プログラミング言語処理系論 (11)
Design and Implementation of Programming Language Processors

佐藤周行

（情報基盤センター/電気系専攻融合情報学コース）

今回の予定

- ローカル最適化
  - Value Numbering
- 大域的データフロー解析をもとにした最適化
  - フローグラフとその解析
  - Motivating Examples
  - データフロー方程式
  - さまざまな問題の定式化
  - データフロー方程式の解き方

ローカル最適化：
基本ブロック内最適化

- 最適化のうち、「冗長計算の除去」を考える。

```
x = 3;
y = 4;
z = x + y;
y = 4;
```

- コード生成の途中で結構出てくる。
- どうやって冗長だとわかるか？

基本ブロック内最適化

- 通常のプログラムでは
  - 変数は複数の場所で計算され、それぞれが値のソースになり得る。

```
if (f(n)):
    x = 1
else:
    x = 2
print(x)
```

  - これ以上の解析がすすまない？

- 最初のステップとして
  - 制御の流れが一直線のものを考える(基本ブロック)。つまり
  - プログラムの実行開始点から実行終了点まで、ジャンプすることなく実行される
- ことを前提として、「冗長性の除去」問題を解いてみる

ローカル最適化

- 言葉の定義を与えておく

Def. [Basic Block] A **basic block** is a straight-line sequence of code with only one entry point and only one exit

- 基本ブロック内で行われる最適化を「ローカル最適化」と言う

Value Numbering

- プログラムはRTLで書かれているとする
  - 定数もいったんレジスタに代入する
  - レジスタ間の単項演算、2項演算とMOVだけ
- Value numbering
  (共通部分式を全部探し出す最適化)

```
m=i+3          m=i+3
j=i      →     j=i
k=j+3          k=m
```

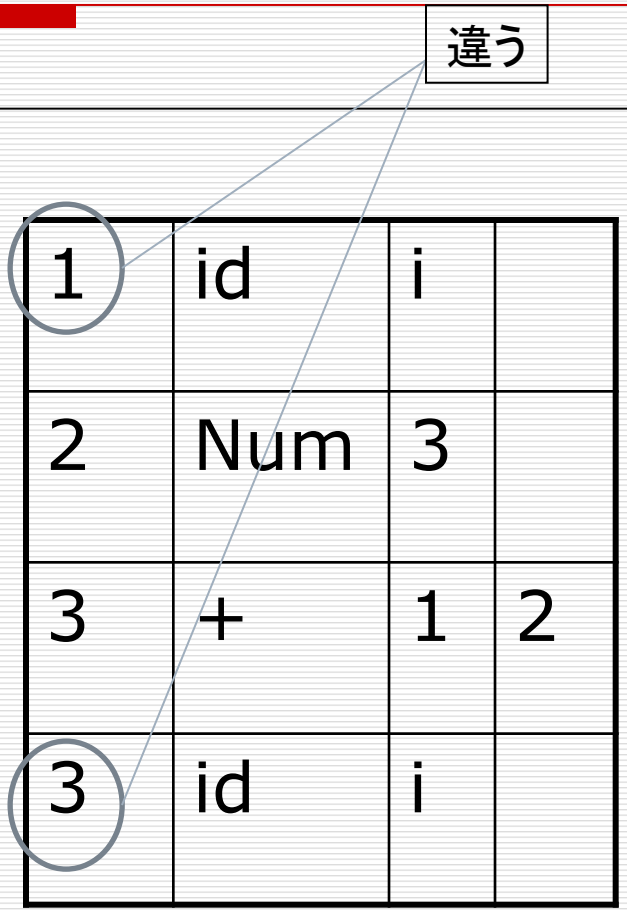
Value Numbering

- 各変数、リテラルに番号 (quantity)をつける
- 計算は、それらの間の演算として表され、結果は変数(レジスタ)に格納される　⇒　計算自体に番号 (quantity) が付けられる
- 新たな代入や計算が現れた時に、過去に現れた番号で一致するものがあるかどうかを調べていく
- 方法論は単純だが、3 address codeとの相性が良く、昔からよく使われる

例でみていく

i=i+3

1. 各変数、リテラル、式に番号をつける
2. 式が出現するたびに、式の番号を計算する。新しい番号である場合もあるし、既出のものであることもある
3. 最後に番号に対応する式を代表元として出力する



計算の例

m=i+3
j=i
k=j+3

(as an ODC program)

mov 3 r9
add i0 r9 i1
mov i0 i2
mov 3 r11
add i2 r11 i3

(as an RTL representation)

計算の例

m=i+3

j=i

k=j+3

k=j+3 の部分で、j+3をたどっていくと
Jの番号– 1
3の番号– 2
+(1) (2)の番号– 3

となって番号3がでてくるはずである。

?→3	+	[j]=1	[3]=2

1	Id	i	
2	lit	3	
3	+	1	2
3	Id	m	
1	Id	j	
3	Id	k	

```
mov 3 r9
add i0 r9 i1
mov i0 i2
mov 3 r11
add i2 r11 i3
```

Qty	op	operand 1	operand 2
1	lit	3	
1	r9	1	
2	i0		
3	add	2	1
3	i1	3	
2	i2	2	
1	r11	1	
3	add	2	1
3	i3	3	

