プログラミング言語処理系論 (8)
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）

データフロー解析をもとにした大域最適化

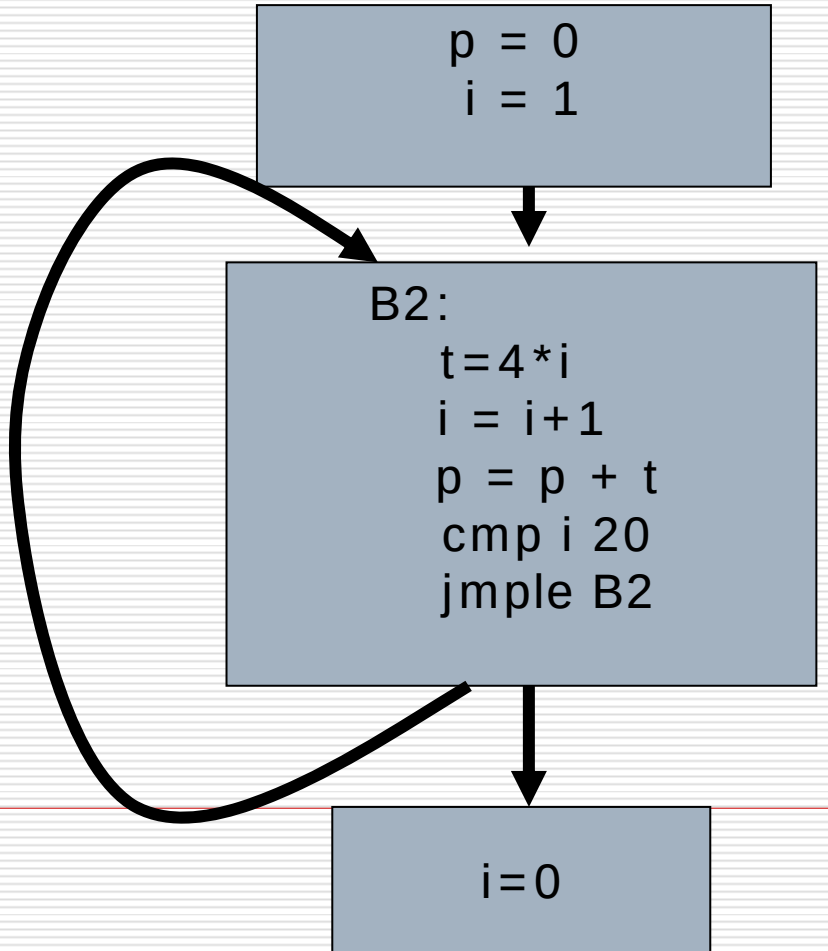
- 以下、基本ブロックの集合から作成した次のグラフを考える。
  - ノード
    基本ブロック
  - エッジ
    基本ブロックx →基本ブロック y

