Основные алгоритмы

1. Алгоритм построения графа потока управления.

1.5.3. Алгоритм построения графа потока управления (ГПУ)

- ◊ **Вход**: последовательность трехадресных инструкций.
- ◊ **Выход**: список базовых блоков для данной последовательности инструкций, такой что каждая инструкция принадлежит только одному базовому блоку.
- ◊ **Метод**:
  - ♦ Строится упорядоченное множество НББ
  - ♦ Каждому НББ соответствует ББ, который определяется как последовательность инструкций, содержащая само НББ и все инструкции до следующего НББ (не включая его) или до конца последовательности инструкций.
  - ♦ Строится множество дуг графа потока управления:
    - ♦ если последняя инструкция ББ не является инструкцией перехода, строится дуга, соединяющая ББ со следующим ББ;
    - ♦ если последняя инструкция ББ является инструкцией безусловного перехода, строится дуга, соединяющая ББ с ББ, НББ которого имеет соответствующую метку;
    - ♦ если последняя инструкция ББ является инструкцией условного перехода, строятся обе дуги.

22

НББ – начала базовых блоков

Напоминаю: НББ по определению это:

- ♦ первая инструкция программы
- ♦ помеченная инструкция программы
- ♦ инструкция, следующая за инструкцией перехода

2. Алгоритм построения ориентированного ациклического графа для базового блока.

1.6.3 Метод нумерации значений – алгоритм построения ОАГ

Алгоритм (на псевдокоде) построения ОАГ для базового блока B, содержащего n инструкций вида $t_i \leftarrow Op_i,\ l_i,\ r_i$.

Функция **#val(s)** определяет номер значения, определяемого сигнатурой

s = (Op, #val(l), #val(r)) .

```
for each "t_i ← Op_i, l_i, r_i" do
    s_i = (Op_i, #val(l_i), #val(r_i))
    if(ТЗ содержит s_j == s_i)
        then
              вернуть j в качестве номера значения #val(s_i)
        else
              завести в ТЗ новую строку ТЗ_k
              записать сигнатуру s_i в строку ТЗ_k
              вернуть k в качестве номера значения #val(s_i)
```


ТЗ – таблица значений

3. Алгоритм локальной нумерации значений.

4. Алгоритмы анализа потока данных.

3.4 Обобщенный итеративный алгоритм

3.4.1 Описание алгоритма

◊ **Алгоритм.** Итеративное решение задачи анализа потока данных

- **Вход**: граф потока управления,
  структура потока данных $\langle D, F, L, \wedge \rangle$,
  передаточная функция $f_B \in F$
  константа из $l \in L$ для граничного условия
- **Выход**: значения из L для $In[B]$ и $Out[B]$ для каждого блока B в графе потока.
- **Метод**: если $D = Forward$ выполнить программу (3.4.2);
  если $D = Backward$ выполнить программу (3.4.3).

3.4.2 Решение задачи потока данных ($D = Forward$)

$Out[Entry] = l;$

for ($each\ B \neq Entry$) $Out[B] = \mathsf{T}$;

while (внесены изменения в $In[B]$)

for ($each\ B \neq Entry$) {

$$In[B] = \bigwedge_{P \in Pred(B)} f_P(In[P])$$

}

3.4.3 Решение задачи потока данных ($D = Backward$)

$Out[Exit] = l;$

for ($each\ B \neq Exit$) $Out[B] = \mathsf{T}$;

while (внесены изменения в $Out[B]$)

for ($each\ B \neq Exit$) {

$$Out[B] = \bigwedge_{S \in Succ(B)} f_S(Out[S])$$

}

5. Алгоритм построения дерева доминаторов.

Алгоритм вычисления доминаторов B: итеративно пока что-то меняется

$$D(B) = \{B\} \cup \left(\bigcap_{P \in Pred(B)} D(P) \right)$$

Граничное условие – D(Entry) = Entry

Определение. Вершина i ГПУ является *непосредственным доминатором* вершины n ($i = Idom(n)$, если

(1) $i = Dom(n)$

(2) не существует вершины m, $m \neq i$, $m \neq n$, такой что $i = Dom(m)$ и $m = Dom(n)$.

Соединяем вершину с ее непосредственным доминатором – получаем дерево доминаторов

6. Алгоритм построения дерева постдоминаторов.

- ◊ *Обратным графом* ориентированного графа $G = \langle N, E \rangle$ называется ориентированный граф $G^R = \langle N, E^R \rangle$, у которого направления всех ребер противоположны.
- ◊ **Постдоминаторы ГПУ – это доминаторы его *обратного графа*.**
- ◊ *Обратная граница доминирования* ($RDF(n)$) вершины $n \in G$ это обычная граница доминирования в обратном графе G^R.

