システムソフトウェア特論'17 # 12 コード解析と最適化

久野　靖 *

2017.11.12

1　最適化の原理と分類

最適化 (optimization) とは、コードにさまざまな変形や工夫を施すことで、性能を向上させる作業です。すなわち、その名前とは裏腹に、コードの実行性能を「改良する」わけです。名前通りであれば「最適」にすることになりますが、実際に「最適」にすることは極めて困難ですし、また CPU やデータの状況により何が「最適」となるかは変化してしまいます。

ここで最適化に関して重要な原則を挙げておきます。

> **原則:** 最適化のための変形により、コードが正しく動作しなくなってはいけない。

たとえば次のようなコードがあったとします (本来は中間コードでやりますが、見やすさのため以下ではソースコードレベルでの説明も多く出て来ます)。

```
while(条件) {
  x = ... /* 複雑な計算 */
  ...
}
```

ここで、x の値はループの周回があっても一定であるものとします。そうすると、ループは何万回も実行されるかもしれないので、x の計算をループの外に出すことで実行時間を削減したくなります。

```
x = ... /* 複雑な計算 */
while(条件) {
  ...
}
```

このような変形をしても大丈夫でしょうか? 一見良さそうですが、実はまずい可能性があります。というのは、x の計算は元のコードでは while の「条件」が満たされた時だけ実行されます。もしかしたらその計算は、「条件」が満たされていないとエラーになるかも知れません。そうなると、元は正しく動いていたプログラムがエラーを出すようになってしまいます。

ですから、最適化に際して行なうべき正しい変形は次のようになります。

```
if(条件) {
  x = ... /* 複雑な計算 */
  while(条件) {
    ...
  }
}
```

*電気通信大学 情報理工学研究科

面倒だと思うかも知れませんが、「コードが正しく動作しなくなってはいけない」を守るというのはこういうことなのです。

では次に、最適化によって具体的にどのようにして実行速度を高めることが可能なのでしょうか。基本的な原理としては次のものが挙げられます。

(a) 実行しなくてもよい命令列を発見し、取り除く。

(b) 複数回実行されるがその間で結果が変わらない命令列を発見し、1 回の実行ですませるように変更する。

(c) 複数回実行される命令列を、より少ない実行回数ですませる。

(d) ある命令列を、それと同じ結果をもたらすより高速な命令列に取り替える。

(a) と (b) は、言い換えれば静的/動的に重複した計算を発見して削除することを意味します。先の while 文の例はこのうちの (b) に対応します。また、ループの実行回数が固定回でかつ小さいなら、ループ本体のコードをその回数だけコピー (展開) してしまえば、ループを制御する命令は不要になります。または、回数が多い場合でも、本体を N 回展開することで、周回数を $\frac{1}{N}$ 倍にできますから、ループ制御命令の実行回数を減らせます。これは (c) に相当します。(d) は、たとえば乗算命令よりは加算命令の方が高速なのが普通ですから、2 倍するときは乗算の代わりに 2 回足す命令を使う、などの場合が相当します。

ここまででも分かるように、最適化には極めて多様な手法があり、それだけで本が何冊も書けるほどです。ここでは、ごく基本的な概念に絞って、できるだけ具体例を挙げる形で説明します。それでも例題として実行できる部分はごく一部になりますが、ご容赦ください。また、ループ最適化は付録としますので、興味があれば読んでください。

2 中間コード生成と制御フロー解析

2.1 中間コードと 4 つ組

標準的なコンパイラの構成においては、抽象構文木から**中間コード** (IR — intermediate representation) を生成し、最適化を行なった後、目的コードへの変換を行なうことは既に学びました。そのようにする目的は、抽象度の高い抽象構文木や、ターゲット CPU に強く依存する目的コードでは、最適化が行ないにくいからです。

中間コードにもさまざなな形式のものがありますが、最適化に適した形式としては**レジスタ転送言語** (RTL — register transfer language) や **4 つ組** (quadleple) と呼ばれるものがよく使われます。

RTL は GCC(Gnu C Compiler) で採用されていることからよく知られています。RTL はその名前通り、多数あるレジスタの間で値を転送しながら計算していくというモデルであり、本質的には 4 つ組と似ていますが、たとえば演算に伴い条件コードビットが辺かすることなど、CPU の様々な機能を併せて扱うことができます。

一方、4 つ組は典型的には「命令, 代入先, 被演算子 1, 被演算子 2」の 4 つの情報が 1 つの命令を構成することからこの名前があるもので、たとえば次のように見慣れた代入文の書き方で記述することができます。

```
  t2 = t1 + 1
  t3 = t4[t2]
  t5 = 0
  if t3 > t5 then L1
  ...
L1:
  ...
```

ただし、ここに現れる名前はいずれも**テンポラリ** (temporary) ないし作業レジスタであり、実際にはコード生成時にメモリ上の位置またはいずれかの CPU レジスタに割り付けられることで動作します。また、個々の命令はアセンブリ言語レベルのものであり、複雑な計算式は存在しません。条件分岐命令やラベルがあることからも、アセンブリ言語に近い水準であると言えます。以下の例題では、主に 4 つ組を説明に使用します (このほか、ソースコードで説明する部分もあります)。

具体例がないと分かりにくいと思うので、ここではいつもの言語で書いた (文法は前回と同じなので省略) 図 1 のプログラムの中間コードを図 2 に掲載しておきます。

```
int sub(int n; int a[]) {
  int remain; remain = 1;
  while(remain) {
    remain = 0; int i; i = 1;
    while(i < n) {
      if(a[i]>a[i-1]) { int z; z=a[i-1]; a[i-1]=a[i]; a[i]=z; remain=1; }
      i = i + 1; } } }
```

図 1: 小さい言語で書いたバブルソート

```
L1000: // B0
  t2 = %rdi
  t3 = %rsi
  t7 = 1
  t4 = t7
L0: // B1
  ifz t4 then L1
L1001: // B2
  t8 = 0
  t4 = t8
  t9 = 1
  t5 = t9
L2: // B3
  if t5 < t2 then L4
L1002: // B4
  t10 = 0
  jmp L5
L4: // B5
  t10 = 1
L5: // B6
  ifz t10 then L3
L1004: // B7
  t11 = t3[t5]
  t12 = 1
  t13 = t5 - t12
  t14 = t3[t13]
  if t11 > t14 then L7
L1005: // B8
  t15 = 0
  jmp L8
L7: // B9
  t15 = 1
L8: // B10
  ifnz t15 then L6
L1007: // B11
  t16 = 1
  t17 = t5 - t16
  t18 = t3[t17]
  t6 = t18
  t19 = 1
  t20 = t5 - t19
  t21 = t3[t5]
  t3[t20] = t21
  t3[t5] = t6
  t22 = 1
  t4 = t22
L6: // B12
  t23 = 1
  t24 = t5 + t23
  t5 = t24
  jmp L2
L3: // B13
  jmp L0
L1: // B14
```

図 2: 小さな言語で書いたバブルソートの中間コード

2.2 基本ブロックとフローグラフ

最適化の観点からは、IR 命令の列は**基本ブロック** (basic block) の集まりとして取り扱います。基本ブロックとは、途中からの飛び出しや途中への飛び込みが無いような IR 命令の列、言い替えれば「先頭から 1 直線に実行される列」を言います。

IR 命令の中で、飛び込みはラベルの箇所にしか起こりません。逆に言えば、ラベルにはよそから飛んで来るので、ラベルがあるとそこから先が 1 つのブロックになります。そして、分岐命令か条件分岐命令があると、そこからは飛び出すことになるので、これらの命令があるとブロックが終わります (このほかに、次のラベルが現れた場合にもブロックが終わります)。

整理すると、ブロックの先頭は手続きの先頭命令、ラベル、分岐命令の直後の命令のいずれかであり、ブロックの最後の命令はラベルの直前の命令と分岐命令のいずれかです。以下では扱いの統一のため、ラベルで始まらないブロックの先頭にもラベルを付加し、ブロックを同定するのに先頭のラベルを用います。図 2 のコードでは、ラベルの後にブロック番号を記載しました。

基本ブロックの重要な性質は、そこに含まれる命令列は必ず最初から最後まで順に実行される、ということです。したがって、例えばブロック中で同じ式を複数回計算していて、なおかつその間にその式の値を変更する可能性を持つ命令がないなら、最初に計算した値を保存しておいて繰り返し利用してもよいことになります。

このように、各ブロック内に範囲を限って最適化を行うことを**局所最適化** (local optimization) と呼びます。これに対し、1 つの手続き内でブロックをまたがって最適化を行うことを**広域最適化** (global optimization) と呼びます。さらに、他の手続きを呼び出した先にまたがって行なう最適化である**手続き間最適化** (interprocedural optimization) までありますが、それをやるとその手続きは単独で呼べなくなるので、必ずしも普及していません (以下でも取り上げません)。以下ではまず、局所最適化の例を挙げ、そのあと広域最適化に必要なことがらを取り上げていきます。