    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. Flowgraph 上の通り作ったグラフ

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

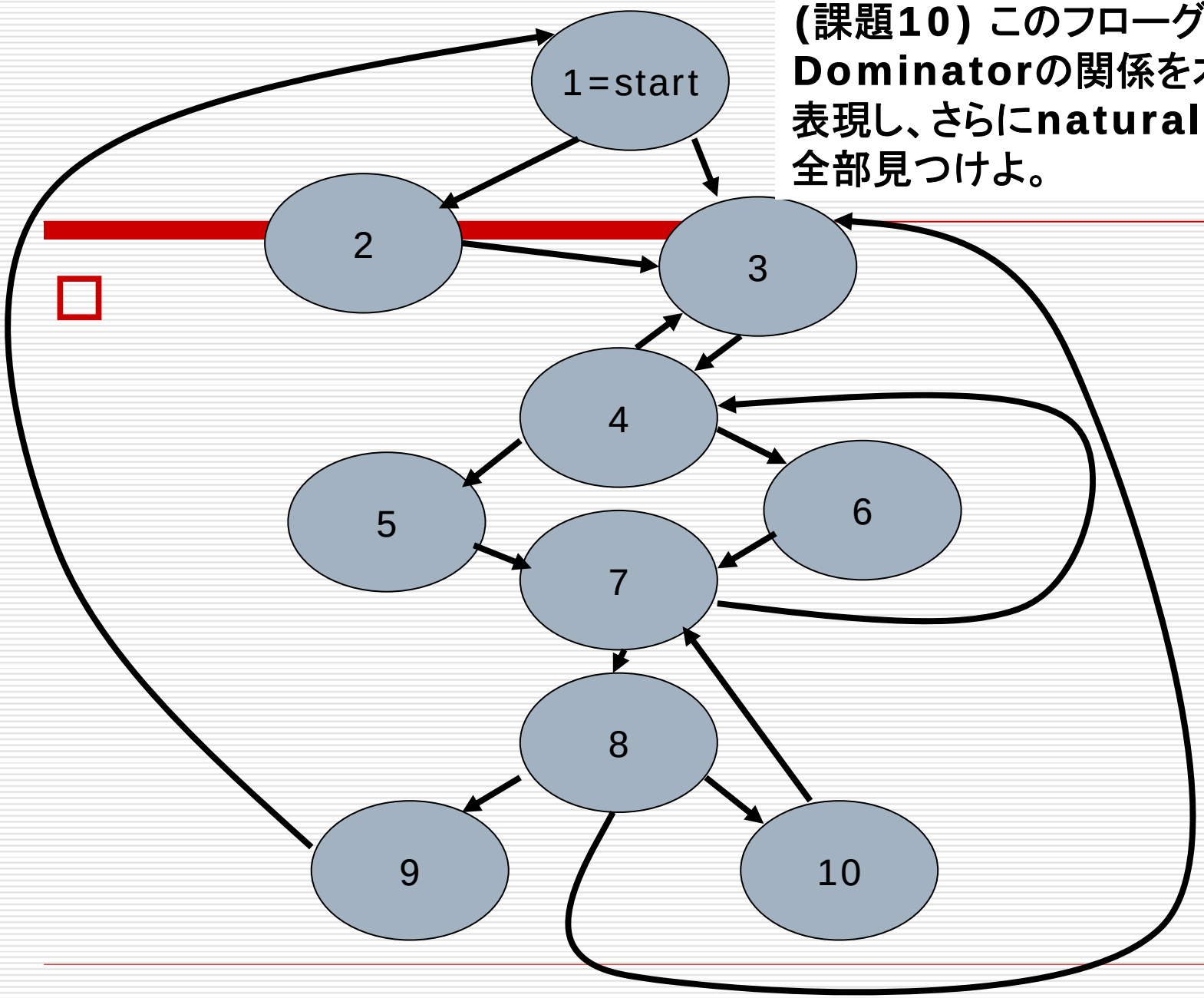
- Def.　start block
  開始ブロックをひとつ指定しておく。
- Def. Dominate
  ブロックdがブロックnをdominateするとは、スタートブロックを始点とし、nに到達するすべてのパスがdを含むことをいう。このとき、dはnのdominatorであるともいう。

- Dominatorの性質1. Dominateするという関係はtreeになる (no cyclicity)。
- Def. immediate dominator dominatorであって、他のdominatorからdominateされないもの

- 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. natural loop
  back edge b: n→dが与えられたとき、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10) このフローグラフで Dominatorの関係を木構造で表現し、さらにnatural loopを全部見つけよ。



今回の予定

- 大域的データフロー解析をもとにした最適化
  - 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

Motivating Example: Reaching Definition

- Def. Definition of t
  t への値の代入のこと
- Def. Use of t
  tの値を使うこと
- 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*