7. Алгоритм построения границы доминирования.

4.2.2. Построение границы доминирования

◊ Свойства узлов границы доминирования позволяют составить простой алгоритм ее построения

Шаг 1. Найти все точки сбора j графа потока, т.е. все узлы j, у которых $|Pred(j)| > 1$.

Шаг 2. Исследовать каждый узел $p \in Pred(j)$ и продвинуться по дереву доминаторов, начиная с p и вплоть до непосредственного доминатора j: при этом j входит в состав границы доминирования каждого из пройденных узлов, за исключением непосредственного доминатора j.

4.2.3. Алгоритм построения границ доминирования

◊ **Вход**: граф потока

◊ **Выход**: множество границ доминирования для узлов графа потока

◊ **Метод**: выполнить следующую программу:

```
for all n ∈ N do DF(n) = ∅;
for all n ∈ N do
    {if |Pred(n)| > 1 then
        for each p ∈ Pred(n) do
            {r = p;
             while r ≠ IDom(n) do
               {DF(r) = DF(r) ∪ {n};
                r = Idom(r);
               }
            };
    }
```

8. Алгоритм построения естественного цикла по обратному ребру ГПУ

Вход: ГПУ $G = \langle N, E \rangle$ с входным узлом $Entry$.

Обратное ребро $e = \langle n, d \rangle \in E$

Выход: подграф $C \subseteq G$, являющийся естественным циклом.

Метод:

(1) начальное значение C – множество $\{n, d\}$.

(2) узел d помечается как «посещенный».

(3) начиная с узла n выполняется поиск в глубину на обратном ГПУ (направления дуг заменены на противоположные).

(4) все узлы, посещенные на шаге (3), добавляются в C.

9, Алгоритм распространения копий.

Алгоритм распространения копий

Система уравнений составляется по аналогии с системой уравнений для достигающих определений:

- в передаточной функции сначала из множества инструкций копирования удаляются инструкции «убитые» в блоке b, потом в него добавляются инструкции копирования блока b

$$Out_{CP}(b) = copy(b) \cup (In_{CP}(b) - kill(b))$$

- подставив $Out_{CP}(b)$ в уравнение сбора по всем путям, (оно в отличие от соответствующего уравнения для достигающих содержит операцию пересечения, а не объединения, так как **по всем путям должны приходить одинаковые копии**)

$$In_{CP}(b) = \bigcap_{p \in Pred(b)} Out_{CP}(p)$$

получим

$$In_{CP}(b) = \bigcap_{p \in Pred(b)} \left(copy(p) \cup (In_{CP}(p) - kill(p))\right)$$

10. Глобальное распространение констант (двухфазный алгоритм).

8.2 Алгоритм глобального распространения констант

8.2.1 Фаза инициализации

```
// Initialization Phase
WorkList = ∅;
for each SSA-name v
initialize Value(v) by rules specified in the
text;
if Value(v) ≠ Undef then
WorkList = WorkList ∪ {v};
```

На фазе инициализации устанавливаются начальные значения SSA-имен и формируется начальное состояние **WorkList**.

Для каждого SSA-имени **v** анализируется операция, определяющая **v** и устанавливающая значение **Value(v)** в соответствии со следующими простыми правилами (следующий слайд).

The rules specified in the text

1. **v** определяется φ-функцией, **Value(v) = Undef**.
2. **v** равно одной из констант c_i, **Value(v) =** c_i.
3. **v** может быть присвоено входное значение, **Value(v) = NAC**.
4. Значение **v** неизвестно, **Value(v)** = **Undef**.

Если **Value(v) ≠ Undef**, алгоритм добавляет **v** в **WorkList** 26

8.2.2 Фаза распространения

```
// Propagation Phase
// Выполнять итерации (до неподвижной точки)
while (WorkList ≠ ∅;)
remove some w from WorkList   //Выбрать
                                очередное имя
for each operation op that uses w
let w be the SSA-name that op defines
if Value(w) ≠ NAC then        //Вычислить снова
                                и проверить не
                                изменилось ли
t ← Value(w)
Value(w) ← result of interpreting op over
           lattice values
if Value(w) ≠ t
then WorkList ← WorkList ∪ {w};
```

27

Фаза распространения проста:

Из **WorkList** извлекается (с удалением) очередное SSA-имя **w**.

Алгоритм анализирует каждую операцию **op**, которая, используя **w**, определяет SSA-имя **w′**.

Если **Value(w′)** = **NAC**, дальнейший анализ **w** не имеет смысла: мы уже скатились в **NAC** и больше не поднимемся из-за монотонности функции.

В противном случае, моделируется выполнение **op** с помощью интерпретации этой операции над полурешетками значений ее операндов.

