Лекция 2

Внутренние представления программ

Внутреннее представления

Внутреннее представление (IR) – тип данных, позволяющий представить программу в форме пригодной для автоматического анализа и трансформации с сохранением семантики.

! достаточно сложный тип данных

Архитектура компилятора

Front-End → Optimizer → Code emitter

Front-End

- разбор входного языка
- построение IR_0 ("идеально" оптимального)

Optimizer

- преобразование $IR_k \rightarrow IR_{k+1}$
- с сохранением/потерей оптимальности

Code emitter

преобразование $IR_{final} \rightarrow$ object code

Внутренние представления
высокого уровня

HIR (High-level Intermediate Representation)

- порождается компонентом Front-end
- хорошо отражает конструкции транслируемого языка (языков)
- не зависит от целевой архитектуры

Внутренние представления
машинно-независимые

MIR (Middle-level Intermediate Representation)

- трансляция HIR → MIR или MIR_k → MIR_{k+1}
- слабо зависит от целевой архитектуры
- может зависеть от исходного языка

Внутренние представления
машинно-ориентированные

LIR (Low-level Intermediate Representation)

- трансляция MIR → LIR (lowering)
- сильно зависит от целевой архитектуры
- слабо зависит от исходного языка
  - тем не менее зависимость может оставаться

Внутренние представления

Основное внутреннее представление:

- автоматический анализ и трансформация
- сохранение семантики программы

Для (императивных) языков используются:

- структурные ВП
  - Аттрибутно-Синтаксическое Дерево
- управляющие графы
- гибридные ВП
  - структурное для выражений, граф - для упр. конструкций

Структурные ВП

- синтаксически-ориентированные
- управляющие операторы представлены явно
- строгая вложенность управляющих конструкций

Однако, использование таких ВП в оптимизаторе накладывает ограничения на:

a) входной язык
b) трансформации ВП
c) качество анализа

! goto и другие низкоуровневые упр. операторы не допускаются

Структурные ВП

Анализ называют *потоково-нечувствительным*, если он не делает никаких предположений о порядке исполнения операторов

Структурные ВП пригодны в основном для:

- синтаксического разбора входного языка
- потоково-нечувствительного анализа
- трансляции в другие ВП

! структурные ВП **непригодны** для многих важных видов анализа и оптимизаций

Управляющие графы

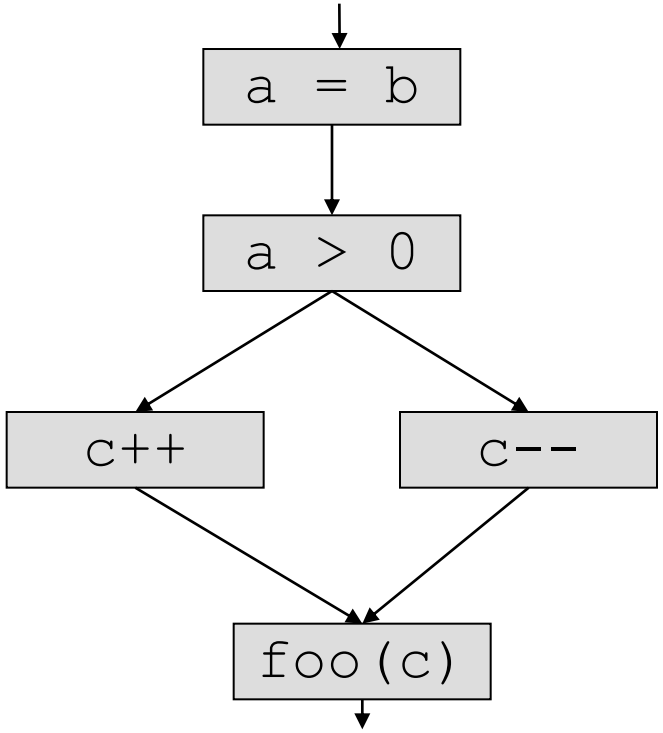
Управляющий граф – корневой орграф <V, E>:

- вершины V – операторы процедуры
- дуги E – переходы между операторами

Program

```
...
a = b;
if (a > 0)
   c++;
else
   c--;
foo(c);
```

Control Flow Graph (CFG)



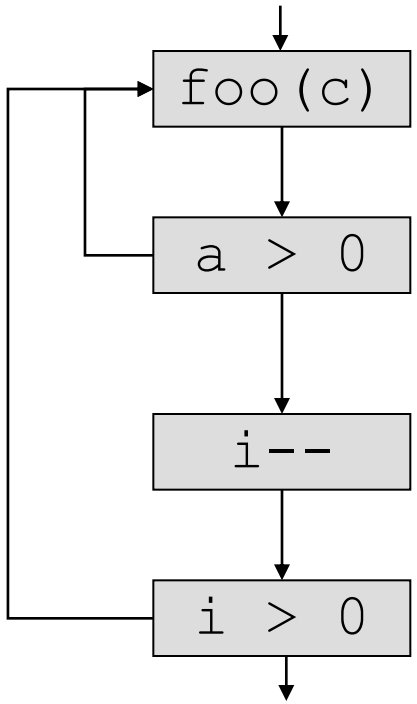
Управляющие графы

Естественное представление для неструктурных операторов:

Program

```
   ...
L1:
   foo(c);
   if (a > 0)
     goto L1;
   i--;
   if (i > 0)
     goto L1 ;
```

CFG



Управляющие графы

Pred, Succ: $V \to \mathcal{P}(V)$ – множества *предшественников* и *потомков* вершины.

Пусть $n, m \in V$

Paths (n, m) – множество путей из n в m

Считаем, что в любом CFG имеется:

- входная (корневая) вершина entry
- выходная вершина exit

$\exists!\text{entry} \,.\, \text{Pred}(\text{entry}) = \varnothing \wedge \forall v \in V \; \exists p \in \text{Paths}(\text{entry}, v)$
$\exists!\text{exit} \,.\, \text{Succ}(\text{exit}) = \varnothing$

Управляющие графы
доминирование

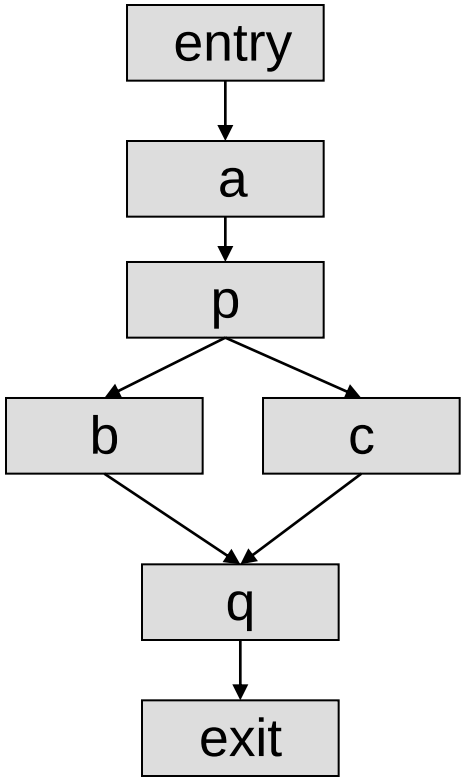
dom $\subset V \times V$ – отношение *доминирования*

x **dom** $y \Leftrightarrow_{def} \forall p \in Paths(entry, y) . p \ni x$

(говорят "x доминирует y")

Примеры:

1. a dom p
2. a dom q
3. a dom c
4. p dom b
5. $\neg$(b dom q)