Reaching Definition

- なぜ大切か？

- 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

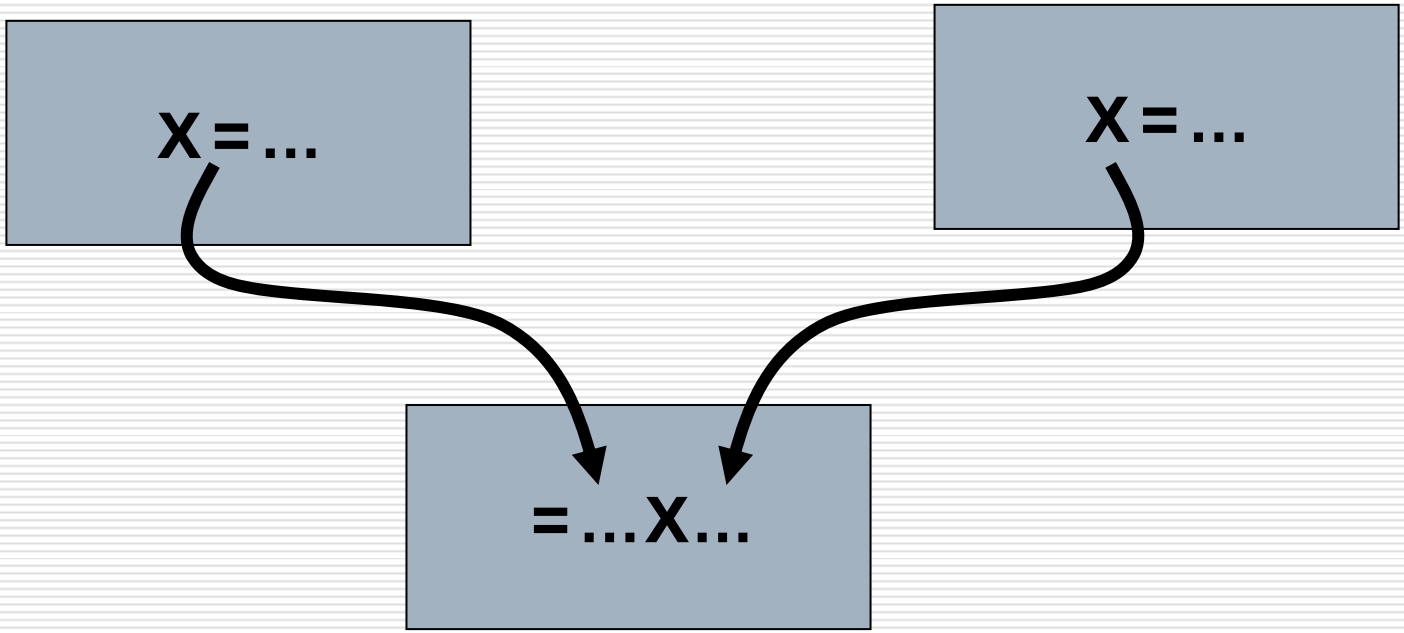
アドホックに求めてはいられない

- 次のように方程式を立てる。

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

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

この方程式を解く問題に帰着する


x = …
x = …
= … x …

Liveness Analysis

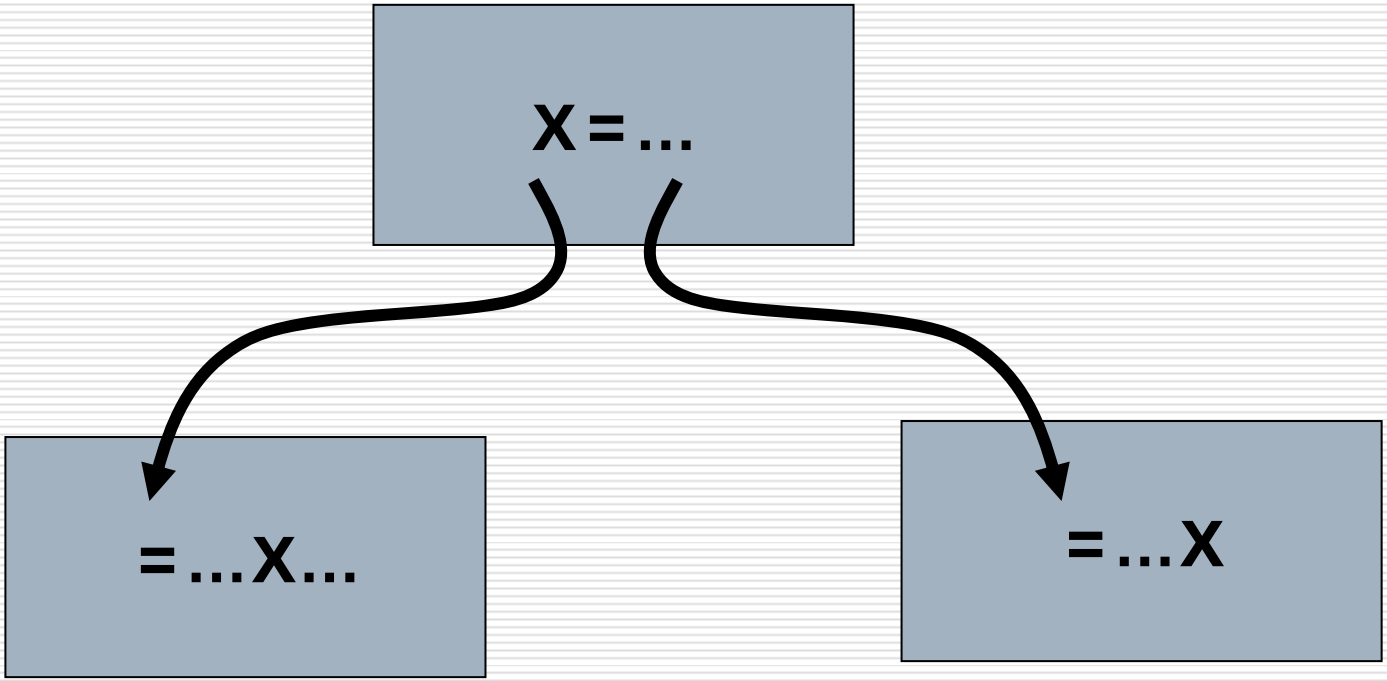
- Def. 変数xのDefinitionがLIVE if and only if
  そのDefinitionからはじまり、useで終わり、xの他のDefinitionを含まないパスが存在する。

- なぜ重要か？
  - すべてのプログラム解析の基本

- 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。