結果として

m=i+3	mov 3 r9
j=i	add i0 r9 i1
k=m	mov i0 i2
	~~mov r9 r11~~
	mov i1 i3

代入の効果

- 変数に代入したら、以前の変数とは別扱いにする

g=x+y
h=u-v
i=x+y
x=u-v
u=g+h
v=i+x
w=u+v

問： 4つのxの出現のうち、同じquantityを持つべきものはどれか？

Value Numberingの適用

- Common Subexpression elimination
  - Quantityが同じ計算をしているものが再度現れたら、最初の計算結果を保持している変数からの代入に置換可能

```
mov 1 r1
mov 2 r2
add r1 r2 r3
mov 2 r4
~~add r1 r4 r5~~  → mov r3 r5
```

Value Numberingの適用

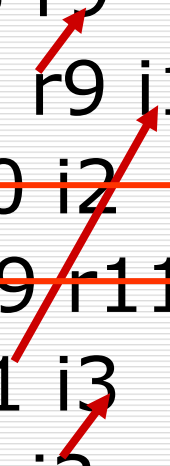
- Redundant assignment elimination
  - Assignmentが、同じquantityを持っているどうしで行われるなら削除可能

    mov 1 r1
    mov 1 r2
    ~~mov r1 r2~~ →

Value Numberingの適用

- Dead Code Elimination
  - BB内でしか用いられないと分かっている変数で、最終結果に影響を及ぼさないものは削除可能

```
mov 3 r9
add i0 r9 i1
mov i0 i2
mov r9 r11
mov i1 i3
return i3
```



- (問題14)以下の命令列に対してvalue numberingを行え。授業中で概説したアルゴリズムを完成させよ。できるならばプログラムを書いてみよ。

g=x+y	add i0 i1 i2
h=u-v	sub i3 i7 i5
i=x+y	add i0 i1 i6
x=u-v	sub i3 i7 i0
u=g+h	add i2 i5 i3
v=i+x	add i6 i0 i7
w=u+v	add i3 i7 i8

- （問題14の続き）以下のコードを考える。
  def f(a,b) {return ((a+b)+3)*(3+(b+a))};
- RTLは以下のようになるだろう（名前a → i0, b → i1）

```
add i0 i1 r2
mov 3 r3
add r2 r3 r4
mov 3 r5
add i1 i0 r6
add r5 r6 r7
mul r4 r7 r8
return r8
```

- この時、value numberingを行え。rで始まる変数はreturnした後使用しないという仮定を置くと、命令のいくつかを削除することができる。これをやってみよ。（dead code elimination）

問題14の注意

- アルゴリズムをプログラムとして実装する必要はない。(プログラムを書いて、それが正しく動けばなおよい)ただし
  - Commutativeな演算については考慮の対象とすること
  - Quantityに対して最終的に割り当てられる変数名を決定するアルゴリズムを含むこと
  - Dead code eliminationやredundant assignment eliminationを行うアルゴリズムを含むこと

- 教材 (OPT.zip)では、実行ファイルrtltestを-Oで実行するとvalue numberingが提供されています。valnum.cに、ASTを対象にしたvalue numbering, rtlvalnum.cにRTLを対象にしたvalue numberingのコードが提供されています。
- このソースファイルを読んで、内容を解読するのでもよい（読みにくいかも）。（バグを見つければなおよい）

基本的なデータ構造

GCC: (10系)

```
struct {
   int qty;
   int kind; /* id | literal | expression */
   union {id_desc  xxx;
              literal_desc yyy;
              expression_desc zzz}
}
```

簡単なRTL（教材のもの）

```
struct {
   int qty;
   int opcode; /* MOV|ADD|SUB|… */
   int op[3];
}
```

実際のコンパイラでの実装

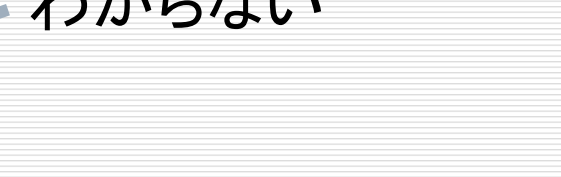
- gcc/cse.cc
  - Common Subexpression Eliminationのローカル版
  - Noe: 12系は、qtyとexpを分離し、後者をhashに格納するなどして、一言では説明が難しいようなデータ構造に変更されている
  - 勉強したいなら10系が良いかもしれない
- 結構効果がある
- 実装には、ハッシュ表の利用が有効
  - hashを使うと、変数のアップデートに対応するコードを書かないといけない

Value Numberingの限界

- 以下のようなコードではどうか？

```
    j  =k
    ifz  j  L1
    m=i+3
    j  =i
     ifz  j L2
L1: k  =j+3
```

