プログラミング言語処理系論 (7)
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

- 最適化の解説 (I)
  - 最適化の役割
  - 最適化の分類
  - 簡単な最適化
  - プログラム解析の初歩
- 次回以降の予定
  - データフロー解析をもとにした最適化
  - SSAをベースにした最適化

最適化の役割

- What is an Optimization?

- Def. Optimization
  同じ出力を得られることを保証しながら、プログラムをよりよいものに変換すること

- What is「同じ出力」?
- What is「よりよい」?

- Def. Same Output
  プログラムAとBが同じ出力を持つとは、すべての入力について「結果」が同一であることをいう。
  「結果」をビット列で考えるか、より抽象的な数学的なドメインで考えるかによって話は違う。
  例:ビット列で考えると　Σ1/i !=　Σ1/(n-i)
  　　数学的などメインで考えると
  　　Σ1/i =　Σ1/(n-i)

- 一般的なものとしては、「同じ出力を持つ」ことを以下で代替する

  「各変数それぞれについて、代入される値とその順番が変更されないこと」

  （もちろんこれを超える定義を使うこともある。その時はいちいち明示して使う）

同じ出力　＝　同じ計算順序

- 以下のものを考える

```
a = 1
b = 2
c = 3
a = b+1
b = a+1
c = c+1
```

- 以下は左と「同じ」

```
b=2
a=1
c=3
c=c+1
a=b+1
b=a+1
```

同じ計算順序（進んだ話題）

- 配列の計算で、この概念は重要

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++)
      for(k=0;k<N;k++)
         c[i,j] += a[i,k]*b[k,j];
```

- 各c[i,j]について、値の計算と代入の順序は変更されていない

```
for (i=0;i<N;i++)
   for (k=0;k<N;k++)
      for(j=0;j<N;j++)
         c[i,j] += a[i,k]*b[k,j];
```

- （課題8）ループi,j,kの順序をどのようにしてもよいことを証明せよ。それならば、どのような順序で実行するのがよいのか。現在支配的なメモリアーキテクチャを前提として考えよ。
- （課題8’）ループアンローリングは計算の順序どころか、変数へのアクセスの順番をまったく変更しないことを示せ。それにも係わらず性能が上がることがよくあるのはなぜか、現在支配的なCPUアーキテクチャを前提として考えよ（ループのオーバヘッドの削減は理由にならない）。

「よりよい」

- Def. よりよい
  ある基準を定めた上でよりよい
  基準の例:
    - スピード
    - コードサイズ
    - メモリ使用量
    - 消費電力
    - …

- 前のスライドにあげた基準はどれも現実的なものである
  - 通常のプロセッサ
  - 組み込みプロセッサ
  - 大量に電力を消費するシステム
  - …

最適化の分類

- 極端な例を2つ。
- x = 1 + 2
  - → x = 3
- for(r=0,i=1; i<10; i++)
  r += i;
  - → r = 45; i=10;

  (not a joke. Mathematicaなどの数式処理システムではもはや常識)

最適化の分類

1. ローカル最適化
   1. 基本ブロック内外で、一定のパターンに当てはまるものを変換(ピープホール最適化)
   2. 基本ブロック内で、変数の挙動を解析して変換
2. データフロー解析をもとにした大域最適化
3. データ依存解析をもとにしたループ最適化
4. 手続き間解析をもとにした手続き間最適化

最適化の分類(その他)

- CPUアーキテクチャを意識した最適化
  - レジスタアロケーション
  - 命令スケジューリング

（ここでは時間の関係で扱わない）

ターゲット言語

```
stmt: block*;
block: label insn*  jump?;
insn: var ‘=‘ oper
    | var ‘=‘ oper op oper
    |cmp oper oper;
oper: var | literal;
op: ‘+’ |‘-’|...;
jump: jmp label;
jmp: ‘jump’ |‘jmpz’ |‘jmpnz’|...;
```

- 以下のアセンブリ言語ライクなものと思えばよい(関数コールはとりあえず無視する)。