(課題11)この(連立)方程式を解け(解き方は後述する)

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

- 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ϕ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ϕ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。

（課題12）各繰り返しにおいてgenとkillが一定であると仮定した上で上の定理2つを証明せよ。この結果として、「出てきた解は最小（または最大）である」ことを併せて証明せよ。

- 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）

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。

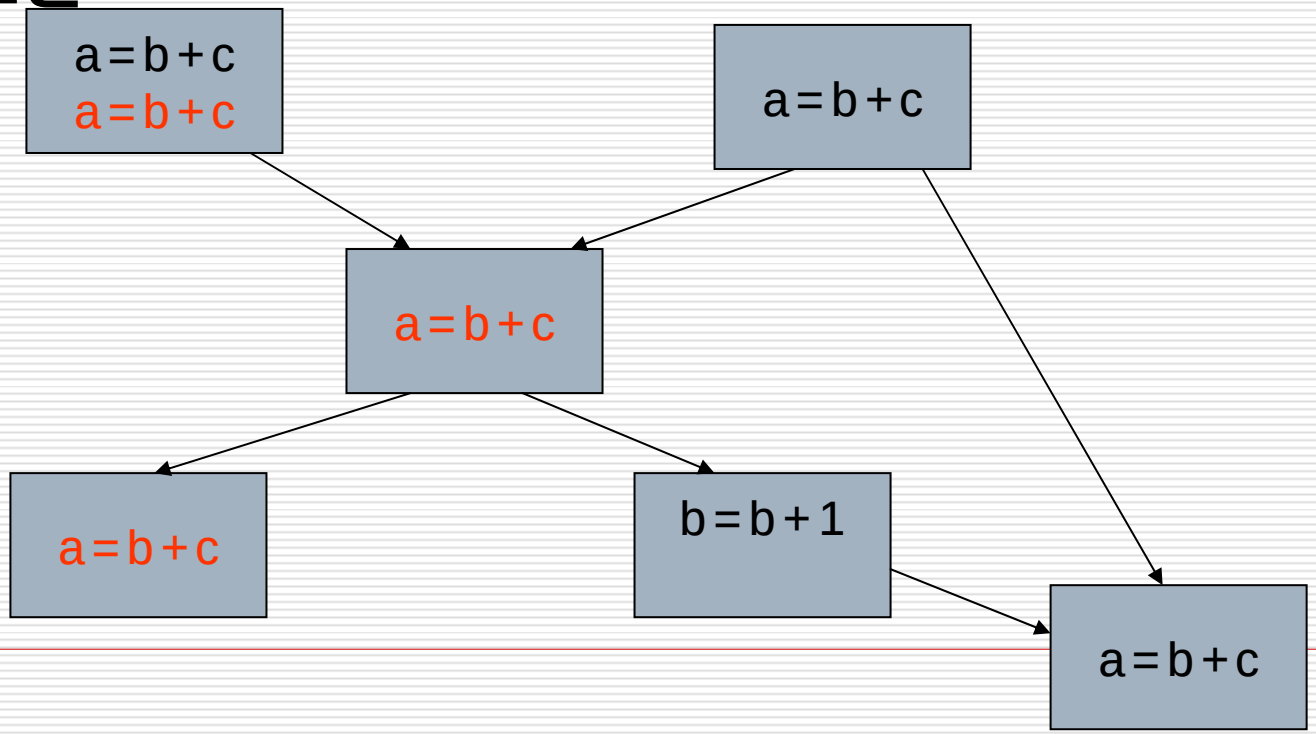
3. Copy Propagation
   t=v がreachしている
   s=…t…中のtはvに置き換えることができる。