このjがどの値を持ってきているかわからない



- ジャンプがあるところから全部やり直し

Value Numberingの限界（続き）

- 変数の値を「以後使わない」と判断するには、特別な仮定が必要だった。（rで始まるとか…）
- ソースプログラムで定義されている変数にはこの仮定は使えない
  - 将来、どこで参照されるかは、関数全体を見ないと判断できない
- Dead code eliminationの適用範囲が制限される
- この制限を緩和するには、関数全体のフローを解析する必要がある
  - 全体を解析して、「この代入は不要である」と判断できればdead code eliminationの対象になる

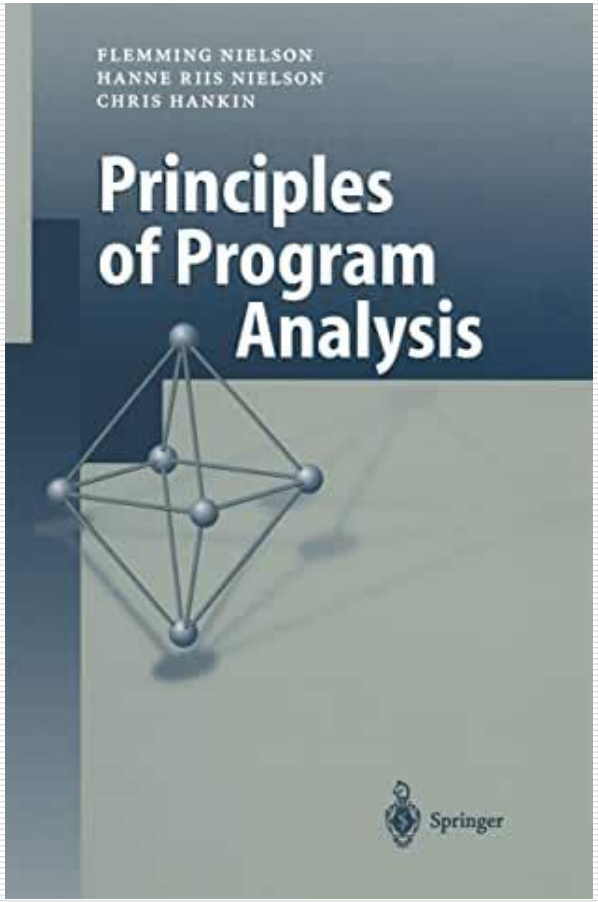
- この問題を解決する(Global Value Numbering)
- 変数のアップデートの情報を変数自身に埋め込む
  - 同じ変数でも、異なるものがあり得る
- GVNにはSSAが本質的(後でやる)
- クラシカルな解決法がある
  - データフロー解析

プログラム解析理論

- プログラム全体を静的に解析する
  - プログラム意味論と並ぶ花形（だった？）
- Static Analysis of Programs
  - ソフトウェア工学の理論のうちで中心的なもののひとつ
  - Slicing, abstract interpretationなどの理論の発展
- プログラム最適化の理論が中心的な話題を提供するのは昔も今も変わらない

より深く勉強したい人に

- Principles of Program Analysis, 1999, 2008(2nd ed.)



Basic Block （基本ブロック）

Def. [Basic Block] A **basic block** is a straight-line sequence of code with only one entry point and only one exit

(1) 先頭には、制御が移ってくる（ラベルでマークしていることが多い）
(2) 最後では、制御がどこかに移る（ジャンプ命令での行き先は高々2か所）
(3) それ以外では、制御が途中から移ってくることはないし、制御が他に移ることもない

データフロー解析をもとにした大域最適化

- 以下、基本ブロックの集合から作成した次のグラフを考える。
  - ノード
    基本ブロック
  - エッジ
    基本ブロックx →基本ブロック y

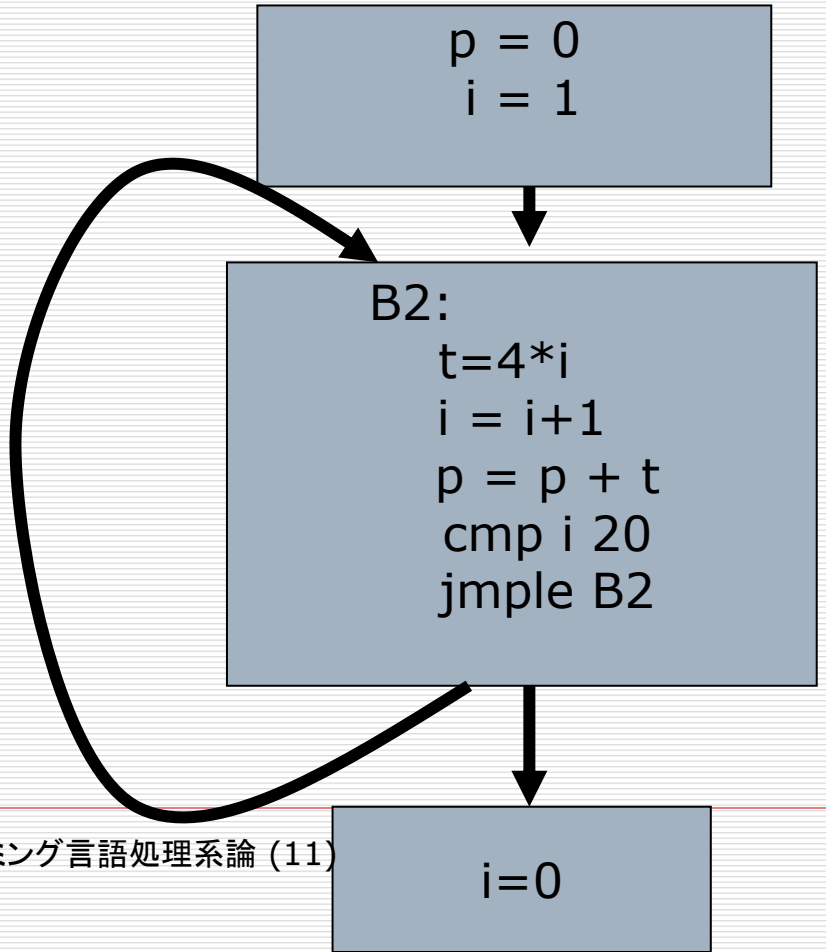
    1. xのジャンプ先がyの先頭のラベルであるとき
    2. xからyにfall throughするとき