Если результат отличен от **Value(w)**, он рассматривается как новое значение **Value(w)**, причем **w** добавляется в **WorkList**.

Алгоритм завершается, когда **WorkList** пуст.

11. Алгоритм межпроцедурного распространения констант.

12.3. Межпроцедурное распространение констант

12.3.5. Алгоритм

- ◊ Рассмотрим простой алгоритм межпроцедурного распространения констант. Он похож на алгоритм глобального распространения констант, рассмотренный в разделе 7.2.
- ◊ На фазе инициализации все значения фактических параметров полагаются равными $Undef$.
- ◊ После этого выполняются итерации для каждого фактического параметра a в каждой точке вызова s, в результате чего значения фактических параметров уточняются, а уточненные параметры снова помещаются в $Worklist$.
- ◊ В конце фазы инициализации образуется начальное множество констант, представленных функциями скачка, а в $Worklist$ записывается список всех формальных параметров.
- ◊ **Вторая фаза** многократно выбирает формальные параметры из $Worklist$ и распространяет их.

  Для того, чтобы распространить формальный параметр f процедуры p анализатор находит каждую точку вызова s процедуры p и каждый формальный параметр x (который соответствует фактическому параметру точки вызова s), такой что $f \in Support(J_s^x)$. Затем вычисляется J_s^x и полученное значение сравнивается с текущим значением x.

  Если сравниваемые значения не совпадают, значение x заменяется и x добавляется в $Worklist$.
- ◊ Вторая фаза завершается, так как каждый параметр может принимать всего три значения: T, *константа* и ⊥

12. Алгоритм перемещения кода инвариантного относительно цикла.

6.2.3 Алгоритм перемещения инвариантного кода

◊ **Алгоритм**:

1. Перед заголовком цикла вставить пустой базовый блок (будущий предзаголовок).

Для всех инструкций в теле цикла:

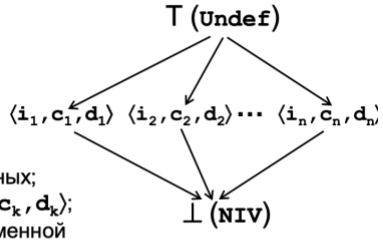
2. Отметить как инвариантные все операнды-константы
3. Отметить как инвариантные все операнды, у которых все определения, достигающие инструкции, находятся вне цикла
4. Отметить как инвариантные все инструкции, все операнды которых отмечены
5. Повторять шаги 2 – 4, пока инвариантные инструкции не перестанут выделяться
6. Переместить все выделенные инструкции в предзаголовок.

13. Алгоритм построения семейств индуктивных переменных.

6.4.3 Обнаружение индуктивных переменных

◊ Рассмотрим переменную **x**.
Возможны следующие варианты:

- ◈ **x** в рассматриваемом цикле ничего не присваивается; тогда над **x** надписывается **Undef**;
- ◈ **x** присваивается значение $c_k*i_k+d_k$, где i_k – одна из основных индуктивных переменных; тогда над **x** надписывается $\langle i_k, c_k, d_k\rangle$; над основной индуктивной переменной i_k надписывается $\langle i_k, 1, 0\rangle$.



- ◈ Если **x**, над которым уже надписано $\langle i_k, c_k, d_k\rangle$ присваивается значение $c_m*i_m+d_m$, где i_m – другая основная индуктивная переменная; тогда старая надпись над **x** заменяется на **NIV**.

◊ Надписывание означает, что существует отображение (функция), которое каждой переменной, встречающейся в исследуемом цикле, ставит в соответствие необходимый элемент рассматриваемой полурешетки. На значениях этой функции будет задана передаточная функция анализа потока данных.

◊ Чтобы начать анализ потока данных необходимо построить начальное отображение (**var**, **val**), где **var** – переменная, а **val** – ее значение. Эти пары помещаются в очередь **Worklist**.

- ◈ Сначала находят **основные** индуктивные переменные. Для этого просматриваются все инструкции цикла и находятся все переменные **i**, которым сначала присваивается целое значение **c** (**i ← c**), а потом ее значения изменяются только инструкциями вида **i ← +, i, d**, или **i ← +, d, i**, где **d** – инвариант цикла.
- ◈ Отметим, что присваивание **i ← c** должно быть **только одно.** Если таких присваиваний несколько, т. е. есть **n** инструкций $i \leftarrow c_k$ с разными c_k, ($k = 1, \ldots, n$), то переменная **i** **расщепляется** на **k** переменных i_k.
- ◈ Значением основной индуктивной переменной **i**, т. е. **val(i)** является «тройка» $\langle i,1,0\rangle$, так как очевидно, что **i = 1*i + 0**. Каждой основной индуктивной переменной **i** соответствует пара (**i**, $\langle i,1,0\rangle$), которая помещается в **Worklist**.