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。

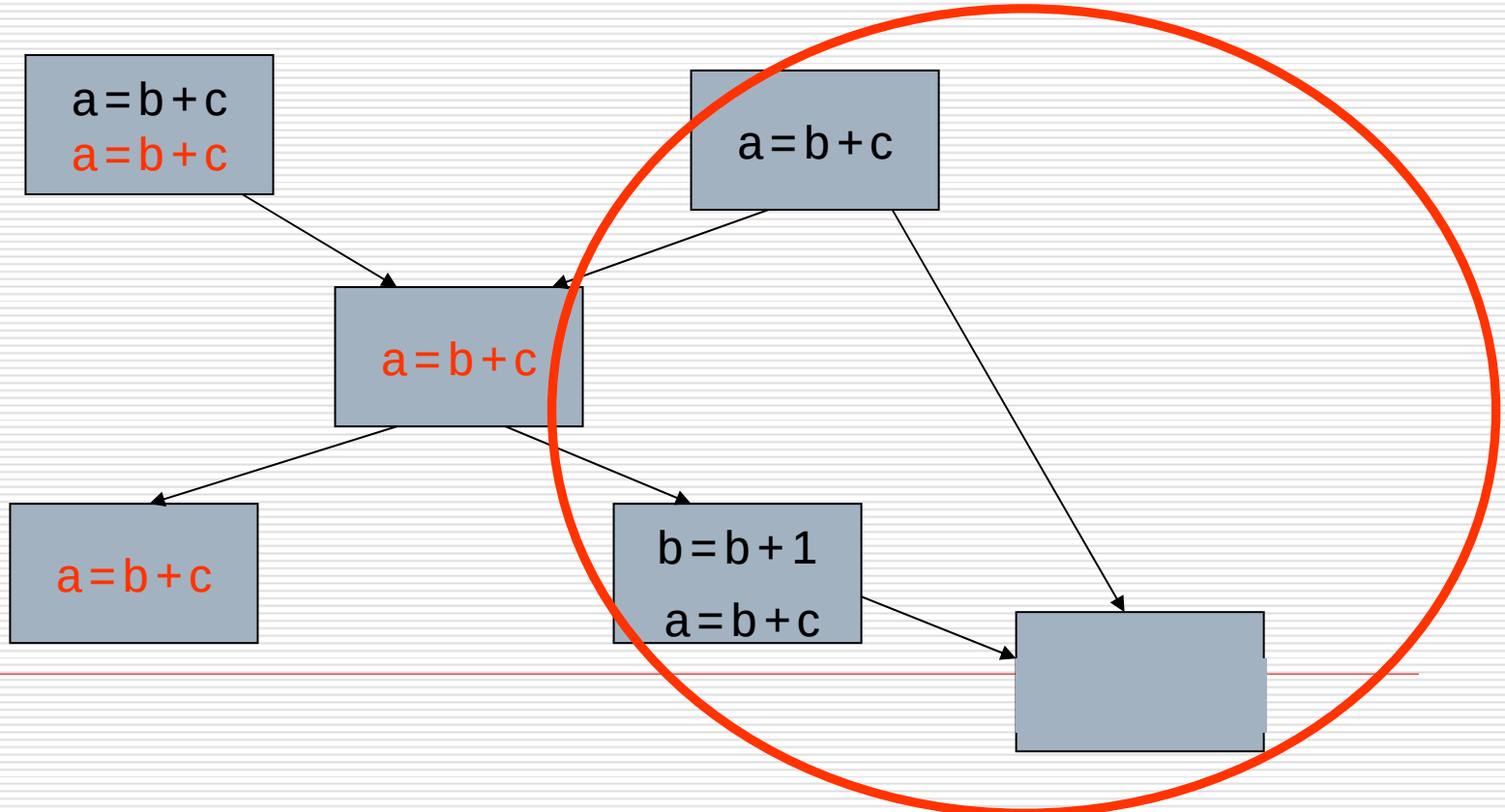
(Partial) Redundancy Elimination

- Def. Redundancy Elimination
  計算のパスにおいて、重複する計算を取り除くこと



- Def. Partially Redundant
  式がプログラムポイントでpartially redundantであるとは、そこにいたるパスの中でredundant expressionな式であるものが存在すること(すべてのパスである必要はない)
- たとえば、ループ不変式はpartially redundantである

Partial Redundancy



- Def. Lazy Code Motion
  Partially Redundant Eliminationのためにコードを移動すること

- Knoop, et.al., Lazy Code Motion, Programming Language Design and Implementation ’92, 1992.

- Def. Anticipable Expressions
  式(x op y)がanticipableとは、その計算が後続のパスのどこかでなされ、しかもそのパス中にx, yの定義が入らないこと(計算を先に実行してもよい)。
- $ANTout[n] = \bigcap_{s \in n\text{のsuccessor}} ANTin[s]$

  $ANTin[n] = Defed[n] \cup (ANTout[n] - kill[n])$

Lazy Code Motion

- Def. Earliest(i,j)　(i,j – ブロック)