あるブロックを実行した後実行が進む先は、そのブロックの末尾が無条件分岐命令ならばその飛び先、条件分岐ならば飛び先と後続ブロックの双方、それ以外では後続ブロックのみとなります。1 つの手続きについて、ブロックを節、行き先関係を矢線で表した有向グラフをその手続きの**フローグラフ** (flow graph) と呼びます。図 3 に、先の中間コードのフローグラフを示します。広域最適化を行う場合にはフローグラフを構成し、その上で各種解析と最適化を行います。

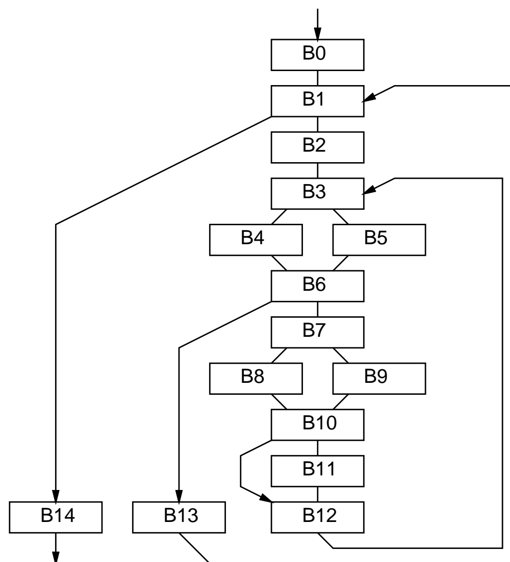

図 3: バブルソートのコードのフローグラフ

2.3 制御フロー解析

フローグラフを作成した後まず**制御フロー解析** (control flow analysis) を行うことが通例ですが、その主な目的はプログラム中のループを同定することです。これは、後で述べるように、ループ最適化によって多くの性能向上が見込めるためです。

今日の言語では、ループはソースコードの繰返し構文に対応しているので、フローグラフからループを再発見する代りに繰返し構文の情報を保持しておく方法もありますが、ここでは一般のフローグラフからのループを同定する方法を説明しておきます。まず、いくつかの用語を定義します。

定義 フローグラフ G に含まれる節 d が節 n を**支配する** (dominate) とは、グラフの出発点から n に至る全ての経路が d を通ることを言う。

定義 フローグラフ G に含まれる辺 $b \to h$ が**帰辺** (back edge) であるとは、h が b を支配する節である場合を言う。

定義 フローグラフ G に含まれる帰辺 $b \to h$ に関する **自然ループ** (natural loop) とは、n から b へ h を通らずに行ける経路があるような節 n(b 自身を含む) と h を合せたものである。ここで h をループの**ヘッダ** (header) と呼ぶ。

図 3 を例に考えると、B1 は B2〜B14 をすべて支配し、B3 は B4〜B13 を支配します。支配とは「そこを通らなければ行けない」ということですね。この 2 つのブロックがループヘッダです。そして、B13 → B1 と B12 → B3 がそれぞれ帰辺になります (B1、B3 へ行くこれ以外の辺は支配された節から出ていません)。そして、B1〜B14、B3〜B13 がそれぞれ自然ループとなります。

ところで「自然な」ループがあるからには「自然でない」ループも存在します。具体的には、自然でないループとは入口が 2 つ以上あるようなループですが、今日の制御構文を使って作られたプログラムではそのようなものはできないので、自然ループのみを最適化の対象とするのが普通です。

自然ループは入口が 1 つ、出口が 1 つで、2 つの自然ループで共有される節がある場合には、片方が他方に完全に含まれるという性質があります。時として、複数の帰辺が 1 つの行き先を持つことがありますが、そうすると 2 つの自然ループでヘッダを共有してしまい、最適化のとき不便なので、適宜空のブロックを追加して、自然ループとヘッダが 1 対 1 で対応するようにします。さらに、ヘッダの直前に空のブロックを追加し、ループ外から入って来るときにはこを通過するようにします。これを**プリヘッダ**と呼び、ループ最適化のときループ内からコードを移して来るのに使います。

3 中間コード生成とブロックの構築

3.1 中間コードのデータ構造

それでは実際に中間コードを生成する部分を見ていきましょう。まず中間コードのためのクラス `IntCode` を見ていきますが、その中に入っているクラス `Inst` とそのサブクラス (個々の中間コード命を表す) は後回しにします。

```
package samc1;
import java.util.*;

public class IntCode {
  List<Block> code = new ArrayList<Block>();
  Block cb;
  int lblno = 1000;
  public IntCode() { cb = new Block("L1000", code.size()); code.add(cb); }
  public List<Block> getBlocks() { return code; }
  public void gLabel(int l) {
    if(cb.getTmp() && cb.getSize() == 0) { cb.setLabel("L"+l); }
    else { cb = new Block("L"+l, code.size()); code.add(cb); }
  }
```

まず、中間コードはブロックの集まりです。変数 `cb` が現在コードを追加中の (最後の) ブロックです。クラス `Block` はすぐ後で読みます。ブロックの区切りのラベルは L1000 以降を使うことにして、その連番を生成するカウンタもインスタンス変数とします。コンストラクタでは最初の空ブロックを生成します。コード内のブロックは ID 番号 (0 からの連番) を持たせ、この番号は変数 `code` に入っているリストの内の位置と一致しています。`gLabel()` はラベルを生成しますが、現在のブロックがまだ空で、そのブロックにもともとのラベルがないなら、そのブロックのラベルを取り換えて終わります。そうでないならそのラベルを持つ新しいブロックを追加します。

以下のメソッドはすべて、中間コード命令を末尾に追加するものです。個々のクラスは後で読みます。

```
  public void gMovereg(int t1, String r) { cb.add(new Movereg(t1, r)); }
  public void gMoveint(int t1, int v) { cb.add(new Moveint(t1, v)); }
  public void gMove(int t1, int t2) { cb.add(new Move(t1, t2)); }
```

```
public void gAdef(int t1, int t2, int t3) { cb.add(new Adef(t1, t2, t3)); }
public void gAref(int t1, int t2, int t3) { cb.add(new Aref(t1, t2, t3)); }
public void gAdd(int t1, int t2, int t3) { cb.add(new Add(t1, t2, t3)); }
public void gSub(int t1, int t2, int t3) { cb.add(new Sub(t1, t2, t3)); }
public void gMul(int t1, int t2, int t3) { cb.add(new Mul(t1, t2, t3)); }
public void gDiv(int t1, int t2, int t3) { cb.add(new Div(t1, t2, t3)); }
public void gJump(int l) { cb.add(new Jump(l)); eb(); }
public void gIfz(int t1, int l) { cb.add(new Ifz(t1, l)); eb(); }
public void gIfnz(int t1, int l) { cb.add(new Ifnz(t1, l)); eb(); }
public void gIfne(int t1, int t2, int l) { cb.add(new Ifne(t1,t2,l)); eb(); }
public void gIfeq(int t1, int t2, int l) { cb.add(new Ifeq(t1,t2,l)); eb(); }
public void gIfgt(int t1, int t2, int l) { cb.add(new Ifgt(t1,t2,l)); eb(); }
public void gIflt(int t1, int t2, int l) { cb.add(new Iflt(t1,t2,l)); eb(); }
public void gIfge(int t1, int t2, int l) { cb.add(new Ifge(t1,t2,l)); eb(); }
public void gIfle(int t1, int t2, int l) { cb.add(new Ifle(t1,t2,l)); eb(); }
private void eb() { gLabel(++lblno); cb.setTmp(true); }
public void show() { for(Block b: code) { b.show(); } }
public void calcconnect() {
  Map<String,Block> map = new HashMap<String,Block>();
  for(Block b: code) { map.put(b.getLabel(), b); }
  for(int i = 0; i < code.size(); ++i) {
    Block b1 = code.get(i);
    for(int j: b1.getJdsts()) {
      if(j < 0 && i < code.size()-1) { b1.connect(code.get(i+1)); }
      if(j >= 0) { b1.connect(map.get("L"+j)); }
    }
  }
}
public void showconnect() { for(Block b: code) { b.showconnect(); } }
```

eb() は分岐命令の後に呼び出され、そこでブロックを切ります。その後はコードの内容を表示するメソッド、ブロック間の接続を計算するメソッド、そして接続を表示するメソッドです。