  Def. 上のグラフをFlowgraphと言う

基本ブロックのフローグラフの例

- 命令列

```
    p = 0
    i = 1
B2:
    t=4*i
    i = i+1
    p = p + t
    cmp i 20
    jmple B2
B3:
    i=0
```



いくつかの性質

- Def.　start block
  開始ブロックをひとつ指定しておく。
- Def. Dominate
  ブロックdがブロックnをdominateするとは、スタートブロックを始点とし、nに到達するすべてのパスがdを含むことをいう。このとき、dはnのdominatorであるともいう。
  - nに行くには必ずdを通らなければならない
- Dominatorは、フローグラフを用いた解析の基本
  - SSAなどのフロー情報を埋め込んだ中間言語の設計
  - ループの検出
- 基本的な性質
  - Start blockはすべてのノードのdominatorである
  - 自分自身は自分のdominatorである

- ノードSのdominatorの集合Dom(S)は、以下の方程式を満たす

$$Dom(S) = \{S\} \cup \bigcap_{P \in \{S\text{のpredecessor}\}} Dom(P)$$

ただしDom(S0) = {S0}

- 方程式を以下の方法で解く

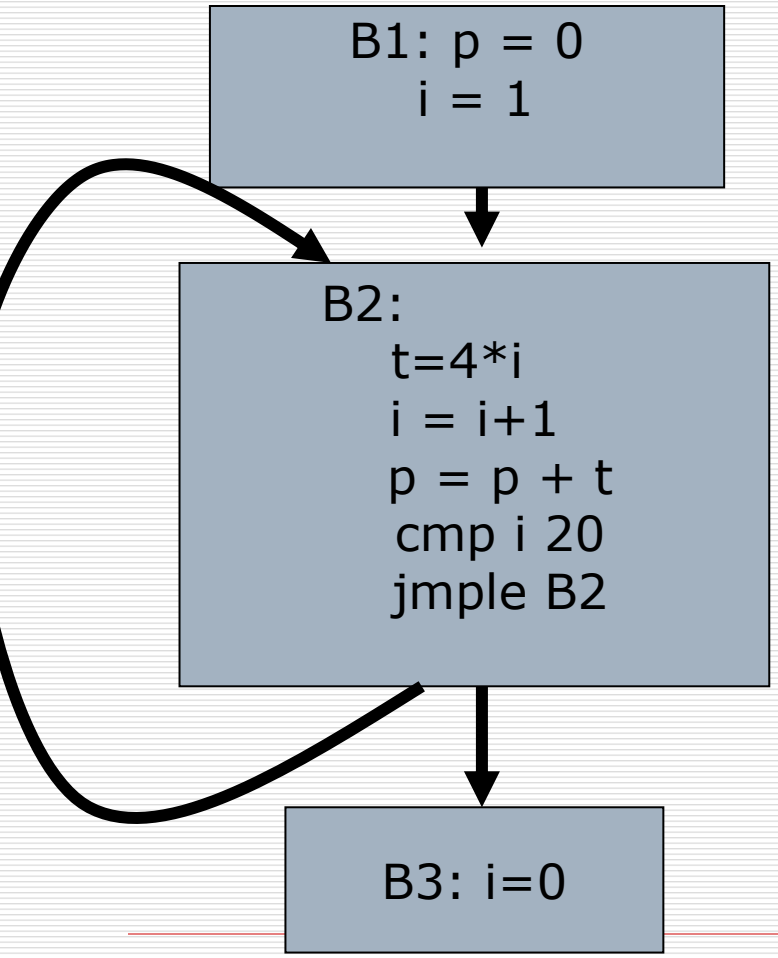
Dom(S_0) := {S_0}; Dom(S) = ALL; // Initialize
while (変化が観測される) {
　S_0以外のノードすべてについて