- ◈ Если обнаруживаются инструкции вида
  p ← +, j, c или **q ← *, j, c**,
  где **j** – индуктивная переменная семейства **i**: **val(j)** = $\langle i,a,b\rangle$, а **c** – инвариант цикла, то **p** и **q** – тоже индуктивные переменные семейства **i**, причем
  val(p) = $\langle i,a,b+c\rangle$ и **val(q)** = $\langle i,a*c,b*c\rangle$.
- ◈ Если обнаруживаются инструкции вида **p ← e**, где **e** – выражение, не содержащее индуктивных переменных, то **val(p) = NIV**
- ◈ Всем остальным переменным сопоставляется значение ⊤ (**Undef**)
- ◈ Все пары (**var**, **val**), найденные на первом этапе, кроме пар, у которых **val** = **NIV** помещаются в том порядке, в котором они были найдены в очередь **Worklist**.
- ◈ На этом первый этап алгоритма обнаружения индуктивных переменных заканчивается.

◊ После того, как начальное отображение построено и **Worklist** заполнен, начинается второй этап алгоритма обнаружения индуктивных переменных.

- ◈ Из очереди **Worklist** выбирается очередная пара (**var, val**)
- ◈ Если эта пара имеет вид (**i**, $\langle i,1,0\rangle$), она помещается в список найденных индуктивных, начиная семейство **i**.
  Теперь все индуктивные переменные, значение которых имеет первым членом «тройки» **i**, будут помещаться в это семейство.
- ◈ Если эта пара имеет вид (**j**, $\langle i,a,b\rangle$), выполняется обход ГПУ, чтобы выяснить, не превратится ли **val(j)** в **NIV**. Каждая пара анализируется независимо от других пар. Это связано со структурой полурешетки. В основном уточняются кандидаты в индуктивные переменные, имеющие значение **Undef**.
- ◈ Анализ состоит в том, чтобы просмотреть все базовые блоки, в которых встречается **j** и убедиться, что во всех блоках **val(j)** остается в своем семействе индуктивных переменных. Основную роль при таком анализе играет операция сбора для функции **val**.

14. Алгоритм исключения умножений при вычислении индуктивных переменных.

- ◊ Как вычислять **j**, используя сложение вместо умножения?
- ◊ Сделать это очень просто:
  Для производной индуктивной переменной **j** из семейства основной индуктивной переменной **i**: **val(j)** = ⟨**i,c,d**⟩ необходимо выполнить следующий простой **алгоритм**:
  1. Создать новые переменные **s** и **k**, и в предзаголовке цикла выполнить присваивания **s = c*i + d;** и **k = c*h;**
  2. В теле цикла заменить определение **j = e** на **j = s;**
  3. В теле цикла после присваивания **i = i + h** вставить присваивание **j = j + k**;

15. Алгоритм *Mark & Sweep* удаления бесполезного кода.

5.2.6. Проход *Mark* (псевдокод)

```
Mark( )
WorkList ← ∅;
for each инструкции i (x ← op, y, z ::: пометка)
      убрать пометку у i
      if (i полезная) пометить i
      WorkList ← WorkList ∪ {i}
while (WorkList ≠ ∅)
      remove i from WorkList
      if (def(y) не помечена)
            пометить def(y)
            WorkList ← WorkList ∪ {def(y)}
      if (def(z) не помечена)
            пометить def(z)
            WorkList ← WorkList ∪ {def(z)}
      for each block b ∈ RDF(block(i))
            пусть j ветвь, оканчивающаяся в b
            if (j не помечена)
                  пометить j
                  WorkList ← WorkList ∪ {j}
```


5.2.6 Проход *Sweep*

```
Sweep( )
for each instruction i
      if (i is unmarked)
            if (i is a branch)
                  rewrite i with a jump
                  to i's nearest marked
                  postdominator
            if (i is not a jump)
                  delete i
```

◊ На втором проходе (*Sweep*) в каждый блок, с которого начинается непомеченная ветвь, помещается безусловный переход на его помеченный постдоминатор.
Это правильно, так как если ветвь не помечена, потомки блока вплоть до его непосредственного постдоминатора, не могут содержать полезных инструкций, так как иначе они были бы помечены.

16. Алгоритм *Mark & Sweep* удаления недостижимого кода

5.3.2 Анализ достижимости

- ◊ Проход *Mark* сначала помечает каждый блок *b* как «недостижимый», потом он начинает обход ГПУ с *Entry* и помечает как «достижимый» каждый блок, которого он может достичь.
- ◊ Если все ветвления и переходы определяются однозначно, то все блоки, помеченные как недостижимые, действительно недостижимы и могут быть удалены на проходе *Sweep*.
- ◊ В случае неоднозначных условий ветвления, компилятор должен сохранить любой блок, достижимый ветвлением или переходом.