クラス Block は基本的にラベルを 1 つ持つ命令の並びですが、制御フロー解析のため相互の接続関係を保持できるようになっていて、またデータフロー解析のため、このブロックで定義されるテンポラリ、参照されるテンポラリの情報を返せます (後で説明します)。

```
public static class Block {
  String label;
  List<Inst> body = new ArrayList<Inst>();
  Set<Integer> prev = new TreeSet<Integer>();
  Set<Integer> succ = new TreeSet<Integer>();
  Set<Integer> ref = null;
  Set<Integer> def = null;
  int bid = 0;
  boolean tmp = false;
  public Block(String l, int i) { label = l; bid = i; }
  public int getId() { return bid; }
```

```
    public String getLabel() { return label; }
    public void setLabel(String lbl) { label = lbl; }
    public boolean getTmp() { return tmp; }
    public void setTmp(boolean t) { tmp = t; }
    public int getSize() { return body.size(); }
    public void add(Inst i) { body.add(i); }
    public List<Inst> getBody() { return body; }
    public Set<Integer> getPrev() { return prev; }
    public Set<Integer> getSucc() { return succ; }
    public Set<Integer> getRef() { if(ref==null) { calcDR(); } return ref; }
    public Set<Integer> getDef() { if(def==null) { calcDR(); } return def; }
    private void calcDR() {
      def = new TreeSet<Integer>(); ref = new TreeSet<Integer>();
      for(Inst op: body) {
        for(Integer i: op.getRefs()) { if(!def.contains(i)) { ref.add(i); }}
        for(Integer i: op.getDefs()) { def.add(i); }
      }
    }
    public void connect(Block b1) { succ.add(b1.getId()); b1.prev.add(bid); }
    public int[] getJdsts() {
      if(body.size() == 0) { return new int[]{ -1 }; }
      Inst last = body.get(body.size()-1);
      int dst = last.getJdst();
      if(last instanceof Jump || dst < 0) { return new int[]{ dst }; }
      return new int[]{ dst, -1 };
    }
    public void show() {
      System.out.println(label + ": // B" + bid);
      for(Inst i: body) { System.out.println("  "+i); }
    }
    public void showconnect() {
      System.out.print("B" + bid);
      System.out.print(" prev(");
      for(int i: prev) { System.out.printf(" B%d", i); }
      System.out.print(" ) succ(");
      for(int i: succ) { System.out.printf(" B%d", i); }
      System.out.println(" )");
    }
  }
  // ここにクラス Inst とそのサブクラス
}
```

3.2 中間コード生成

それでは、実際に中間コードを生成する部分です。フロントエンド (意味解析まで) はこれまでと同じなので、SableCC の記述ファイル、`Log.java`、`Symtab.java`、`TypeChecker.Java` は省略します (パッケージ名のみ修正)。以下に中間コード生成用クラスを示します。

```
package samc1;
import samc1.analysis.*;
import samc1.node.*;
import java.io.*;
import java.util.*;

class GenIntcode extends DepthFirstAdapter {
  HashMap<Node,Integer> pos;
  Symtab st;
  IntCode ic;
  int pcnt = 0, lcnt = 0;
  static String preg[] = {"%rdi", "%rsi", "%rdx", "%rcx", "%r8", "%r9"};
  private int ti(Object o) { return (Integer)o; }
  public GenIntcode(Symtab s, HashMap<Node,Integer> p, IntCode i) {
    st = s; pos = p; ic = i;
  }
```

インスタンス変数としてはこれまでのものに加え、IntCodeのインスタンスを保持します。ti()というのはObject型から整数にキャストするための下請けメソッドです。

まず、冒頭のパラメタ部ではMoveregというのを生成して固定レジスタからテンポラリに値を移します。以降はすべてテンポラリのみで値をすべて扱います。式などはすべて意味スタックにテンポラリ番号を入れて戻るようにしています。そして各命令の生成はIntCodeのメソッド「gなんとか」を呼べばよいように作ってあったのでした。

```
  @Override
  public void outAIdclDcl(AIdclDcl node) {
    ic.gMovereg(pos.get(node), preg[pcnt++]);
  }
  @Override
  public void outAAdclDcl(AAdclDcl node) {
    ic.gMovereg(pos.get(node), preg[pcnt++]);
  }
  @Override
  public void outAAssignStat(AAssignStat node) {
    ic.gMove(pos.get(node), ti(getOut(node.getExpr())));
  }
  @Override
  public void outAAassignStat(AAassignStat node) {
    ic.gAdef(pos.get(node), ti(getOut(node.getIdx())), ti(getOut(node.getExpr())));
  }
```

if文やwhile文は条件ジャンプとラベルが必要なので、ラベル番号を用意し、条件ジャンプ命令の行き先や生成ラベルで使用します。あと、比較演算も大小に応じて0か1を返すので、内部ではラベルと条件ジャンプを使って組み立てています。

```
  @Override
  public void caseAIfStat(AIfStat node) {
```

```
    int lbl = lcnt; lcnt += 1;
    node.getExpr().apply(this);
    ic.gIfnz(ti(getOut(node.getExpr())), lbl);
    node.getStat().apply(this);
    ic.gLabel(lbl);
  }
  @Override
  public void caseAWhileStat(AWhileStat node) {
    int lbl = lcnt; lcnt += 2;
    ic.gLabel(lbl);
    node.getExpr().apply(this);
    ic.gIfz(ti(getOut(node.getExpr())), lbl+1);
    node.getStat().apply(this);
    ic.gJump(lbl);
    ic.gLabel(lbl+1);
  }
  @Override
  public void outAGtExpr(AGtExpr node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gIfgt(ti(getOut(node.getLeft())), ti(getOut(node.getRight())), lcnt);
    ic.gMoveint(f.pos, 0);
    ic.gJump(lcnt+1);
    ic.gLabel(lcnt++);
    ic.gMoveint(f.pos, 1);
    ic.gLabel(lcnt++);
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outALtExpr(ALtExpr node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gIflt(ti(getOut(node.getLeft())), ti(getOut(node.getRight())), lcnt);
    ic.gMoveint(f.pos, 0);
    ic.gJump(lcnt+1);
    ic.gLabel(lcnt++);
    ic.gMoveint(f.pos, 1);
    ic.gLabel(lcnt++);
    setOut(node, new Integer(f.pos));
  }
```

以下は普通の式の演算類なので、これまでと基本的に変わりません。ただ中間コードを生成するというだけです。

```
  @Override
  public void outAOneExpr(AOneExpr node) { setOut(node, getOut(node.getNexp())); }
  @Override
  public void outAAddNexp(AAddNexp node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
```

```
    ic.gAdd(f.pos, ti(getOut(node.getNexp())), ti(getOut(node.getTerm())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outASubNexp(ASubNexp node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gSub(f.pos, ti(getOut(node.getNexp())), ti(getOut(node.getTerm())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outAOneNexp(AOneNexp node) { setOut(node, getOut(node.getTerm())); }
  @Override
  public void outAMulTerm(AMulTerm node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gMul(f.pos, ti(getOut(node.getTerm())), ti(getOut(node.getFact())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outADivTerm(ADivTerm node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gDiv(f.pos, ti(getOut(node.getTerm())), ti(getOut(node.getFact())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outAOneTerm(AOneTerm node) { setOut(node, getOut(node.getFact())); }
  @Override
  public void outAIconstFact(AIconstFact node) {
    Symtab.Ent e = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gMoveint(e.pos, new Integer(node.getIconst().getText()));
    setOut(node, new Integer(e.pos));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, new Integer(pos.get(node)));
  }
  @Override
  public void outAArefFact(AArefFact node) {
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    ic.gAref(f.pos, pos.get(node), ti(getOut(node.getExpr())));
    setOut(node, new Integer(f.pos));
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOut(node.getExpr())); }
}
```

これが生成する中間コードについては、既に図 2 で示してあります。

3.3 コンパイラドライバ

今回はかなり色々な処理を試したり結果を表示したりしなかったりしたいので、コマンド引数で「java samc1/SamC1 ソース コマンド…」のようにしてコマンドを順番に実行するようなコンパイラドライバになりました。実は、図 2 の中間コードは「pi」コマンドで出力させたものです。なお、とりあえず、まだ出て来ていない部分はコメントアウトしたので、使う時はコメントを外してください。

```
  package samc1;
  import samc1.parser.*;
  import samc1.lexer.*;
  import samc1.node.*;
  import java.io.*;
  import java.util.*;

  public class SamC1 {
    public static void main(String[] args) throws Exception {
      if(args.length == 0) { showoptions(); return; }
      Parser p = new Parser(new Lexer(new PushbackReader(
        new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
          1024)));
      Start tree = p.parse();
      Symtab st = new Symtab();
      HashMap<Node,Integer> vtbl = new HashMap<Node,Integer>();
      TypeChecker tck = new TypeChecker(st, vtbl); tree.apply(tck);
      if(Log.getError() > 0) { return; }
      IntCode ic = new IntCode();
      GenIntcode gen = new GenIntcode(st, vtbl, ic); tree.apply(gen);
      ic.calcconnect();
//    Dataflow df = new Dataflow(ic.getBlocks());
      for(int i = 1; i < args.length; ++i) {
        if(args[i].equals("pt")) { st.show(); }
        else if(args[i].equals("pi")) { ic.show(); }
        else if(args[i].equals("pc")) { ic.showconnect(); }
//      else if(args[i].equals("vn")) { ValNumber.run(ic, false); }
//      else if(args[i].equals("pvn")) { ValNumber.run(ic, true); }
//      else if(args[i].equals("clio")) { df.calcLiveInOut(); }
//      else if(args[i].equals("plio")) { df.showLiveInOut(); }
        else { System.err.println("unknown command: "+args[i]); }
      }
    }
    private static void showoptions() {
      String[] msg = {
        "pt: print table",
        "pi: print intcode",
        "pc: print block-connection",
//      "vn: value numbering",
```

```
//      "pvn: value numvering with printout",
//      "clio: calculate LiveIn/LIveOut",
//      "plio: print LiveIn/LiveOut",
      };
      System.out.println("usage: java samc1/SamC1 <source> <option>...");
      for(String s: msg) { System.out.println("  " + s); }
    }
  }
```