$$Dom(S) := \{S\} \cup \bigcap_{P \in \{S\text{のpredecessor}\}} Dom(P)$$

}

- Dominatorを効率よく求めるアルゴリズムは昔から知られています（e.g. Lengauer-Tarjan）

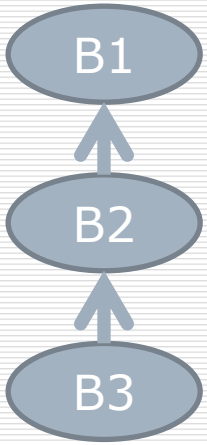
計算の例



- Initialize:
  Dom(B1) = {B1}
  Dom(B2) = {B1, B2, B3} ;; all
  Dom(B3} = {B1, B2, B3} ;; all
- 1st iteration:
  Dom(B1) = {B1}
  Dom(B2) = {B1, B2}
  Dom(B3) = {B1, B2, B3}
- 終了

Domの性質

- Dominator Tree
  - 2つのノードSとTに関し、SがTをdominateし、かつTがSをdominateすることはあり得ない
  - Dominateという関係は木構造を作る（この木構造をdominator treeという）



課題15

- 前ページの命題を証明せよ:
  - 2つのノードSとTに関し、SがTをdominateし、かつTがSをdominateすることはあり得ないことを証明せよ
  - Dominateという関係は木構造を作ることを証明せよ

Edgeの分類

Def.　back edge
　　自分のdominatorにのびるエッジ

Def. forward edge
　　自分のdominatorからのびているエッジ

Def. cross edge
　　これ以外をcross edgeという

Def. natural loop
　　back edge b: n→dが与えられたとき、d、dを通らずにnに到達可能なノードとそのエッジ全体を「bから定まるnatural loop」という

課題16:プログラム

- 以下の(疑似)プログラムを考える。

```
while () {
  if (S1) {S2};
  while (S3) {
     while () {
        switch (S4) {
        case 0: continue;
        case 1: S5; break;
        default: S6; break;
        }
        if (S7) break;
     }
     switch (S8) {
     case 0: continue;
     case 1: S9; goto Label;
     default: return S10;
     }
  }
Label:
}
```

（課題16続き）
次のデータ構造を考える

BB[1] = {succ = {2,3}, pred={9}}

BB[2] = {succ = {3}, pred={1}}

BB[3] = {succ = {4}, pred={1,2,4,8}}

BB[4] = {succ={3,5,6},pred={3,7}}

BB[5] = {succ={7}, pred={4}}

BB[6] = {succ={7}, pred={4}}

BB[7] = {succ={4,8},pred={5,6}}

BB[8] = {succ={3,9,10}, pred={7}}

BB[9] = {succ={1}, pred={8}

BB[10] = {succ={}, pred={8}}

課題16

(1) 前述のデータ構造が、SnがBB[n]に対応するフローグラフになっていることを確認せよ。

(2) S1がentry pointであるとして、各ブロックのdominatorを計算するアルゴリズムを設計せよ(プログラムまでは書く必要がない)。さらに、それに従って、前ページのスライドにあげたフローグラフの各ブロックのdominatorの集合を列挙せよ。

(3) 上の結果を用いてdominator treeを計算するアルゴリズムを設計せよ(プログラムまでは書く必要がない)。さらに、それに従って前ページのスライドにあげたフローグラフのdominator treeを計算せよ。

(4) backedgeをすべて挙げ(4個)、それに対応する(自然)ループ構造を元の疑似プログラム上で示せ。

プログラムを書くならば…

- プログラムを書くことを推奨します。アルゴリズムを示せば、プログラムの詳細を記載する必要はありません。結果のプリントだけでOk。

- 集合を扱うことは、現代的なプログラミング言語では自然にできますから、それを活用するとよい
  - C++, Java, Python, …
- 集合に対応するiteratorも用意されているのが普通なので、それを利用するとよい
- Cでプログラムを書くときは、集合とその操作を実現するデータ構造の設計から始めるとよい (GNU Cでは、ビット演算を多用している)

基本ブロックをまたぐ解析と最適化

- 基本ブロックを越えて情報が伝播していくことが観測できれば、それに対応した最適化を考えることができます
- 通常は、関数を単位として、その中で最適化をします
- その時の基本的なツールがデータフロー方程式です

Motivating Example: Reaching Definition

- Def. Definition of t
  変数tへの値の代入のこと
- Def. Use of t
  変数tの値を参照すること
- Def. Reaching Definition
  Definition *d* (of *t*) reaches a statement *u* if there is a path from d to u that does not contain any definition of *t*
- 定数伝播、コピー伝播の最適化にとって本質的な解析

Example

```
L0:  a=5
     c=1
L1: cmp c, a
    jmplt L3
L2: c=c+c
    jmp L1
L3: a=c-a
    c=0
```

- Definition a=5はどこにreachするか？
- Definition c=1はどこにreachするか？
- Definition c=c+cはどこにreachするか？
- …
- とりあえず、ピープホール最適化はしないでおこう(したあとでは話がだいぶ変わる)