17. Алгоритм оптимизации потока управления.

5.4. Оптимизация потока управления

5.4.3. Функция `Clean()`

```
Clean()
while ГПУ продолжает изменяться
compute
  Postorder
  OnePass()

OnePass()
for each block Bi            || in postorder
   if (Bi оканчивается ветвлением)
     if (обе цели одинаковы)
        заменить ветвление на переход                    /* 1 */
   if (Bi оканчивается переходом на Bj)
     if (Bi пуст)
        заменить все переходы на Bi переходами на Bj     /* 2 */
     if (Bj имеет только одного предшественника)
        совместить Bi и Bj                               /* 3 */
     if (Bj пуст и оканчивается ветвлением)
        заменить Bi переход
        на копию ветвления из Bj                         /* 4 */
```

5.4. Оптимизация потока управления

5.4.3. Функция `Clean()`

◊ Функция **`Clean()`** многократно вызывает функции **`Postorder()`** (обычная нумерация блоков) и **`OnePass()`** (однократный проход), выполняя последовательность преобразований 1 – 4 итеративно до тех пор, пока ГПУ оптимизируемой процедуры продолжает изменяться.

◊ В начале каждой итерации выполняется новая нумерация блоков, так как после каждого применения четверки преобразований ГПУ может сильно измениться.

18. Алгоритм анализа потока данных на основе областей.

10.2. Анализ потока данных на основе областей

10.2.1. Схема анализа потока данных на основе областей

◊ **<u>Первый этап</u>. Построение передаточных функций всех областей (от листьев к корню дерева управления).**

- Сначала обрабатываются области-листья (отдельные блоки):

  для каждой области-листа R, состоящей из блока B,

  $f_{R,In[B]} = I$ (тождественная функция)

  $f_{R,Out[B]} = f_B$ (передаточная функция блока B).

- Перемещение вверх по иерархии:
  - R **– область-тело**: ребра, принадлежащие R, образуют ациклический граф на подобластях R, что позволяет при вычислении передаточных функций использовать топологический порядок областей.
  - R **– область-цикл**: учитывается только влияние обратных ребер, ведущих к заголовку R.
- В конце обработки достигается вершина иерархии и вычисляются передаточные функции области R_n, представляющей собой весь граф потока.

◊ **<u>Второй этап</u>. Анализ иерархии областей (от корня к листьям дерева управления).**

- Области просматриваются в обратном порядке, начиная с области R_n и далее, опускаясь вниз по иерархии. Для каждой области вычисляются значения потока данных на входе.
- Чтобы получить значения потока данных на входе $R \in R_n$ используется передаточная функция $f_{R_n,In[R]}$
- Вычисления повторяются, до тех пор, пока не будут достигнуты области-листья (базовые блоки).

19. Алгоритм «агрессивная вставка».

◊ Агрессивная вставка состоит в выполнении нескольких фаз клонирования и вставки.
Такая многофазная структура выбрана потому, что очень трудно предсказать влияние каждого клона и каждой вставки на оптимизацию. Выполняя все вставки в одну фазу, трудно учесть все особенности вставляемой процедуры прежде всего потому, что они выявляются в процессе первого этапа вставки.

◊ Управление фазой вставки обеспечивается введением параметра *бюджет*. Бюджет (**B**) это оценка возрастания времени компиляции в связи с увеличением длины вызывающей процедуры в результате данной вставки.

◊ Бюджет вычисляется по формуле

```
B = Q*sizeof(R)
```

По умолчанию для каждой вставляемой процедуры **Q** полагается равным 2, хотя, как показал опыт эксплуатации, среднее значение **Q** = 1.2 (длина процедуры возрастает примерно на 20%).

◊ Пользователь имеет возможность вручную уточнять бюджет в любом направлении.
В начале алгоритма бюджет распределяется между отдельными фазами клонирования и вставки (распределение бюджета тоже может уточняться пользователем).

◊ Затем начинается *основной цикл*, который состоит в выполнении клонирований и вставок и заканчивается либо при исчерпании бюджета, либо по истечении лимита времени (**limit**), отведенного на выполнение данной группы вставок.
Клоны сохраняются в файле (базе данных) **D** для ускорения процесса.

20. Алгоритм построения максимальной SSA-формы.