では使用例として、ブロックの接続関係を表示させてみましょう (図 4)。

```
% java samc1/SamC1 test.min pc
B0 prev( ) succ( B1 )
B1 prev( B0 B13 ) succ( B2 B14 )
B2 prev( B1 ) succ( B3 )
B3 prev( B2 B12 ) succ( B4 B5 )
B4 prev( B3 ) succ( B6 )
B5 prev( B3 ) succ( B6 )
B6 prev( B4 B5 ) succ( B7 B13 )
B7 prev( B6 ) succ( B8 B9 )
B8 prev( B7 ) succ( B10 )
B9 prev( B7 ) succ( B10 )
B10 prev( B8 B9 ) succ( B11 B12 )
B11 prev( B10 ) succ( B12 )
B12 prev( B10 B11 ) succ( B3 )
B13 prev( B6 ) succ( B1 )
B14 prev( B1 ) succ( )
%
```

図 4: ブロックの接続関係表示

演習 1 自分でも同じものを動かしてみよ。動いたら、次のことをやってみよ。

a. 別のプログラムを作成して中間コードを表示してみよ。そのあと、手でフローグラフ (図 3 のようなもの) を描き、bc コマンドで表示させたものと一致していることを確認せよ。
b. 今回のコードでは自然ループの検出機能は実装していない。実装してみよ。
c. 上記に加え、ループごとに 1 つループヘッダのブロックがあるように修正してみよ。

4 局所最適化

4.1 局所最適化の考え方と手法

前述のように、局所最適化とは基本ブロックの中だけで実行できるような最適化を言います。具体的に考えるため、ソースコードで「`int z; z=a[i-1]; a[i-1]=a[i]; a[i]=z; remain=1;`」に対応する部分を図 2 のコードから抜き出したものを図 5 に示します。

よく見ると色々な無駄が見つかると思います。定数「`1`」を何回も別のテンポラリに入れていますし、添字「`i-1`」の計算も 2 回行なっていますし、代入するのにわざわざ別のテンポラリに計算して、それを改めてコピーしています。このような無駄は、普通にコード生成を行なうとどうしても生じてしまいます (頑張れば減らせますが、速度が問題になるようなコンパイラであれば、どのみち最適化で対処するのでほっていあるというべきかも)。

これらに対処するための局所最適化の手法として、次のようなものがあります。

```
L1007: // B11
  t16 = 1
  t17 = t5 - t16
  t18 = t3[t17]
  t6 = t18
  t19 = 1
  t20 = t5 - t19
  t21 = t3[t5]
  t3[t20] = t21
  t3[t5] = t6
  t22 = 1
  t4 = t22
```

図 5: 比較交換部分の中間コード (原型)

- **定数伝播** (constant propagation) — 定数どうしの演算式があったら、それをコンパイル時に計算してしまい、定数で置き換える。そんな無駄なコードは書かないと思うかも知れませんが、たとえば `SIZE` を 100 と定義し、添字上限として `SIZE-1` と書く、みたいなのは結局定数計算になります。
- **コピー伝播** (copy propagation) — 値を次々に代入 (コピー) している場合、最初の値を最後の行き先に直接代入すれば途中のコピーが不要になります。
- **共通部分式の削減** (common sub-expression elimination, CSE) — 演算式やその一部で同じ計算が重複している場合、最初の計算結果を保存して利用することで 2 回目以降の計算を省く。
- **不要コードの削減** (usless code elimination) — コピー伝播や CSE の結果、後から使われないテンポラリへの代入があれば、その代入コード自体を削除できます。また、その代入のための式の計算も芋づる式に削除できることがあります。

そのほか、上とは毛色が違いますが、計算の効率化やコード削減を行なう次のものもあります。

- 演算の知識を用いた簡略化や高速化 — 2 倍する代わりに足し算というのを先に挙げましたが、「1 倍する」「0 を足す」などはそもそも演算が不要になります。これらは定数伝播の結果変数だったものの値が分かり、それにより適用できることがあります。
- **不到達コードの削減** (dead code elimination) — if の条件が常に成立/不成立などと分かれば (これも定数伝播の結果分かることがあります)、決して通らないコードが分かり、削除できます。これはブロック内の最適化というより、定数伝播のあと制御フロー解析を行なうことで実行できる最適化になります。

4.2 命令クラス群の機能と構造

本題に入る前に、そろそろ避けて通れないので、保留にしてきた命令を表すクラス群 (`IntCode.Inst` とそのサブクラス群) を見ておきます。基本的な機能は次の通りです。

- 最初に命令を作ると、文字列表現を用意する。また、その命令が**定義する** (define、代入する) テンポラリと**参照する** (refer) テンポラリの番号を記録しておきます。
- 値番号による最適化のため、オペランド (0/1/2 個) を与えて右辺の計算式を表す文字列を返すメソッドと、新たなテンポラリ番号 (0/1/2 個) を与えて自身と同じ種別の命令を返すメソッドを用意しています。

```
public static class Inst {
  int[] defs = {};
```

```
  int[] refs = {};
  String s;
  int jdst = -1, val;
  protected void sd(int... a) { defs = a; }
  protected void sr(int... a) { refs = a; }
  public int[] getDefs() { return defs; }
  public int[] getRefs() { return refs; }
  public int getJdst() { return jdst; }
  public String toString() { return s; }
  public int assDest() { return -1; }
  public int refNum() { return 0; }
  public String refExp() { return null; }
  public String refExp(String v1) { return null; }
  public String refExp(String v1, String v2) { return null; }
  public Inst renew() { return this; }
  public Inst renew(int t1) { return this; }
  public Inst renew(int t1, int t2) { return this; }
}
static class Movereg extends Inst {
  public Movereg(int t1, String reg) {
    s = String.format("t%d = %s", t1, reg); sd(t1);
  }
}
static class Move extends Inst {
  public Move(int t1, int t2) {
    s = String.format("t%d = t%d", t1, t2); sd(t1); sr(t2);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 1; }
  public String refExp(String v1) { return v1; }
}
static class Moveint extends Inst {
  public Moveint(int t1, int v) {
    val = v; s = String.format("t%d = %d", t1, val); sd(t1);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 0; }
  public String refExp() { return val+""; }
}
static class Adef extends Inst {
  public Adef(int t1, int t2, int t3) {
    s = String.format("t%d[t%d] = t%d", t1, t2, t3); sd(t1); sr(t2, t3);
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Adef(defs[0], t1, t2); }
}
```

```
static class Aref extends Inst {
  public Aref(int t1, int t2, int t3) {
    s = String.format("t%d = t%d[t%d]", t1, t2, t3); sd(t1); sr(t2, t3);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 2; }
  public String refExp(String v1, String v2) { return v1+"["+v2+"]"; }
  public Inst renew(int t1, int t2) { return new Aref(defs[0], t1, t2); }
}
static class Add extends Inst {
  public Add(int t1, int t2, int t3) {
    s = String.format("t%d = t%d + t%d", t1, t2, t3); sd(t1); sr(t2, t3);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 2; }
  public String refExp(String v1, String v2) { return v1+"+"+v2; }
  public Inst renew(int t1, int t2) { return new Add(defs[0], t1, t2); }
}
static class Sub extends Inst {
  public Sub(int t1, int t2, int t3) {
    s = String.format("t%d = t%d - t%d", t1, t2, t3); sd(t1); sr(t2, t3);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 2; }
  public String refExp(String v1, String v2) { return v1+"-"+v2; }
  public Inst renew(int t1, int t2) { return new Sub(defs[0], t1, t2); }
}
static class Mul extends Inst {
  public Mul(int t1, int t2, int t3) {
    s = String.format("t%d = t%d * t%d", t1, t2, t3); sd(t1); sr(t2, t3);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 2; }
  public String refExp(String v1, String v2) { return v1+"*"+v2; }
  public Inst renew(int t1, int t2) { return new Mul(defs[0], t1, t2); }
}
static class Div extends Inst {
  public Div(int t1, int t2, int t3) {
    s = String.format("t%d = t%d / t%d", t1, t2, t3); sd(t1); sr(t2, t3);
  }
  public int assDest() { return defs[0]; }
  public int refNum() { return 2; }
  public String refExp(String v1, String v2) { return v1+"/"+v2; }
  public Inst renew(int t1, int t2) { return new Div(defs[0], t1, t2); }
}
static class Jump extends Inst {
```

```
  public Jump(int lbl) {
    s = String.format("jmp L%d", lbl); jdst = lbl;
  }
}
static class Ifz extends Inst {
  public Ifz(int t1, int lbl) {
    s = String.format("ifz t%d then L%d", t1, lbl); sr(t1); jdst = lbl;
  }
  public int refNum() { return 1; }
  public Inst renew(int t1) { return new Ifz(t1, jdst); }
}
static class Ifnz extends Inst {
  public Ifnz(int t1, int lbl) {
    s = String.format("ifnz t%d then L%d", t1, lbl); sr(t1); jdst = lbl;
  }
  public int refNum() { return 1; }
  public Inst renew(int t1) { return new Ifnz(t1, jdst); }
}
static class Ifeq extends Inst {
  public Ifeq(int t1, int t2, int lbl) {
    s = String.format("if t%d == t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifeq(t1, t2, jdst); }
}
static class Ifne extends Inst {
  public Ifne(int t1, int t2, int lbl) {
    s = String.format("if t%d != t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifne(t1, t2, jdst); }
}
static class Ifgt extends Inst {
  public Ifgt(int t1, int t2, int lbl) {
    s = String.format("if t%d > t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifgt(t1, t2, jdst); }
}
static class Iflt extends Inst {
  public Iflt(int t1, int t2, int lbl) {
    s = String.format("if t%d < t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
```

```
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Iflt(t1, t2, jdst); }
}
static class Ifge extends Inst {
  public Ifge(int t1, int t2, int lbl) {
    s = String.format("if t%d >= t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifge(t1, t2, jdst); }
}
static class Ifle extends Inst {
  public Ifle(int t1, int t2, int lbl) {
    s = String.format("if t%d <= t%d then L%d", t1, t2, lbl); sr(t1, t2);
   jdst = lbl;
  }
  public int refNum() { return 2; }
  public Inst renew(int t1, int t2) { return new Ifle(t1, t2, jdst); }
}
```