データフロー解析

- 基本ブロックを取って、その入り口と出口について以下を観察する。
  - 入り口から入ってくるDefinitionの集合はどれか？
  - 基本ブロック内で生成されるDefinitionは何か？
  - 基本ブロック内で無効になるDefinitionは何か？
  - 最終的に出口で有効なDefinitionは何か？

- L2:
  c=c+c
  jmp L1

- c=c+cは、入り口において有効だったc=1を無効にしている
- 代わりに、c=c+cがcのDefinitionとして生成されている。

答え

-

	L0	L1	L2	L3
in	--	a=5 c=1 c=c+c	a=5 c=1 c=c+c	a=5 c=1 c=c+c
out	a=5 c=1	a=5 c=1 c=c+c	a=5 c=c+c	a=c-a c=0

組織的に計算する

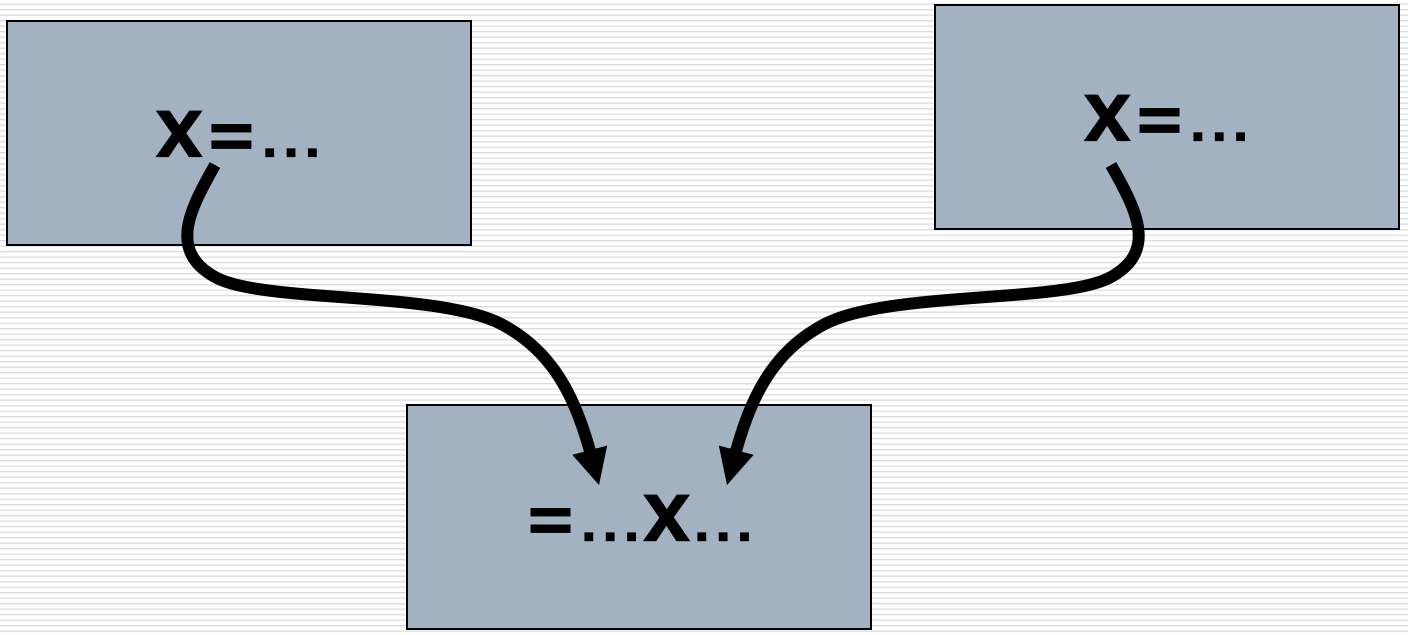
- 次のように方程式を立てる。

$$In[L] = \bigcup_{p \in L\text{のpredecessor}} Out[p]$$

$$Out[L] = gen[L] \cup (in[L]\text{-}kill[L])$$

この方程式を解く問題に帰着する

- この形式の方程式をデータフロー方程式という
  - Dominatorの計算で似たようなやつが出てきた


x=…
x=…
=…x…

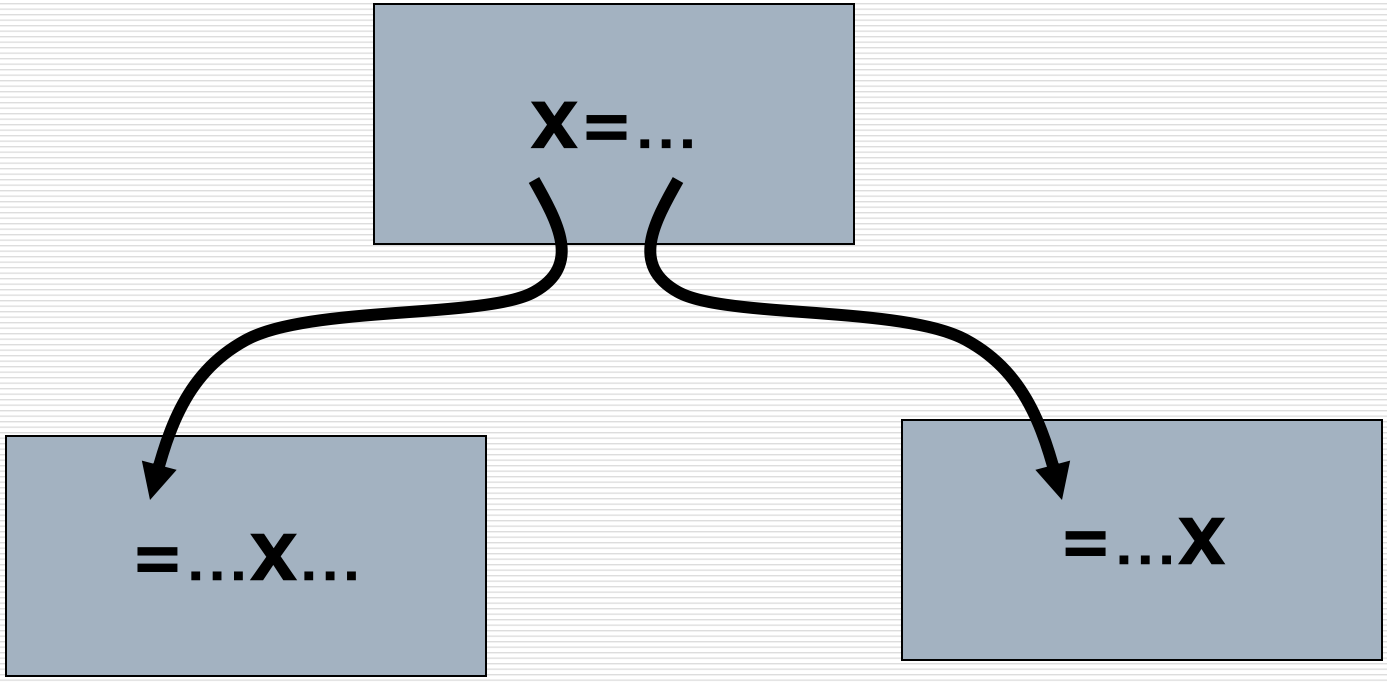
Liveness Analysis

- Def. 変数xのDefinitionがLIVE if and only if
  そのDefinitionからはじまり、useで終わり、xの他のDefinitionを含まないパスが存在する。

- Reaching definitionの計算に使った用語をそのまま使って、以下のようなデータフロー方程式が得られる。

$$In[L] = Use[L] \cup (Out[L] - Def[L])$$

$$Out[L] = \bigcup_{s \in L\text{のsuccessor}} In[S]$$


x=…
=…x…
=…x

具体的な計算

```
      a=0          // 1
L :   b=a+1        // 2
      c=c+b        // 3
      a=b*2        // 4
      cmp a N      // 5
      jmpl L       // 6
      return c     // 7
```

- ここでは、行ごとに計算をする。基本ブロックごとには行なわない。
- Defineされている変数は左辺に、Useされている変数は右辺に出現する。

この（連立）方程式を解く問題に帰着

in[1]=out[1]-{a}; out[1] = in[2]

