосредственных преемников которой есть вершина, не принадлежащая данному подграфу.

Легко видеть, что преобразование корректно только в том случае, когда ни одна вершина подграфа не достижима из тех вершин, которые используют результаты оператора, которым помечена выносимая вершина.

Данное преобразование также повторно по отношению к самому себе.

Далее мы перейдем к описанию преобразований, для которых недостаточно рассмотренных выше способов представления программы.