4.3 値番号法による最適化

値番号法 (value numbering method) とは、比較的古くからある最適化手法で、共通部分式の削減とコピー伝播がまとめて行なえます。ブロックをまたがった広域最適化に拡張した **GVN**(global value numbering) もありますが、ここでは分かりやすい局所最適化の範囲で説明します。

その考え方は次のようなものです。プログラムの中で各テンポラリが保持する値は実行時まで分かりませんが、これを「1 番目の値」「2 番目の値」のように番号づけすることで区分します。表記が長いので `v1`、`v2` のように記すこととして、演算命令で例えば `v1+v2` というのができたら、その式に対応する値番号を割り当てます。後で再度同じものが出て来たら、それらは同じ値なので片方だけ計算すれば済みます。

それぞれのテンポラリについて、値がいきなり参照されたら (それはもっと前のブロックで計算された値が入っているわけなので) 新しい値番号を割り当て、また代入されたら右辺の値番号に対応づけます。

こうすることで、見た目は (コード上では) 同じ式でも、中に現れる変数に別の値が入っている場合は、値番号も代わるので同じ式であると勘違いすることは避けられるわけです。

では、先の中間コード辺に値番号をつけてみましょう (図 6)。かなり演算が削減できることが分かります。

では、これを行なうコードを見てみます。局所最適化なので、各ブロックごとに独立に `ValNumber.run` を (ブロックの本体を渡して) 呼び出します。その中では `ValNumber` オブジェクトを (ブロック本体を渡して) 生成したあと、`exec()` メソッドを呼びます。オブジェクトの中では「テンポラリ番号→値番号」「値番号→テンポラリ番号」「式文字列→値番号」の対応を保持する 3 つの表が保持されています。

`exec()` の中では 1 命令ずつ順に取り出し、まず定義なしで参照されるテンポラリに値番号を割り振ります (`insttmp()`)。次に、命令の種別ごとに分かれます。

- 代入しない命令の場合、作り直すだけでよい。このとき参照しているテンポラリは必要なら値番号に応じてコピー元のテンポラリに置き換えられる。

```
L1007: // B11
  t16 = 1           ; t16: 1
  t17 = t5 - t16    ; t17: v2 = v1 - 1
  t18 = t3[t17]     ; t18: v4 = v3[v2]
  t6 = t18          ; t6:  v4
  t19 = 1           ; t19: 1
  t20 = t5 - t19    ; t20: v2
  t21 = t3[t5]      ; t21: v5 = v3[v1]
  t3[t20] = t21     ;      v3[v2] = v5
  t3[t5] = t6       ;      v3[v1] = v4
  t22 = 1           ; t22: 1
  t4 = t22          ; t4:  1
```

図 6: 中間コードに値番号をつけたもの

- 代入する場合は、その代入する値番号を保持しているテンポラリがあるか調べる。もしなければ、計算は必要なので同じ種別の計算命令を生成し直し、そのテンポラリに値番号を登録する。
- 代入する場合で、同じ値番号を持つテンポラリがある場合、そのテンポラリの値番号が既に変更されてしまっていないかチェックする。OK なら、この命令をコピー命令に置き換え、代入先のテンポラリの値番号を登録する。変更されてしまっているなら、前項と同じ動作。

```
package samc1;
import java.util.*;

public class ValNumber {
  Map<Integer, String> tmpval = new HashMap<Integer,String>();
  Map<String, Integer> valtmp = new HashMap<String,Integer>();
  Map<String, String> expval = new HashMap<String,String>();
  int vcnt = 1;
  public void exec(List<IntCode.Inst> lst, boolean pr) {
    for(int k = 0; k < lst.size(); ++k) {
      IntCode.Inst op = lst.get(k), op1 = op;  insttmp(op);
      int d = op.assDest();
      if(d < 0) {
        op1 = renew(op);
      } else {
        String exp = newexp(op), v = exp2val(exp);
        if(!valtmp.containsKey(v)) {
          op1 = renew(op);
          tmpval.put(d, v); valtmp.put(v, d);
          if(pr) { System.out.printf(" t%d:%s = %s\n", d, v, exp); }
        } else {
          int t = valtmp.get(v);
          if(tmpval.get(t).equals(v)) {
            op1 = new IntCode.Move(d, t);
            tmpval.put(d, v);
            if(pr) { System.out.printf(" t%d:%s = t%d\n", d, v, t); }
          } else {
            op1 = renew(op);
```

```
          tmpval.put(d, v); valtmp.put(v, t);
          if(pr) { System.out.printf(" t%d:%s = %s\n", d, v, exp); }
        }
      }
    }
    if(pr) { System.out.println("        " + op + " ==> " + op1); }
    lst.set(k, op1);
  }
  if(!pr) { return; }
  System.out.println("--------");
  for(Integer i: tmpval.keySet()) { pl("  t"+i+" = "+tmpval.get(i)); }
  System.out.println("--------");
  for(String e: valtmp.keySet()) { pl("  "+e+" = t"+valtmp.get(e)); }
  System.out.println("--------");
  for(String e: expval.keySet()) { pl("  "+e+" = "+expval.get(e)); }
  System.out.println("--------");
}
private void pl(String s) { System.out.println(s); }
private boolean alldigit(String s) { return s.matches("^[0-9]+$"); }
private String exp2val(String e) {
  if(expval.containsKey(e)) { return expval.get(e); }
  if(alldigit(e)) { return e; }
  if(e.charAt(0) == 'v' && alldigit(e.substring(1))) { return e; }
  String v = "v"+vcnt++; expval.put(e, v); return v;
}
private void insttmp(IntCode.Inst op) {
  for(int i = 0; i < op.refNum(); ++i) {
    int t = op.getRefs()[i];
    if(!tmpval.containsKey(t)) {
      String v = "v"+vcnt++; tmpval.put(t, v); valtmp.put(v, t);
    }
  }
}
private String newexp(IntCode.Inst op) {
  String e = "?";
  int[] ref = op.getRefs();
  if(op.refNum() == 0) {
    e = op.refExp();
  } else if(op.refNum() == 1) {
    e = op.refExp(tmpval.get(ref[0]));
  } else {
    e = op.refExp(tmpval.get(ref[0]), tmpval.get(ref[1]));
  }
  return e;
}
private IntCode.Inst renew(IntCode.Inst op) {
```

```
      int[] r = op.getRefs();
      if(op.refNum() == 0) {
        return op.renew();
      } else if(op.refNum() == 1) {
        Integer i1 = valtmp.get(tmpval.get(r[0]));
        return i1 == null ? op : op.renew(i1);
      } else {
        Integer i1 = valtmp.get(tmpval.get(r[0]));
        Integer i2 = valtmp.get(tmpval.get(r[1]));
        return i1 == null || i2 == null ? op : op.renew(i1, i2);
      }
    }
    public static void run(IntCode ic, boolean pr) {
      for(IntCode.Block b: ic.getBlocks()) {
        if(pr) { System.out.println(b.getLabel()); }
        new ValNumber().exec(b.getBody(), pr);
      }
    }
  }
```