in[2]={a}+(out[2]-{b});

out[2]=in[3]+in[6]

in[3]={c,b}+(out[3]-{c}); out[3]=in[4]

in[4]={b}+(out[4]-{a}); out[4]=in[5]

in[5]={a}+{out[5]); out[5]=in[6]

in[6]=out[6]; out[6]=in[7]+in[2]

in[7]={c}+out[7]; out[7]=Φ

データフロー方程式

- データフロー情報を一つ定める。
- 基本ブロック内で情報が「生成される」アクション(gen)と「無効化される」アクション(kill)を定める。
- 基本ブロック内に入るときの情報をinとし、そこから出るときの情報をoutとする。
- Def. Dataflow Equation
  各基本ブロック（またはその拡張概念）n に対して gen[n], kill[n], in[n], out[n] が満たす等式のことをデータフロー方程式という。

具体例

- Reaching defintion
  gen[n] – definitions at n
  kill[n] -- definitions invalidated at n
  (variables reassigned at n)

- Liveness analysis
  gen[n] – uses at n
  kill[n] – definitions at n

Equations

- Reaching Definitions

$$In[L] = \bigcup_{p \in L\text{のpredecessor}} Out[p]$$

$$Out[L] = gen[L] \cup (in[L]\text{-}kill[L])$$

- Liveness Analysis

$$In[L] = Gen[L] \cup (Out[L] - Kill[L])$$

$$Out[L] = \bigcup_{s \in L\text{のsuccessor}} In[S]$$

様々なデータフロー方程式

- Available expressions
- Def. Available expressions
  (x op y)がポイントpでavailableであるとは、pに至るすべてのパスにおいて、(x op y)が計算されていて、かつその計算後にx, yに代入がないことを言う。
- gen[n] = (x op y)の計算
  kill[n] = n中のdefinitionsを含む式
- これが、common subexpression eliminationの拡張版につながります

- 方程式

$$in[n] = \bigcap_{p \in n\text{のpredecessor}} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