  (1)ブロックjの先頭まで移動でき、
  (2)ブロックiの終端ではavailableでなく(よって、挿入してもredundantにならない)、
  (3)ブロックiの先頭には移動できない
  式の全体　→　エッジ(i,j)に挿入できる

- Earliest(i,j) = ANTin(j) – AVAILout(i)
  ∩ (Kill(i) – ANTout(i))

- Def. e ∈LaterIn(k)
  kに到達するパスすべてについて、そのどこかでeがEarliestであり、そこからkまで、eを評価していない。(eの計算はkから前に移すことができる)
- Def. Later(i,j)
  Earliestであるか、またはiから後ろに移動でき、かつiの先頭に移動するとiではanticipableでない式の全体
- Later(i,j) = Earliest(i,j)∪(LaterIn(i)–Defed(i))
- $LaterIn(j) = \bigcap_{i \in predecessor(j)} Later(i,j)$

- INSERT(i,j) = Later(i,j)– LaterIn(j)
- Delete(k) = Defed(k) – LaterIn(k)

- (課題13) KnoopらのLazy Code Motion の論文を読んでサマリーを書け。

ここで反省

- データフロー方程式で集める情報の基本はDefinitionとUseに関するもの。
- Def Def/Use Chain
- 似たような事を何回もやっている。
- Def/Use Chainを効率的に表現できる中間表現があれば、この種の最適化も効率的にできるはずである。

次回の予定

- SSA (Static Single Assignment)とそれを用いた最適化について