7.2.2. Базовый алгоритм построения *SSA*-формы

◊ **Вход**: программа в промежуточном представлении

◊ **Выход**: промежуточное представление программы в *SSA*-форме

◊ **Метод**: Выполнить следующие действия:

(1) *Вставить φ-функции*:
в начало каждого блока B, у которого $|Pred(B)| > 1$ вставить φ-функцию вида $y = \varphi(y, y, \ldots)$ для каждого имени y, которое либо определяется, либо используется в B.
Вставленная φ-функция должна иметь по одному аргументу для каждого $B' \in Pred(B)$:
Порядок вставляемых φ-функций несуществен

(2) *Переименовать переменные*:

(a) вычислить достигающие определения; при этом только одно определение будет достигать любого использования:
это гарантируют вставленные φ-функции, которые тоже являются определениями;

(b) переименовать каждое использование (как основных, так и временных) переменных таким образом, чтобы новое имя соответствовало единственному определению, которое достигает его.

(3) *Отсортировать определения*,
достигающие каждой φ-функции и для каждой φ-функции обеспечить соответствие имен ее аргументов путям, по которым определения этих аргументов достигают блока, содержащего указанную φ-функцию.

21. Алгоритм построения частично-усеченной SSA-формы.

Алгоритм построения множеств *Globals* и *Blocks*(*x*)

```
Globals = ∅;
for each variable x do
    Blocks(x) = ∅;
    for each block B do {
        for each instruction i ∈ B do {
        || пусть команда i имеет вид: x ← op, y, z
            if y ∉ def_B then Globals = Globals ∪ {y};
            if z ∉ def_B then Globals = Globals ∪ {z};
            if x ∉ use_B then def_B = def_B ∪ {x};
            Blocks(x) = Blocks(x) ∪ {B}
            }
        }
    }
```

Globals – множество глобальных имен

Blocks – множество базовых блоков, в которых определяется x

7.3 Построение частично усеченной SSA-формы

7.3.2. Размещение φ-функций

Алгоритм размещения φ-функций

- **Вход**: исходный ГПУ
- **Выход**: преобразованный ГПУ
- **Метод**: Выполнить следующие действия

```
for each name x ∈ Globals do {
    WorkList = WorkList ∪ Blocks(x);
    for each block B ∈ WorkList do
        for each block D ∈ DF(B) do {
            if (D не содержит φ-функции для x )
                вставить φ-функцию для x в D;
            WorkList = WorkList ∪ {D};
```

Чтобы учесть новое определение x, реализуемое вставленной φ-функцией

22. Алгоритм переименования переменных при построении SSA-формы.

7.3.4. Переименование переменных

Алгоритм

- ◊ **Вход**: программа с размещенными φ-функциями
- ◊ **Выход**: программа, в которой каждой переменной сопоставлено ее *SSA-имя*.
- ◊ **Метод**: Сначала (в основном алгоритме) инициализируются стеки и счетчики, после чего из корня дерева доминаторов n_0 вызывается рекурсивная функция *Rename*.

  Rename обрабатывает блок, рекурсивно вызывая его последователей по дереву доминаторов.

  Закончив обрабатывать очередной блок, *Rename* выталкивает из стеков все имена, помещенные в них во время обработки блока.

  Функция *NewName*, манипулируя со счетчиками и стеками, в случае необходимости создает новые имена.

Функция $Rename(B)$:

```
for each φ-function ∈ B: x = φ(…) do
     rename x as NewName(x);
for each instruction ∈ B: x ← op, y, z do{
     rewrite y as top(stack[y]);
     rewrite z as top(stack[z]);
     rewrite x as NewName(x);
for each successor of B in the flowgraph do
     fill in φ-function parameters;
for each successor S of B in the
     dominator tree do Rename(S)
for each φ-function ∈ B: x = φ(…)do
          Pop(stack[x]);
for each instruction ∈ B: x ← op, y, z do
          Pop(stack[x]);
```

23. Алгоритм глобальной нумерации значений.

9.4.3 Рекурсивный алгоритм глобальной нумерации значений

◊**Вход**: (1) граф потока управления $\langle N, E\rangle$, дерево доминаторов DT
(2) множество Val значений переменных, констант и выражений

◊**Выход**: отображение VN: $Val \to N \cup \{0\}$ (N – множество натуральных чисел), ставящее в соответствие каждому значению его номер: натуральное число или 0.