Управляющие графы
постдоминирование

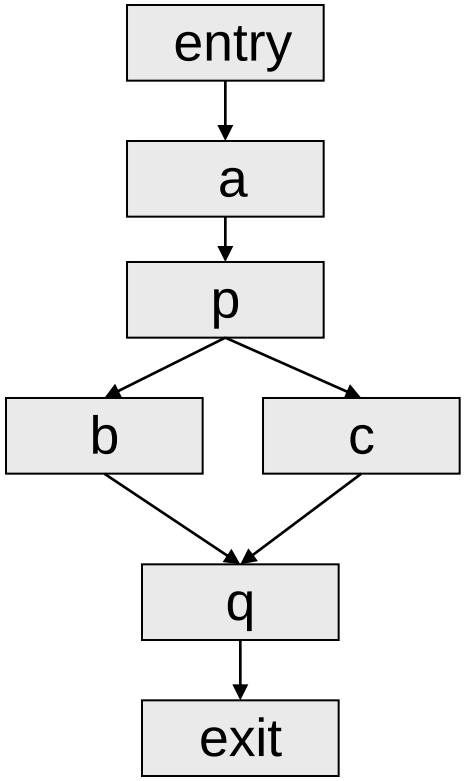
pdom $\subset V \times V$ – отношение *постдоминирования*

y **pdom** x $\Leftrightarrow_{def} \forall p \in Paths(x, exit)$. $p \ni y$

(говорят "y постдоминирует x")

Примеры:

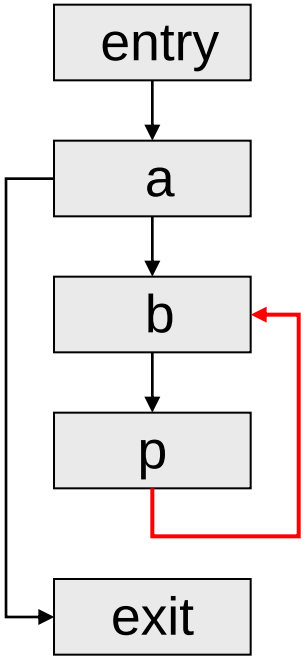
1. p pdom a
2. q pdom p
3. q pdom b
4. ¬(b pdom p)
5. ¬(c pdom a)



Управляющие графы

постдоминирование

Как доопределить отношение постдоминирования для случая "мертвых" циклов?



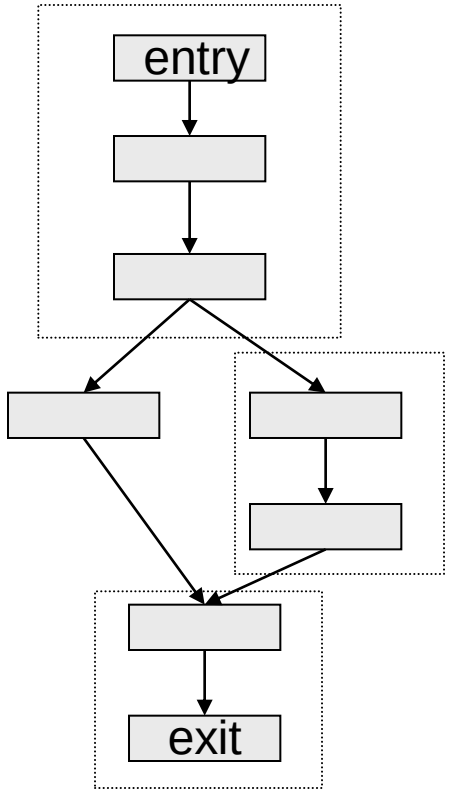
Управляющие графы

факторизация: линейные участки

Пусть $s,f \in V$.

Линейный участок (basic block) – максимальное множество $BB(s, f) \subseteq V$ со свойствами:

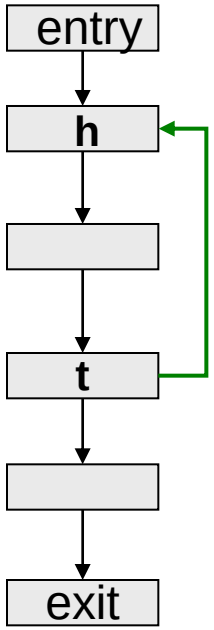
1. $|Paths(s, f)|=1$
2. $\forall a \in BB(s,f)$. s dom a
3. $\forall a \in BB(s,f)$. f pdom a