では「java samc1/SamC1 test.min vn pi」で出力した局所最適化すみ中間コードの当該ブロック部分を見てみましょう (図 7)。

```
L1007: // B11
  t16 = 1
  t17 = t5 - t16
  t18 = t3[t17]
  t6 = t18
  t19 = t16
  t20 = t17
  t21 = t3[t5]
  t3[t17] = t21
  t3[t5] = t18
  t22 = t16
  t4 = t16
```

図 7: 最適化を行なった中間コード

確かに、余分な演算が削減されています。ただ、使われなくなったコピー文も (ここまででは不要命令の削除をしないため) 残っている。参照されていないテンポラリへの代入は削除していいでしょうか? それは NO で、このブロックから外に出たところで参照さているかも知れないので、やみくもに削除はできません。つまり、それをやるためにはより詳しい解析情報の抽出、具体的にはデータフロー解析が必要になるのです。

演習 2 例題をそのまま動かせ。動いたら、自分で作成したさまざまなコードに対して、値番号による局所最適化の有無で中間コードがどのように違うかを検討せよ。

5　データフロー解析

5.1　データフロー解析とデータフロー方程式

前節で述べたように、ある程度以上の最適化のためには「この変数の値はこの後参照されるだろうか。」(されないなら、その変数に値を格納しておく命令は不要)、「この場所でこの変数に入っている値を設定する命令はどれとどれだろうか。」(それが 1 つだけで、その値がたまたま別の変数にも入っているなら実は変数そのものが要らないかもしれない)、などの情報が必要になります。

この種の情報をフローグラフから抽出するのが**データフロー解析** (dataflow analysis) です。以下では代表的なデータフロー解析の問題について説明していきます。

生きている変数の問題

データフロー解析の最初の例として、先にあげた「この変数の値はこの後参照されるだろうか。」という問題を考えます。これは変数が**生きている** (live)、つまりその変数の値が後で参照される可能性があるか、あるいは**死んでいる** (dead)、つまりその可能性がないか、を決めるものです。

個々のブロック b について見ると、その入口で生きている変数と出口で生きている変数の間には次の関係があります。

$$
\begin{aligned}
LiveOut[b] &= \bigcup_{b' \in Succ[b]} LiveIn[b'] \\
LiveIn[b] &= Ref[b] \cup (LiveOut[b] - Ass[b])
\end{aligned}
$$

これは、b の出口で生きている変数の集合とは b に引き続くようなブロック b' の入口で生きている変数の和集合であり、また b の入口で生きている変数の集合は出口で生きている変数の集合から b で値を設定してしまう変数は除き、代りに b で (もし値を設定するならそれより前に) 参照する変数を加えたものであることを表しています。

次にフローグラフ G に関してこの**データフロー方程式** (dataflow equation) を解きます。それには次の手順を用います。

1. まず、各ブロックについて $Ref[b]$、$Ass[b]$ を求める。
2. $LiveIn[b]$、$LiveOut[b]$ についてはとりあえず空集合とする。
3. 各ブロックについて上の方程式に従って $LiveIn[b]$、$LiveOut[b]$ を計算することを反復することをもはや各集合が変化しなくなるまで行う。

各反復において、$LiveIn[b]$ と $LiveOut[b]$ は (変化するとすれば) 要素がつけ加わる方向にのみ変化し、そして変数の個数は有限ですから、この手順は必ず停止し解が求まります。

アルゴリズムは上記の通りですが、上の方程式はブロック出口の値に基づいて入口の値を求めます。そのため、実際にブロックを調べる順番は後のブロックから前に向かって行なうと効率がよくなります。このような問題を**後向きフロー解析** (backward flow analysis) の問題と呼びます。

UD 連鎖の問題

次の例として、「この場所でこの変数に入っている値を定義する命令はどれとどれか」の問題を取り上げます。その前にいくつか用語を説明しましょう。

定義　変数 x **への定義** (definition) とは、変数に値を設定する可能性をもつ命令を言う。典型的には x への代入がそうだが、x を参照渡し引数とする手続き呼出しは x を更新する可能性があるため x の定義となる。x に必ず値が設定される場合にはその定義は**曖昧でない** (unambiguous) という。

定義 変数 x に対する定義 d と命令 s について、d から s へ至る経路上に x に対する別の曖昧でない定義 d' があるとき、d' は d を**殺す** (kill) という。言い換えれば d' によってその経路を通るときは s への d の影響が及ばなくなる。

定義 定義 d と命令 s について、d を殺す別の定義が存在しないような d から s への経路が 1 つ以上存在するとき、d は s に**到達** (reach) する、という。

以上の用語から、この問題は**到達する定義** (reaching definition) の問題、または変数を使う (use) 場所ごとに、それに影響し得る定義 (definition) の連鎖を求めるため **UD 連鎖** (use-definition chain) の問題と呼ばれます。この問題については、個々のブロック b について次のデータフロー方程式が成り立ちます。

$$
\begin{aligned}
DefIn[b] &= \bigcup_{b' \in Pred[b]} DefOut[b'] \\
DefOut[b] &= Def[b] \cup (DefIn[b] - Kill[b])
\end{aligned}
$$

すなわち、ブロック b に入ってくる定義の集合はそのブロックの上流である全ブロックから出てくる定義を全部合せたものであり、また b から出ていく定義は b における定義と、入ってきた定義のうち b で殺されないものとを合せたものとなる、ということです。

このデータフロー方程式は先の「生きている変数」の式と In/Out が逆になった形をしています。このため、解を求める手順は前と同様でよいのですが、実用的には手続きの入口点から先に向かって反復計算すると効率が良くなります。このような問題を**前向きフロー解析** (forward flow analysis) の問題と呼びます。

DU 連鎖の問題

UD-連鎖が「各参照ごとに、そこに到達し得る定義を求める」のに対し、その逆、つまり「各定義ごとに、それに到達され得る参照を求める」問題を **DU 連鎖** (definition-use chain) の問題と呼びます。これを求めるデータフロー方程式は次のようになります。

$$
\begin{aligned}
UseOut[b] &= \bigcup_{b' \in Succ[b]} UseIn[b'] \\
UseIn[b] &= ExposedRef[b] \cup (UseOut[b] - UseKill[b])
\end{aligned}
$$

ただし $ExposedRef[b]$ は b の中で定義されないか、または定義されてもそれより前の位置で値を参照する文の集合、$UseKill[b]$ は b で定義される値を参照する b 以外の場所にある文の集合を意味します。各集合の意味は違っても、方程式の形は最初に述べた生きている変数の問題と同じですね。

5.1.1 利用可能式の問題

ここまでに示したデータフロー方程式はいずれも、上流/下流のどこかで問題としている事象 (値の定義とか変数の参照) が起き得るかどうかを知るためのものであり、その主旨から**いずれかの経路解析** (any-path analysis) と呼ばれます。

データフロー解析にはそれと対をなす**全ての経路解析** (all-path analysis) も存在します。その例として、ブロックにまたがる共通部分式の最適化を考えましょう。コード上のある場所に出てきた式の値を改めて計算しないですむためには、その上流の「全ての経路で」その式の値が計算ずみである必要がありますね。この問題を**利用可能式** (available expression) の問題と呼び、次のデータフロー方程式で表されます。

$$
\begin{aligned}
AvailIn[b] &= \bigcap_{b' \in Pred[b]} AvailOut[b'] \\
AvailOut[b] &= DefExp[b] \cup (AvailIn[b] - KillExp[b])
\end{aligned}
$$

ここで $DefExp[b]$ は b で計算される式の集合、$KillExp[b]$ は b において式に含まれる変数のどれかが殺されるため計算ずみの値が有効でなくなる式の集合を意味します。この方程式もこれまでと同様の解法で扱うことができますが、ただし計算が $\bigcap$ を用いて集合を小さくする方向に進むので、$AvailIn[b]$、$AvailOut[b]$ の初期値は初期ブロックのみ空集合、あとは全ての式の集合とする必要があります。

コピー文の問題

利用可能式と類似した問題に**コピー文** (copy statement) の問題があります。これは例えばある場所に `y = x` という代入 (コピー文) があり、後続するブロックに `y` の参照があったとき、これを `x` に置き換えてしまってよいかどうかを決めるものです。つまり、値番号法でやったものを広域最適化でやるわけです。置換えが可能なためには、次の条件が必要です。