◊**Метод**: Применить к корню DT рекурсивную процедуру $DBGVN$ (*Dominator Based Global Value Numbering*)

(1) Обработка φ-функций

```
procedure DBGVN(Block B)
    Отметить начало новой области имен
    ||Обработка φ-функций
    for each p ∈ B, где p – φ-функция вида "n ← φ(…)"
        if p бессмысленна или избыточна
            поместить номер значения p в VN[n]
            удалить p
        else
            VN[n] ← n;
        добавить p в ТЗ
```

(2) Обработка остальных инструкций

```
procedure DBGVN(Block B)
    for each a∈B где a – присваивание вида "x ← op, y, z"
        заменить y на VN[y] и z на VN[z]
        expr ← op, y, z          || expr – вход в ТЗ
        if expr может быть упрощено до expr′
            Заменить a на "x ← expr′"
            expr ← expr′
        if expr имеется в ТЗ с номером v
            VN[x] ← v
            Удалить a
        else Добавить expr в ТЗ с номером x VN[x] ← x
```


(3) Окончание обработки блока B и переход к обработке его дочерних блоков (по дереву доминаторов)

```
procedure DBGVN(Block B)
    for each s ∈ Succ(B)
        скорректировать входы φ-функций в s
    for each дочернего блока c узла B по дереву доминаторов
        DBGVN(c)                 || рекурсивный вызов
    Очистить ТЗ при выходе из области (стэк)
```

24. Алгоритм генерации кода для размеченных деревьев выражений.

1. Генерация машинного кода для внутреннего узла с меткой k и двумя дочерними узлами с **равными** метками (в этом случае метки обоих дочерних узлов равны $k-1$):
   (a) Рекурсивная генерация кода для правого дочернего узла, используя регистры R_{b+1}, …, R_{b+k-1}. Результат правого дочернего узла помещается в регистр R_{b+k-1}.
   (b) Рекурсивная генерация кода для левого дочернего узла, используя регистры R_b, …, R_{b+k-2}. Результат левого дочернего узла помещается в регистр R_{b+k-2}.
   (c) Генерация команды
   OP R_{b+k-1}, R_{b+k-2}, R_{b+k-1},
   где **OP** – операция в узле с меткой k.

2. Генерация кода для внутреннего узла с меткой k и двумя дочерними узлами с **разными** метками (в этом случае один из дочерних узлов («*большой*») имеет метку k, а второй («*маленький*») – метку $m < k$):
   (a) Рекурсивная генерация кода для ***большого*** узла, используя k регистров R_b, ..., R_{b+k-1};
   результат получается в регистре R_{b+k-1}.
   (b) Рекурсивная генерация кода для ***маленького*** узла, используя m регистров R_b, ..., R_{b+m-1};
   результат находится в регистре R_{b+m-1}.
   $m < k$, следовательно при вычислениях не используются ни регистр R_{b+m}, ни какой-либо иной регистр с большим номером.
   (c) Генерируем команду
   OP R_{b+k-1}, R_{b+m-1}, R_{b+k-1} или **OP** R_{b+k-1}, R_{b+k-1}, R_{b+m-1}
   в зависимости от того, является большой узел правым или левым.

3. Для листа, представляющего операнд x, при использовании базы b генерируем команду **LD** R_b, x.

25. Генерация кода с использованием алгоритма динамического программирования.

13.7.3 Алгоритм динамического программирования

◊ Алгоритм динамического программирования состоит из трех фаз.

(1) В восходящем порядке для каждого узла n дерева выражения T вычисляется массив стоимости C, i-й элемент которого $C[i]$ представляет собой оптимальную стоимость вычисления поддерева S с корнем n с при условии, что для вычисления имеется i ($1 \leq i \leq r$) доступных регистров
(r – число регистров целевой машины)

(2) Обход дерева выражения T с использованием векторов стоимости для определения, какие из поддеревьев T должны быть вычислены в память.

(3) Обход каждого дерева с использованием векторов стоимости и связанных с ними команд для генерации конечного целевого кода. Первым генерируется код для поддеревьев, вычисляемых в память.

◊ Каждая из фаз может быть выполнена за время, линейно пропорциональное размеру дерева выражения.

26. Алгоритм распределения регистров методом линейного сканирования.

◊ На псевдокоде

```
active ← {}
for each ИЖ i в порядке возрастания начал
     ExpireOldIntervals(i)//окончание
                    старых интервалов
if length(active) = R then
     SpillAtInterval(i)  //слив интервала i
else
     register[i] ← регистр, удаляемый из
          пула свободных регистров
     добавить i в active, отсортированный
          по возрастанию концов ИЖ

ExpireOldIntervals(i)
foreach interval j in active, in order of
increasing end point
if endpoint[j] ≥ startpoint[i] then
     remove j from active
     add register[j] to pool of free
               registers
     return

SpillAtInterval(i)
spill ← last interval in active
if endpoint[spill] > end point[i] then
     register[i] ← register[spill]
     location[spill] ← new stack location
     remove spill from active
     add i to active, sorted by increasing
          end point
else
location[i] ← new stack location
```

```
SpillAtInterval(i)
spill ← last interval in active
if endpoint[spill] > end point[i] then
     register[i] ← register[spill]
     location[spill] ← new stack location
     remove spill from active
     add i to active, sorted by increasing
          end point
else
location[i] ← new stack location
```

27. Модифицированный алгоритм линейного сканирования.