解けるか？

和を取るか、積を取るか

- 多くの方程式は以下の形をしている。

$$in[n] = \bigcap_{p \in n\text{のpredecessor}} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Predecessorについてやるかsuccessorについてやるか

	和型	積型
Predecessor型	in, outを∅からはじめて、終わりから計算して繰り返し	in, outをTからはじめて、終わりから計算して繰り返し
Successor型	in, outを∅からはじめて、はじめから計算して繰り返し	in, outをTからはじめて、はじめから計算して繰り返し

定理：前スライドの繰り返しは単調に増加（和型）、または単調に減少（積型）する。

定理：コンパイラ最適化では、in,outの各集合は有限集合であるので、必ず収束する。

定理：Predecessor型の場合は、predecessorに対する計算を先にやるほうが効率がよい。（dual in successor型）

これらの定理は、以下の計算方法の裏付けになる

forall n. in[n] = out[n] = ALL;
while (変化が観測される) {
    forall n. {
        in[n] = $\bigcap_{p \in Predecessor}$ out[p];
        out[n] = gen[n] $\cup$ (in[n] – kill[n])
    }
}

具体的に解いてみる

in[1] = in[2] = … = in[7] = ∅
out[1] = out[2] = … = out[7] = ∅

1st round:

in[1] = out[1]-{a} = ∅
in[2] = {a} + (out[2]-{b}) = {a}
in[3] = {c,b}+(out[3]-{c}) = {c,b}
in[4] = {b} + (out[4]-{a}) = {b}
in[5] = {a} + out[5] = {a}
in[6] = out[6] = ∅
in[7] = {c}+out[7] = {c}
out[1]=in[2] = {a}
out[2]=in[3]+in[6] = {c,b}
out[3]=in[4] = {b}
out[4]=in[5] = {a}
out[5]=in[6] = ∅
out[6]=in[7]+in[2]={c,a}
out[7]=∅

2nd round:

in[1] = out[1]-{a} = ∅
in[2] = {a} + (out[2]-{b})= {c,a}
in[3] = {c,b}+(out[3]-{c}) = {c,b}
in[4] = {b} + (out[4]-{a}) = {b}
in[5] = {a} + out[5] = {a}
in[6] = out[6] = {c,a}
in[7] = {c}+out[7] = {c}
out[1]=in[2] = {c,a}
out[2]=in[3]+in[6] = {c,b,a}
out[3]=in[4] = {b}
out[4]=in[5] = {a}
out[5]=in[6] = {c,a}
out[6]=in[7]+in[2]={c,a}
Out[7]=∅

3rd round:

in[1] = out[1]-{a} = {c}
in[2] = {a} + (out[2]-{b})= {c,a}
in[3] = {c,b}+(out[3]-{c}) = {c,b}
in[4] = {b} + (out[4]-{a}) = {b}
in[5] = {a} + out[5] = {c,a}
in[6] = out[6] = {c,a}
in[7] = {c}+out[7] = {c}
out[1]=in[2] = {c,a}
out[2]=in[3]+in[6] = {c,b,a}
out[3]=in[4] = {b}
out[4]=in[5] = {c,a}
out[5]=in[6] = {c,a}
out[6]=in[7]+in[2]={c,a}
out[7]=∅

4th round:

in[1] = out[1]-{a} = {c}
in[2] = {a} + (out[2]-{b})= {c,a}
in[3] = {c,b}+(out[3]-{c}) = {c,b}
in[4] = {b} + (out[4]-{a}) = {c,b}
in[5] = {a} + out[5] = {c,a}
in[6] = out[6] = {c,a}
in[7] = {c}+out[7] = {c}
out[1]=in[2] = {c,a}
out[2]=in[3]+in[6] = {c,b,a}
out[3]=in[4] = {c,b}
out[4]=in[5] = {c,a}
out[5]=in[6] = {c,a}
out[6]=in[7]+in[2]={c,a}
out[7]=∅

5th round: same as 4th round

データフロー方程式の応用

- 方程式を解いて得られた情報を用いて行なう最適化のいくつか。

1. Dead Code Elimination

   もし、代入 a=…において、aがliveでなければ、この代入は不要 （liveness analysis）

データフロー方程式の応用(続き)

2. Constant Propagation
   t=c (c, constant)がreachしている
   s=…t…中のtはcに置き換えることができる。

3. Copy Propagation
   t=v がreachしている
   s=…t…中のtはvに置き換えることができる。

4. Common Subexpression Elimination

t=x op y の出現pointにおいて(x op y)がavailableであれば、この計算は削除できる。

5. Global Value Numbering

各変数の参照 & quantityの割り当てにおいてReaching Definitionに対応するquantityが1つのみであれば、そのDefのquantityをそのまま使うことができる

Otherwise　新たにquantityを割り当てる

（1-4は、global value numberingを適用した最適化になっていることに注意する）

実際のコンパイラでは

- Pythonでは、見ないですね…
- GCCでは、普通に観察されます
  - コードを読むには、それなりの準備が要求されます
  - Minor versionが異なるだけでコードの入れ換えが発生するので気が抜けない
  - 次回は、どこにあるかを探すことだけは示しましょう