```
_main:
        s = s + b
        s = s – 24
        s = s & (-16)
        a = 0
        a = a + 15
        …
```

- Def. Basic Block (基本ブロック)
  連続した命令(insn)列であり、
  1. 制御フローが必ず先頭からはじまり
  2. 最後で離れる。
  3. その間(最終行以外)に終了したり、他にジャンプすることがない。

- 今後、基本ブロックはすべてのプログラム解析の基本になる。

ローカル最適化

- まずは基本ブロック内での最適化を考える。

- Value numbering
  (共通部分式を全部探し出す)

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

i=i+3

1	id	I	
2	Num	3	
3	+	1	2
3	id	I	

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

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

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。

- （課題9）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補完し、計算の過程も記すこと。

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

ローカル最適化

- Peephole Optimization
  - ごく短い命令列を走査していき、一定のパターンをみつけて、それをよりよいパターンに置換していく
  - Redundant instruction elimination
  - Flow-of control optimization
  - Algebraic simplification
  - Machine idiom recognition

Redundant instruction elimination

- Redundant load/store

  r0=a
  a=r0
  (下の命令は不要)

- unreachable code elimination

```
        cmp debug 1
        jmpz L1
        jmp L2
L1:  …
        …
L2:
```

(下のように書き換えられる)

```
        cmp debug 1
        jmpnz L2
        …
```

Flow of control optimization

- (厳密に言えば基本ブロック内ではないのだが…)

```
        jmp L1
        …
   L1:  jmp L2
```

→

```
        jmp L2
        …
   L1:  jmp L2
```

(ラベルL1は最終的になくなるだろう)

Flow of control optimization

```
    jmp L1
    ...
L1: jmpnz L2
L3:
→
    jmpnz L2
    jmp L3
    ...
```

Algebraic simplification

- Strength reduction

  x * 2 → x <<1

  x ^ 2 → x * x

  x / 2 → x * 0.5

- 代数的に等価な表現への置き換え

  x + 0 → x

  x * 1 → x

  x / x → 1

  (浮動小数点ではやらないのが無難)

Machine idioms

- ハードウェア命令があれば

Peephole optimizationの効果

- 結構ばかにできない（丁寧な対応が求められる）
- パターンの発見と適用がすべて（アドホックな理論しかない）

- パズルを解くような感じ

データフロー解析をもとにした大域最適化

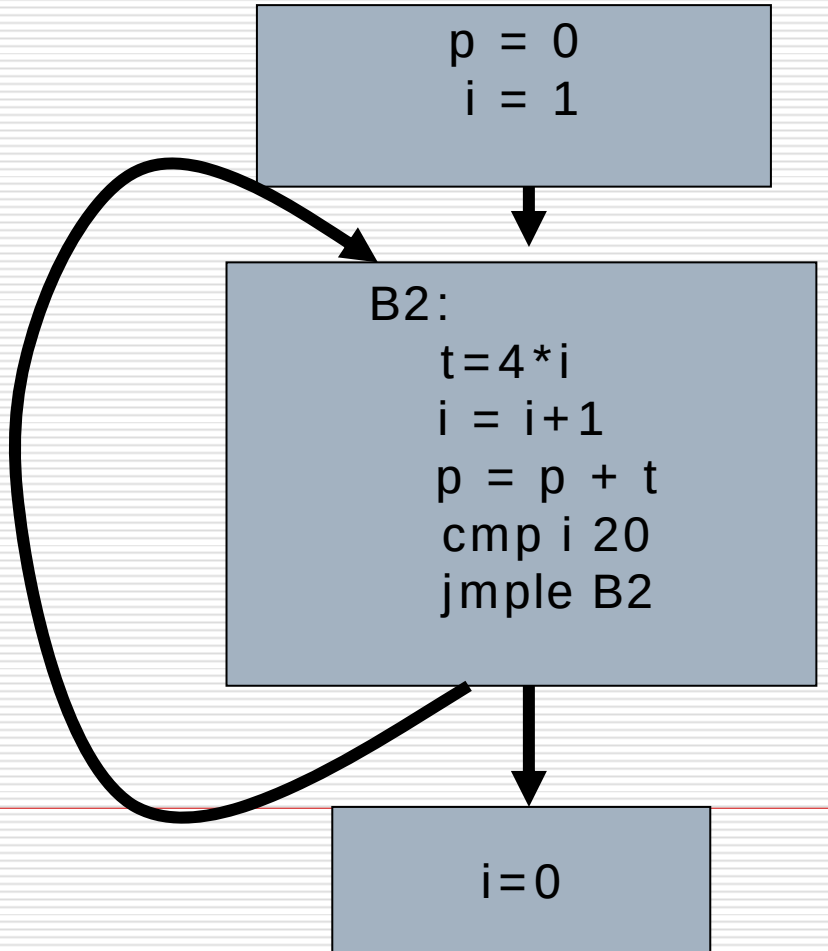
- 以下、基本ブロックの集合から作成した次のグラフを考える。
  - ノード
    基本ブロック
  - エッジ
    基本ブロックx →基本ブロック y

    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。