- `y` を参照している箇所に到達する全ての `y` に対する定義が `y := x` である。
- `y = x` が行われた後 `y` の参照までに、`x` の値を変更する可能性がない。

このうち前者は UD 連鎖の情報をもとに知ることができ、後者は次のデータフロー方程式により計算できます。

$$
\begin{aligned}
CopyIn[b] &= \bigcap_{b' \in Pred[b]} CopyOut[b'] \\
CopyOut[b] &= DefCopy[b] \cup (CopyIn[b] - KillCopy[b])
\end{aligned}
$$

ここで $CopyIn[b]/CopyOut[b]$ はそれぞれ b の入口/出口で置き換え可能なコピー文の集合、$DefCopy[b]$ は b に含まれるコピー文でその後 b 内で左辺、右辺どちらも変更されないものの集合、$KillCopy[b]$ は b **以外の**場所にある全てのコピー文から b 内で左辺、右辺どちらかが変更されるものを除いた集合です。

5.2　生きている変数の問題を解く

それでは 1 つだけ具体例として、生きている変数の問題を解くコードを見ていただきましょう。今回のコードはこれで最後です (おつかれ様です)。クラスの前半にサポート用のメソッドがあり、そこで「各ブロックごとに空集合を用意」とか和集合、共通集合の演算を作成しています。

実際にデータフロー方程式を解くところは、先に述べた定義通りに集合演算をおこない、変化がなくなるまで繰り返すだけです。

```
package samc1;
import java.util.*;
public class Dataflow {
  List<IntCode.Block> blocks;
  public Dataflow(List<IntCode.Block> b) { blocks = b; }
  private List<Set<Integer>> newListSet() {
    List<Set<Integer>> lst = new ArrayList<Set<Integer>>();
    for(IntCode.Block b: blocks) { lst.add(new TreeSet<Integer>()); }
    return lst;
  }
  private static Set<Integer> add(Set<Integer> s1, Set<Integer> s2) {
    Set<Integer> s3 = new TreeSet<Integer>();
    for(Integer i: s1) { s3.add(i); }
    for(Integer i: s2) { s3.add(i); }
```

```
    return s3;
  }
  private static Set<Integer> sub(Set<Integer> s1, Set<Integer> s2) {
    Set<Integer> s3 = new TreeSet<Integer>();
    for(Integer i: s1) { if(!s2.contains(i)) { s3.add(i); } }
    return s3;
  }

  List<Set<Integer>> liveIn, liveOut;
  List<Set<Integer>> blkRef, blkDef;
  public List<Set<Integer>> getLiveIn() { return liveIn; }
  public List<Set<Integer>> getLiveOut() { return liveOut; }
  public void calcLiveInOut() {
    liveIn = newListSet(); liveOut = newListSet();
    boolean chg = true;
    while(chg) {
      chg = false;
      for(IntCode.Block b: blocks) {
        Set<Integer> li1, lo1 = new TreeSet<Integer>();
        for(Integer i: b.getSucc()) { lo1 = add(lo1, liveIn.get(i)); }
        li1 = add(b.getRef(), sub(lo1, b.getDef()));
        int i = b.getId();
        if(!(liveIn.get(i).equals(li1))) { liveIn.set(i, li1); chg = true; }
        if(!(liveOut.get(i).equals(lo1))) { liveOut.set(i, lo1); chg = true; }
      }
    }
  }
  public void showLiveInOut() {
    for(IntCode.Block b: blocks) {
      int id = b.getId();
      System.out.printf("LiveIn/Out[B%d] = {", id);
      for(Integer i: liveIn.get(id)) { System.out.print(" "+i); }
      System.out.print(" } {");
      for(Integer i: liveOut.get(id)) { System.out.print(" "+i); }
      System.out.println(" }");
    }
  }
}
```

生きている変数の問題は変数 (テンポラリ) の集合を対象にしていたので比較的作りやすかったのですが、UD 連鎖などの問題は「計算を行なう命令」が対象なので、個々の命令に番号をつけて扱う必要があり、だいぶ面倒になると思います。そういうわけで、ここでは作成していません。

とりあえず、局所最適化したあとの中間コードについて生きている変数の結果を表示させてみました (図 8)。

これで、先に出て来たブロック (B11) の出口で生きている変数は t2, t3, t4, t5 だけと分かります。そうなれば、コードを下から順に処理し、生きていない変数を定義する命令には削除の印をつけ、またそうでない命令が参照している変数は生きている集合に追加します (図 9)。

```
% java samc1/SamC1 test.min vn clio plio
LiveIn/Out[B0] = { } { 2 3 4 }
LiveIn/Out[B1] = { 2 3 4 } { 2 3 }
LiveIn/Out[B2] = { 2 3 } { 2 3 4 5 }
LiveIn/Out[B3] = { 2 3 4 5 } { 2 3 4 5 }
LiveIn/Out[B4] = { 2 3 4 5 } { 2 3 4 5 10 }
LiveIn/Out[B5] = { 2 3 4 5 } { 2 3 4 5 10 }
LiveIn/Out[B6] = { 2 3 4 5 10 } { 2 3 4 5 }
LiveIn/Out[B7] = { 2 3 4 5 } { 2 3 4 5 }
LiveIn/Out[B8] = { 2 3 4 5 } { 2 3 4 5 15 }
LiveIn/Out[B9] = { 2 3 4 5 } { 2 3 4 5 15 }
LiveIn/Out[B10] = { 2 3 4 5 15 } { 2 3 4 5 }
LiveIn/Out[B11] = { 2 3 5 } { 2 3 4 5 }
LiveIn/Out[B12] = { 2 3 4 5 } { 2 3 4 5 }
LiveIn/Out[B13] = { 2 3 4 } { 2 3 4 }
LiveIn/Out[B14] = { } { }
%
```

図 8: 生きている変数の結果

これで「X」とついている命令は削除でき、11 命令が 7 命令に減らせました。

演習 3 この例題をそのまま動かしてみよ。また、自分でも別のプログラムを処理してみて、どれくらい削除できるか検討せよ。

演習 4 上記の手順による不要コード削除を実装してみよ。

```
L1007: // B11
  t16 = 1            ↑ ; t2 t3 t4 t5 t16 t17 t18 t21
  t17 = t5 - t16     ↑ ; t2 t3 t4 t5 t16 t17 t18 t21
  t18 = t3[t17]      ↑ ; t2 t3 t4 t5 t16 t17 t18 t21
  t6 = t18           ↑ ; X
  t19 = t16          ↑ ; X
  t20 = t17          ↑ ; X
  t21 = t3[t5]       ↑ ; t2 t3 t4 t5 t16 t18 t21
  t3[t17] = t21      ↑ ; t2 t3 t4 t5 t16 t18 t21
  t3[t5] = t18       ↑ ; t2 t3 t4 t5 t16 t18
  t22 = t16          ↑ ; X
  t4 = t16           ↑ ; t2 t3 t4 t5 t16
```

図 9: 不要コードの削除のようす

6 課題 12A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル –「システムソフトウェア特論 課題# 12」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。

- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。
  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

A ループ最適化

ループの内部はコードの他の部分に比べて多数回実行されるため、ループ外コードの実行時間を犠牲にしてでもループ内コードの実行時間を短縮することはプログラム全体の実行時間短縮に寄与します。これを**ループ最適化** (loop optoimization) と呼びます。ループ最適化も中間コードに対して適用しますが、読みやすさのため以下ではソースコード表現を用いて説明します。

ループ不変文の移動

ループ不変文の移動 (loop invariant code motion) とは、ループの周回を通じて結果が変化しない計算を行う文をプリヘッダに移動して 1 回の実行で済ませることを言います。まず、ループ L について不変な文を次のようにして求めます。

1. オペランドが定数、または到達する定義が全て L 外にある変数のみから成る文に「ループ不変」の印をつける。
2. オペランドが定数、または到達する定義が全て L 外にある変数、または到達する定義が 1 つだけであり、それが L に含まれていて、その定義に「ループ不変」の印がついているような変数のみから成る文にも「ループ不変」の印をつける。

上記により見つかったループ不変文をそのままプリヘッダに移してよいわけではありません。今回の冒頭の説明のように、ループの条件が成り立った時しか実行したらまずいものもあるからです。そこで、次の条件を満たす時だけ移動するという方法があります。

(a) s は必ず 1 回以上実行される。
(b) x と同じ変数を定義する文は L 内では s だけである。
(c) L 内の全ての x の使用について、そこに到達する定義は s だけである。