Как и обычный, но учитывает «дыры» в интервале жизни переменных, перемещая их то в список Active, то в Inactive

28. Трехфазный алгоритм раскраски графа конфликтов.

14.3.10 Алгоритм раскраски графа конфликтов

◊ ***1 фаза. Установление порядка рассмотрения узлов***

узлы по очереди удаляются из ГК и помещаются в стек.

- Узел ГК называется неограниченным, если его степень $< n$, и ограниченным, если его степень $\geq n$.
- Сначала в произвольном порядке удаляются неограниченные узлы вместе с дугами, соединяющими их со смежными узлами, при этом степень части смежных узлов понижается, так что некоторые из ограниченных узлов после удаления могут стать неограниченными.
- Если после удаления всех неограниченных узлов в ГК все еще остаются узлы, то все они ограничены. Для каждого из ограниченных узлов вычисляется их степень (количество смежных узлов).
- Ограниченные узлы удаляются из графа и помещаются в стек в порядке возрастания степени.

В конце фазы граф конфликтов пуст, а все его узлы (ИЖ) находятся в стеке в некотором порядке.

◊ ***2 фаза. Раскраска узлов***

распределитель восстанавливает ГК, выбирая из стека очередной узел l и раскрашивая его в цвет, отличный от цвета смежных узлов. Если оказывается, что все цвета использованы, узел l остается нераскрашенным.

В конце фазы стек пуст, а ГК восстановлен и часть его узлов раскрашена.

◊ ***3 фаза. Проверка на окончание процесса раскраски***

Если нераскрашенных узлов не осталось, алгоритм завершается. Если часть узлов ГК осталась нераскрашенной, то для каждого такого узла либо генерируются команды сброса (*слив*), либо интервал жизни, соответствующий узлу *расщепляется* (на два или более подинтервалов), после чего ГК перестраивается с учетом слива и/или разделенных узлов. После перестройки ГК делается переход на первую фазу.

◊ Ключевым моментом является порядок в котором узлы ГК помещаются в стек.

Наиболее распространенный эвристический критерий слива узла – минимум отношения

$$\frac{цена\_сброса}{степень\_узла}$$

29. Планирование базовых блоков. Алгоритм планирования списка.

15.5.3 Алгоритм планирования списка

Вход: вектор ресурсов машины $R = [r_1, r_2, ...]$, граф зависимостей по данным $G = (N, E)$; каждой операции $n \in N$ сопоставлена таблица резервирования ресурсов RT_n; каждое ребро $e = n_1 \rightarrow n_2$ из E помечено задержкой d_e: n_2 можно выполнить не ранее чем через d_e тактов после n_1.

Выход: план S, который отображает операции из N на временны′е интервалы, когда удовлетворяются все ограничения по ресурсам для этих операций и когда может начинаться их выполнение.

Метод: выполнить программу (псевдокод):

```
RT = пустая таблица резервирования ресурсов;
for (каждый n ∈ N в приоритетном
топологическом порядке) {
/* Вычисление самого раннего времени S(n) = s
   возможного начала команды n на основе заданного
   S(p)- времени начала предыдущей команды p*/
      s = max(e = p→n)∈E (S(p) + de);
/* Вычисление задержки начала выполнения команды n
   до тех пор, пока не будут доступны все ресурсы.*/
      while(∃ i: RT[s + i]+ RTn[i] > R[i])
            s = s + 1;
      S(n) = s;
/* Корректировка таблицы ресурсов для учета
   ресурсов, используемых командой n */
      for (всех ресурсов i)
            RT[s + i] = RT[s + i] + RTn[i]
```

30. Алгоритм глобального планирования кода на основе областей.

15.7.3 Алгоритм глобального планирования на основе областей

- ◊ **Вход**: граф потока управления и описание машинных ресурсов.
- ◊ **Выход**: план S, отображающий каждую команду на базовый блок и интервал времени.
- ◊ **Метод**: выполнить следующую программу:

```
for (каждая область R в топологическом порядке, при котором
    внутренние области обрабатываются раньше внешних) {
    Вычисление зависимостей данных;
    for (каждый базовый блок B области R в приоритетном
    топологическом порядке ) {
        CandBlocks = ControlEquiv(B) ∪ DominatedSucc (ControlEquiv (B));
        CandInsts = готовые команды из CandBlocks;
        for (t = 0,1,..., пока не будут спланированы все
        команды из B) {
            for (каждая команда n из CandInsts в приоритетном порядке)
                if (n не имеет конфликтов ресурсов в момент t) {
                        S(n, B, t);
                        Обновление имеющихся ресурсов;
                        Обновление зависимостей данных;
                }
                Обновление Candlnsts;
            }
        }
}
```

31. Алгоритм программной конвейеризации.