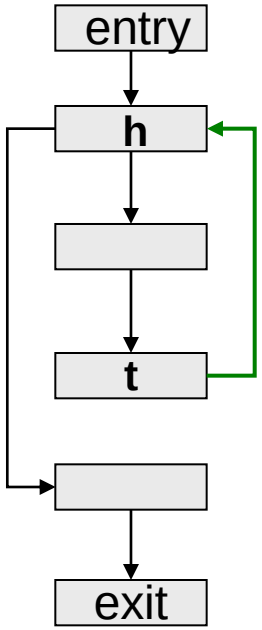


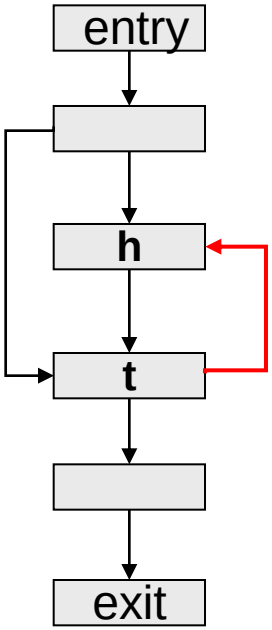
Управляющие графы
факторизация: циклы

$(t \rightarrow h) \in E$ – *обратная дуга цикла*, если h dom t

! не все обратные дуги циклов являются обратными дугами аранжированного графа:







Управляющие графы
факторизация: циклы

Пусть (t→h) – обратная дуга цикла.

Цикл Loop(h, t) – подграф $\langle Loop_V, Loop_E \rangle \subset \langle V, E \rangle$:

$$Loop_V(h, t) = \{ p \mid p \in Paths(Succ(h), t) \wedge p \notin h \} \cup \{h\}$$

$$Loop_E(h, t) = Loop_V(h, t)^2 \cap E$$

Вершины:

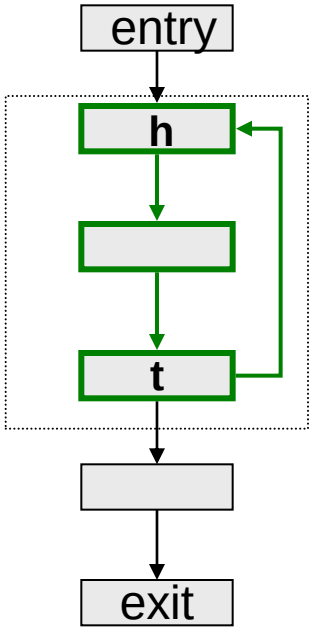
- всех путей, ведущих из h в t и не содержащих h
- собственно h – *заголовок* (header) цикла

Дуги:

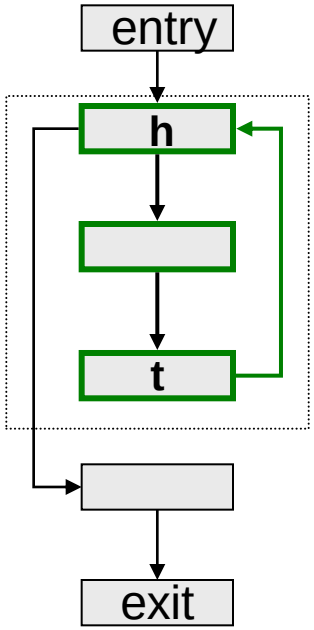
все дуги из E, соединяющие выбранные вершины

Управляющие графы
факторизация: циклы

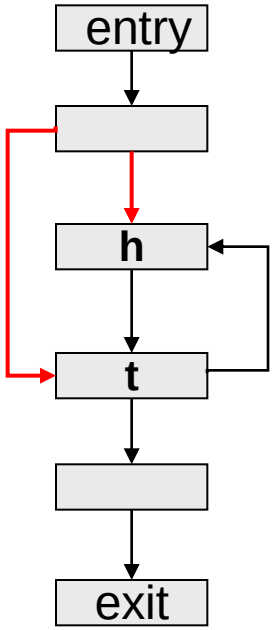
Примеры:



`do-while loop`



`while loop`



`@#$... loop`

Управляющие графы
факторизация: проблемы циклов

! Несводимые графы затрудняют оптимизацию:

есть компоненты сильной связанности не являющиеся циклами в смысле данного ранее определения ⇒

⇒ есть "циклы" с несколькими входами ⇒

⇒ многие оптимизации неприменимы

- на практике таких циклов немного
- путем клонирования подграфа можно получить сводимый CFG с сохранением семантики

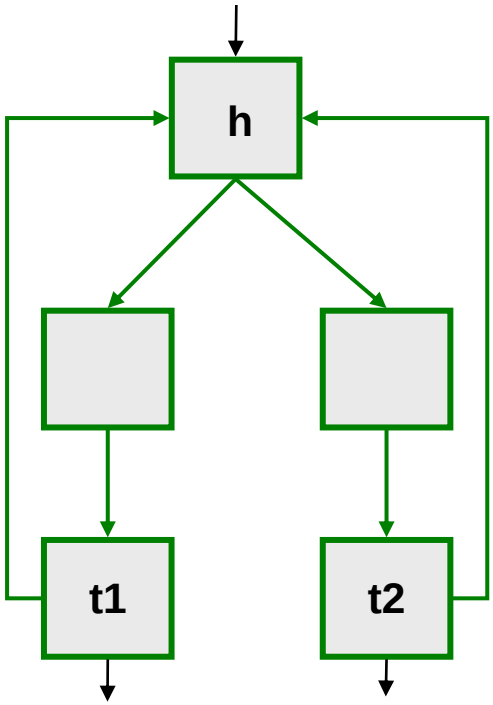
 goto позволяет "написать" несводимый граф

 break, continue – не позволяют

Управляющие графы
факторизация: проблемы циклов

Если CFG $\supset Loop_1(h) \cup Loop_2(h) \mid Loop_1(h) \neq Loop_2(h)$, нельзя определить что имелось в виду в исходной программе:

- единый цикл
- $Loop_1$ (текстуально) вложен в $Loop_2$
- $Loop_2$ (текстуально) вложен в $Loop_1$



- трактовать как единый цикл
- использовать эвристики

Управляющие графы

факторизация

Факторизация CFG называется *анализом потока управления* (control flow analysis):

- Выделяются линейные участки
- Далее используется фактор-CFG (вершинами являются линейные участки)
- Выделяются циклы

Выделить линейные участки можно проще (без привлечения предикатов dom/pdom). Как?

Управляющие графы
представление операторов

Как представляются операторы, находящиеся внутри линейных участков?

Вариации одного из трех классов ВП:

- стековая машина
  - аргументы и результаты – на стеке
- регистровая машина
  - аргументы и результаты – в переменных
- деревья и дэги (DAGs) выражений
  - лес из AST ("ASD")

Представление операторов
стековая машина

`a=2;`	`loadimm 2` `storelocal a`

`b+=a;`	`loadlocal b`
	`loadlocal a`
	`add`
	`storelocal b`

`A.f = foo (a, b)`	`loadlocal a`
	`loadlocal b`
	`call foo`
	`storeglobal A.f`

- промежуточные результаты – на стеке
- доступ к переменным – загрузка на/выгрузка со стека
- загрузки/выгрузки могут быть избыточны

Представление операторов
регистровая машина

```
...                          //let t2 hold value of b
a=2;                         t1 = 2
b+=a;                        t3 = t2 + t1
A.f = foo (a, b);            t4 = call foo (t1, t3)
                             putfield A.f (t4)
```

- **<результат, операция, аргументы>**
- аргументы и результат – переменные
- выражения разбиваются на несколько операторов (для промежуточных результатов вводятся новые переменные)
- могут быть избыточные переменные
- историческое название – 3-х адресное ВП

Представление операторов
лес деревьев и дэгов

```
...
a=2;                    2(r1)        *(r2)        foo
                                    /  \          /  \
b=a*(a+1);                         |    +        r1   r2
                                    \  /  \
foo (a, b);                          r1   1
```

- дерево/дэг имеет не более чем один результат
- дэг появляется в случае общих подвыражений
- лес упорядочен (порядок операторов в лин. участке)
- вожможна склейка дерева/дэга B с A (необходимое условие – результат B используется только в A)