(a) を調べるにはループの出口となるブロック (L に含まれない後続ブロックをもつような L のブロック) 全てが s を含むブロックに支配されるかどうかを調べます。もしそうなら、s を通らずにループを出る方法はないのだから、s は必ず 1 回以上実行されます。これらの制約を満たすことは s を移動できる必要十分条件ではありませんが、手持ちのコード解析情報から調べるのが容易であり、移動をそのような s に限ったとしてもある程度の最適化が行えることが経験的に知られています。

`while` ループや `for` ループではループ本体が 1 回も実行されない可能性があるので、条件 (a) がある限りループ本体に含まれる文はどれも移動できません。それに対しては、冒頭でやったように if で囲むようにすれば、移動できるようになります。

ループ内分岐の移動

ループ不変文の検出に伴い、if 文などの条件式がループ不変であることが分かる場合があります。そのときは、その if を次のように外に出すことが考えられます。つまり、左を右のように変更するわけです。これによりコード量は増えますが、実行時間は短くできます。

```
t1 = ...                t1 = ...
while(...) {            if(t1)
    C;                      while(...) {
    if(t1)                    C; A; D;
         A;                 }
    else                else
         B;                 while(...) {
    D;                        C; B; D;
}                           }
```

帰納変数の最適化

ループ最適化で重要なものの 1 つに、ループとともに変化する変数 (特にループ周回ごとに定数値だけ増減するもの) に関するものがあります。まず次の用語を定義します。

定義 **永続ループ変数** (persistent loop variable) とは、ループ入口において生きていて、なおかつループ内で定義される (言い換えればループ内でまず参照され、続いて定義される) 変数を言う。

定義 **ループ帰納変数** (loop induction variable) とは、ループの周回とともに値が一定値ずつ増加 (または減少) するような変数を言う

多くの場合ループを制御する変数は永続ループ変数になります。次の例を考えてみてください。

```
i := 0;                         p := top;
while i < x do begin            while p <> nil do begin
    ...                             ...
    i := i + 1                      p := p^.next
end                             end
```

左の場合 `i` は永続ループ変数でありかつ帰納変数でもあります。右の場合 `p` は永続ループ変数だが帰納変数ではありません。

帰納変数は永続ループ変数である場合 (基本帰納変数とよばれる) 以外に、別の帰納変数の値をもとに値を計算している場合もありますが、そのような帰納変数ば基本機能変数に変換できます (ループ 1 周回ごとに定数を加減するコードに置き換える)。このような変換は多くの場合乗算を加減算で置き換えることになるので、演算強さの軽減が行えることになります。

さらに、別の帰納変数と常に同じ値をもつことがわかる帰納変数や、条件判定にしか現れず、その判定条件を別の帰納変数を使うように書き換えられる帰納変数は、取り除いてしまうことができます。また、帰納変数はループ入口での値と終了条件を調べることで取り得る値の範囲を比較的容易に知ることができ、この情報を用いることで条件式が定数式である場合を多く検出できます。

まず基本帰納変数の検出は次のようにして行えます。

- 変数 i が永続ループ変数であり、かつループ内での i に対する定義 s が `i := i ± ` C(ただし C はループ不変) の形であり、s がループ周回ごとに必ず 1 回実行されるならば i は帰納変数である。

s がループ周回ごとに必ず 1 回実行されるかどうかは、ループ帰辺の直前のブロック b が s を含むブロックに支配されるかどうかで調べられます。次に、基本帰納変数以外の帰納変数を考慮します。

- 変数 j が永続ループ変数でなく、かつループ内の j に対する定義 s が `j := i * c ± d`(ただし `i` は帰納変数、`c`、`d` はループ不変) の形であり、s がループ周回ごとに必ず 1 回実行されるなら j は帰納変数である。

ここで i が基本帰納変数であるとき、j は i の**家族** (family) であると言います。i が基本帰納変数でないとき、i は別の基本帰納変数 i_0 の家族なので、次の 2 条件が成り立てば j も i_0 をもとに計算するように (つまり i_0 の家族に) できます。

1. i への定義と s の間に i_0 への定義がはさまっていない。
2. ループ外の i への定義が s に到達することがない。

例として、次のコードを考えてみます。

```
i0 = 0;
while(...) {
    i = i0 * 2 + 1;
    ...
    i0 = i0 + 1;
    j = i * 2 + 3;  /* s */
    ...
 }
```

```
i0 = 0;
while(...) {
    j = i * 2 + 3;   /* s */
    i = i0 * 2 + 1;
    ...
    i0 = i0 + 1;
}
```

ここで左側のコードは条件 1、右側のコードは条件 2 を満たしません。右の場合は j の値がループの最初の周回と 2 周回目の間で定数増減にならないので、帰納変数ではありません。左の場合は j の値が s の時点におけるより 1 周回前の i_0 の値によって計算しなければならないことを注意しておけば、帰納変数として最適化することができます。

次に永続ループ変数でない帰納変数 j を永続ループ変数に変換しますが、その手順は次の通り。

1. 新しい変数 u を導入し、その初期値設定をループのプリヘッダに追加する。
2. 家族の基となっている変数の更新の直後に、u の増減を行う定義を追加する。
3. j の定義を j := u に取り替える。

もちろん、新しく追加される複数の変数が同じ初期値、増減値、挿入位置であるなら、これらを 1 つで兼ねます。上の例に対して適用した結果は次のようなコードになるでしょう。

```
i0 = 0; u = 1; v = 5;
while(...) {
    i = u;
    ...
    i0 = i0 + 1; u = u + 2; v = v + 4; j = v;
    ...
}
```

この状態では演算強さは軽減された代りに値のコピーが増えていますが、コピー伝播を行なえば不要なコピーは削除できます。加えて、帰納変数を基本機能変数に変換した結果、これまで他の帰納変数を計算するためだけに使われていた帰納変数の参照がなくなる場合があります。その場合はその変数の計算も含めて不要コードとして削除できます。

また、同じ参照でも条件式においてループ不変式と比較されるもののみが残った場合は、これを別の帰納変数に関する判定に置き換えて取り除くことができます。

ループ展開

ループ展開 (loop unrolling) とは、ループ本体を複数回重複させることによりループのための分岐や判定の実行回数を減らす最適化を言います。最も極端には、ループの周回回数が翻訳時にわかり、

その値があまり大きくなく、ループ本体も小さい場合には、その回数だけ本体を展開することでループを全くなくしてしまう場合もあります。

そのように極端でなくても、ループ周回数が偶数であるとわかればループ本体を 2 回展開することにより、展開後のループ周回数は半分になり、ループ制御に要するオーバヘッドは半分にできます。また、展開された 2 個のループ本体について、共通式のくくり出しができる場合も多く、そのときはさらに実行速度を高められます。ループ周回数がわからない場合には、ループの終了判定を展開した本体 1 個につき 1 回ずつ行います (例えば下記の左のコードを右のように変形する)。

```
while(C) {              while(C) {
    B;                      B;
}                           if(!C) break;
                            B;
                        }
```

この場合でもループ先頭への分岐は減りますし、この変形の結果、共通式のくくり出しが可能になる場合もあります。

ループ融合

ループ融合 (loop fusion) とは、同一周回数をもつループが隣接している場合にそれらをまとめて 1 つのループにしてしまうことを言います。例えば次のような場合です。

```
for i := 1 to n do a[i] := i;        for i := 1 to n do begin
for i := 1 to n do b[i] := n - i;        a[i] := i; b[i] := n - 1
                                     end;
```

融合によってループ制御のオーバヘッドは半分になります。制御変数の範囲が同一でない場合には、単純に半分ではありませんが、終了判定とループ先頭への分岐は節約できます。もちろん、融合しても元のコードと意味が変らないためには、双方のループ本体に干渉がないことが必要です。

配列の線形化

ループ融合と類似していますが、2 次元配列 (一般には多次元配列) を順番にアクセスするコードはループの入れ子になります。

```
a: array[1..10][1..10] of ...;
...
for i := 1 to 10 do
    for j := 1 to 10 do a[i][j] := 0.0;
```

この場合、配列 a は主記憶上に連続して配置されているので、それを仮想的に 1 次元配列だとみなして 1 重ループに変換できる場合があります。

```
for i := 1 to 100 do a[i] := 0.0;
```

これを**配列の線形化** (linearizaton) と呼びます。線形化はループのオーバヘッドを減らす効果があり、またベクトル命令 (1 つの命令で多数の番地に同じ演算を行なえる命令) を持つプロセサの場合には、ベクトル長を長くするという点でも実行効率を高める効果があります。

線形化が行えるためには、最内側のループにおいて配列の一番最後の次元の添字式が対応する次元の上限から下限まで変化することと、その 1 つ外側のループで 1 つ前の次元の添字式が 1 つずつ変化することが条件となります。ここで最後 (最左側) の次元となっているのは配列が行単位 (column-wise) で配置される言語の場合であり、Fortran のように列単位 (row-wise) で配置される場合には最初 (最右側) から順に調べます。