Представление операторов
уровень ВП

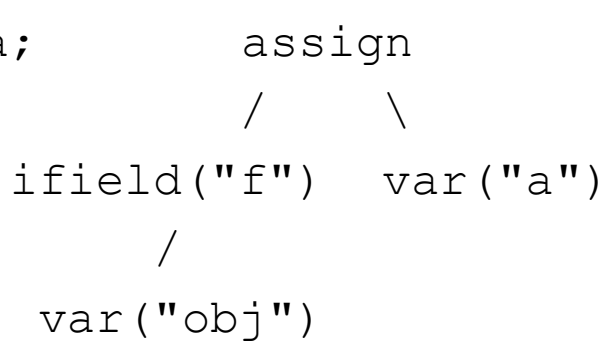
После трансляции из HIR, MIR характеризуется:

- "семантически нагруженными" операторами

Program

```
obj.f = a;
```

HIR



MIR

```
putfield f (t_obj, t_a)
```

- неогр. числом переменных/элементов стека
  - абстрактная регистровая/стековая машины
- независимостью от целевой архитектуры

Представление операторов
уровень ВП

При трансляции $MIR_k \rightarrow MIR_{k+1}$ применяется *декомпозиция* ("семантически нагруженные" операторы заменяются на более простые)

MIR_k	MIR_{k+1}

`putfield f(t_obj, t_a)`	`nullcheck (t_obj)`
	`tt = t_obj + OFFS_f`
	`store (tt, t_a)`

Декомпозиция может делаться с целью:

- более эффективной оптимизации
- учета особенностей целевой архитектуры
  - операции с 64-битными числами на 32-битном CPU

Представление операторов
уровень ВП

При трансляции MIR→LIR учитываются следующие факторы:

- число регистров/элементов стека ограничено
  - часть переменных размещается в памяти
- зависимость от архитектуры
  - команд вида reg3 = op (reg2, reg1) может не быть
  - операторы MIR могут не иметь аналогов в CPU (должны быть представлены набором команд)
  - не все регистры м.б. равноправны (неортогональность)

! оптимальная трансляция MIR→LIR и дальнейшие преобразования LIR являются одними из самых сложных фаз компиляции

Внутренние представления

Вспомогательные внутренние представления:

- часть семантики исходной программы (аспект)
- эффективная реализация анализа

Вспомогательные ВП
множества доминаторов

GFG = <V, E>

Для проведения анализа и оптимизации требуется функция

$$Dom: V \to P(V) \mid x\ dom\ y \Leftrightarrow x \in Dom(y)$$

Количество обращений к Dom достаточно велико

! при перестройке CFG желательно обновлять Dom инкрементально

Вспомогательные ВП
графы информационных зависимостей

Орграф DFG = <Vars, A>, где

Vars – множество переменных

(b→a) ∈ A, если mainIR содержит оператор a=b

Использование DFG:

- начальная разметка свойств
- удаление (collapsing) компонент сильной связанности
- обход в прямом или обратном топологическом порядке с переносом свойств на остальные переменные

! источник консервативности: порядок выполнения присваиваний не учитывается. Существуют графы инф. зависимостей, которые учитывают поток управления при вычислении свойств

Вспомогательные ВП
DU-цепочки

CFG = <V, E>

Определения и использования переменной a – множества операторов $Def(a) \subseteq V$ и $Use(a) \subseteq V$), определяющих и использующих значение a.

Пусть $d \in Def(a)$.

Использования определения d

$Use_d(a) =_{def} \{ u \in Use(a) \mid \exists p.p \in Paths(d,u) \wedge p \cap Def(a) = \{d\} \}$

$DU(d) =_{def} \{d\} \cup Use_d(a)$ – *DU-цепочка*

Вспомогательные ВП
DU-цепочки

Пример.

```
1.  x = 1;          //def x
2.  if (a > 0)
3.    y = x;        //use x
4.  else {
5.    y = x+1;      // use x
6.    x = 2         // def x
7.  }
8.  z = x+y         // use x, use x
```

DU(op1) = {op1, op3, op5, op8}

DU(op6) = {op6, op8}

Вспомогательные ВП
сети (webs)

Пусть DUChains(a) – множество всех DU-цепочек для переменной a; $\{c_1, c_2, m\} \subseteq$ DUChains(a).

Определим отношение эквивалентности:

$$c_1 \cong c_2 \Leftrightarrow_{def} c_1 \cap c_2 \neq \varnothing \vee (\exists m.\ c_1 \cap m \neq \varnothing \wedge m \cong c_2)$$

Сеть для переменной a, Web(a) – элемент фактор-множества DUChains(a) $/\cong$

! сеть определяет семантический контекст использования переменной (имя переменной его не определяет)

Вспомогательные ВП
сети (webs)

Пример 1.

```
1.  x = 1;          //def x
2.  if (a > 0)
3.    y = x;        //use x
4.  else {
5.    y = x+1;      // use x
6.    x = 2         // def x
7.  }
8.  z = x+y         // use x, use x
```

DU(op1) = {op1, op3, op5, op8}

DU(op6) = {op6, op8}

Web(x) = DU(op1) ∪ DU(op6)

Вспомогательные ВП
сети (webs)

Пример 2.

```
1.  x = 1;          //def x
2.  if (a > 0)
3.    y = x;        //use x
4.  else
5.    y = x+1;      // use x
6.  x = 2           // def x
7.  z = x+y         // use x
```

DU(op1) = {op1, op3, op5}

DU(op6) = {op6, op7}

$Web_1(x) = DU(op1)$, $Web_2(x) = DU(op6)$

Вспомогательные ВП
сети (webs)

Утверждение.

Если $Web_1(x) \neq Web_2(x)$, то в одном из них можно переименовать переменную x в x' без потери семантики программы (при условии что x' – не использующееся имя)

Даны Web(x), Web(y). При каких условиях:

1. можно выполнить замену y на x без потери семантики?
2. после преобразования из п. 1, обе сети объединятся в одну?
3. можно выполнить частичную замену x на x', после которой сеть Web(x) распадется на две?

Вспомогательные ВП
UD-цепочки

CFG = $\langle V, E \rangle$, $u \in \mathrm{Use}(a)$.

Определения, *достигающие* использования u

$$\mathrm{ReachingDefs}_u(a) =_{\mathrm{def}} \{ d \in \mathrm{Def}(a) \mid \exists p.p \in \mathrm{Paths}(d,u) \wedge p \cap \mathrm{Def}(a) = \{d\} \}$$

$$\mathrm{UD}(u) =_{\mathrm{def}} \{u\} \cup \mathrm{ReachingDefs}_u(a)$$ – *UD-цепочка*

Записать условие *корректности* CFG – каждая переменная должна определятся до использования.

Выбор ВП
критерии

- желаемая точность анализа
  - точнее степень консервативности
- емкостная сложность ВП
  - пиковое и среднее потребление памяти компилятором
- временная сложность построения ВП
  - время компиляции
- легкость модификации
  - в процессе преобразования ВП

! Выбор внутренних представлений определяет качество оптимизирующего компилятора

Выбор ВП

проблемы масштабируемости

Алгоритм (анализа) *масштабируется*, если

с ростом размера программы потребляемые время и

память мажорируются (почти) линейной функцией:

$O(n)$ либо $O(n \cdot \alpha(n))$, где

α – медленнорастущая функция

Используется $\alpha(n) = A^{-1}(n, n)$, где A – ф. Аккермана

Многие важные для оптимизации алгоритмы имеют

сложность $O(n^2)$ или хуже.

Что делать?

Выбор ВП
проблемы масштабируемости

- уменьшение области
  - самые сильные оптимизации делаются внутрипроцедурно (что хорошо заметно при неумеренной открытой подстановке методов)

- составной анализ
  - факторизация: лин. участок / процедура / набор процедур / программа
  - наиболее ресурсоемкие алгоритмы применяются в меньших областях
  - результаты анализа сохраняются
  - используются при оптимизации бОльших областей

Q&A