Modern Compiler Implementation in ML; 第18章

久野　靖*

1999.3.31

1　イントロ

□ プログラムはループで時間を食う→最適化の価値

- でもループって何?
- Webster:　「終了条件が成立するまで繰り返し実行される命令の列」→△

□ ループの定義:　ノードの集合 S。ヘッダノード h を含み、以下の条件:

- S 中のすべてのノードから h へのパスが存在。
- h から S 中のすべてのノードへのパスが存在。
- S の外のノードから S の中のノードへの辺は h への辺のみ。

□ 図 18.1。

- 入口ノード:　先行ノードがループ外のノードであるようなノード。
- 出口ノード:　後続ノードがループ外のノードであるようなノード。
- 出口は複数でもよい。ループは入れ子になってもよい。

1.1　縮約可能

□ 図 18.2a:　ループではない。

□ 図 18.2c:　x が y の唯一の先行ノードであるとき (x, y) を併合→ 18.2a と同じに。

- 18.2a と同じになるようなグラフ→縮約可能でない。
- 18.2a と同じものを含まない→縮約可能→ヘッダノードがある。

□ if-then(-else), while, repeat, for, break のみ→縮約可能なグラフだけができる

- Tiger, Java, goto なしの C →縮約可能

□ 関数型言語→再帰がループ→任意の再帰では縮約可能とは限らない

□ 縮約可能だと何がいい?→解析が効率よくなる。

- でも以下では縮約可能には限定しない。

2　支配節

□ まずループを見つける必要→支配節について知っておくとよい。

- ノード s_0:　開始ノード (先行節がない)。
- s_0 から n までの経路に常に d が含まれる→ d が n を支配する
- すべてのノードは自身を支配

2.1　支配節発見アルゴリズム

□ n の先行節が $p_1 \cdots p_k$ とする。

- d ($\neq n$) が n を支配⇔ d がすべての p_i を支配

□ n を支配する節の集合を $D[n]$ と書くと、

$$D[s_0] = \{s_0\}$$

$$D[n] = \{n\} \cup (\cap\ (p \in pred[n])D[p])$$

- s_0 以外の $D[n]$ は全ノードの集合であるように初期設定。
- 反復法で解ける。トポロジカルソートしておけば効率よく解ける。
- 到達不可能なノードは別扱いすべし。

*筑波大学大学院経営システム科学専攻

2.2 直接支配

□ 定理: 連結グラフで $ddomn$、$edomn \to ndome$ または $edomn$

□ →すべての n は $idom(n)$(直接支配節) をたかだか 1 つ持つ。

- $idom(n) \neq n$
- $idom(n)domn$
- $idom(n)$ は他の d(n の支配節) を支配しない

□ s_0 以外の節はちょうど 1 つの直接支配節を持つ。

2.3 支配木

□ 直接支配関係のみを辺として持つ木→支配木。

- 支配木の辺はグラフの辺と対応しないものも。

□ $hdomn$ であるような n から h へのグラフの辺→帰辺

- 帰辺 1 つにつきループが 1 つある。

2.4 ループ

□ $n \to h$ が帰辺であるような自然ループとは、$hdomx$ かつ x から n への h を通らないパスがあるような x の集合

- h は 1 つ以上の自然ループのヘッダであることも。
- 自然ループでなくても最適化はできる。
- しかし最内側のループから最適化したい。
- ヘッダが共有されているとどっちが内側かわからない。
- よくある解法は、ヘッダを共有するループを併合すること (結果は自然ループではなくなるかも)。

2.5 入れ子のループ

□ A、B がループでヘッダが a、b のとき、$adomb$ であれば B の要素は A の要素に完全に含まれる (B は A の内側ループ)

□ 入れ子関係を表すためループネスト木を作る。

- G の支配木を作る
- すべての自然ループとそのヘッダ h を見つける
- ヘッダ h に対して $loop[h]$ を定める (ヘッダ共有ループの併合)
- ヘッダの支配関係に基づき h を木構造にする

□ ループネスト木の葉→最内側ループ

□ どのループにも含まれていない節を入れるため、ループ外の節は最外側の「仮想ループ」に含まれていることにする。

2.6 プリヘッダ

□ ループ最適化ではループの直前に何かを挿入したい→プリヘッダ p を用意し、p から h への辺を作り、またループ外から h への辺はすべて p への辺につけかえる。

3 ループ不変式

□ ループ内で値が変わらない式は外に出したい (ホイスティング)

- 例によって控え目な見積りが必要

□ $d : t \leftarrow a_1 op a_2$ がループ L について不変であるとは、各 a_i について

- a_i が定数であるか、
- d に到達する a_i の定義がすべてループ外にあるか、
- d に到達する a_i の定義が 1 つだけでありかつループ不変のとき

□ これも反復解法で解ける。

3.1 ホイスティング

□ ホイスティングでプログラムは早くなるが、「正しくない」変形をしてはいけない→図 18.6

□ $d : t \leftarrow a_1 op a_2$ をプリヘッダへ移動してもよい条件:

- d がループの全出口を支配するか、ループ出口で生きていない
- かつ、t の定義がループ内で 1 箇所
- かつ、t がループプリヘッダの出口で生きていない

□ 暗黙の副作用: 演算が例外などを出す場合は手当必要 (練習 18.7)。

□ while を repeat-untli に: 上記の 1 番目の条件のため、while ループには適用できないことが→while を if + repeat-until に変形すれば避けられる。

4 帰納変数

□ i がループ周回ごとに一定数増減し、$j = i \cdot c + d$、というのはよくある。その場合、j も一定数増減に書き換えられる。

- 帰納変数解析→一定増減の変数を見つける。強さ軽減→乗算を加減算に置き換える。
- i は基本帰納変数。j などは i の派生帰納変数。1 つのものから派生した仲間→ファミリ

□ (i, a, b) 記法…$j = a + i \cdot b$ のときこう書く。

□ 帰納変数が常に一定数増減→リニア帰納変数。18.9a の i はリニアではない。また j も一時的に取り残されることが。

4.1 帰納変数の定義

□ 基本帰納変数: ループ L 内での定義が $i \leftarrow i+c$、$i \leftarrow i-c$ (c はループ不変) であるような変数。

□ 派生帰納変数: k が L 内で派生帰納変数とは:

- L 中の k の定義が 1 つだけで $k \leftarrow j \cdot c$、$k \leftarrow j + d$、c、d はループ不変、j は帰納変数であるとき。
- かつ、j が i の派生帰納変数であるなら、
- (a)k の定義に到達する j の定義はループ中にあるものだけで、
- (b) かつ、j の定義から k の定義までの間のいかなるパスにも i の定義がないとき。

□ j が (i, a, b) なら k は $(i, a, b \cdot c)$ または $(i, a + d, c)$。

- 引くときは c をマイナスとして扱う (時として問題があることが…)

□ 除算も $1/c$ 倍と考えてよい。ただし誤差の問題も。整数除算はどうしょうもない。

4.2 演算強さの軽減

□ 多くのマシンで乗算は加減算より遅い。

□ j が (i, a, b) のとき、$i \leftarrow i + c$ の直後に $j' \leftarrow j' + c \cdot b$ を入れ、j の定義を $j \leftarrow j'$ にする。$c \cdot b$ がループ定数ならプリヘッダで計算できる。j' の初期設定も必要。

- j を j' で置き換えるのは、標準のコピー伝播によればよい。
- これで乗算がプリヘッダに移せるが、乗算の遅延を動的隠蔽できるマシンだとあんまり効果がない場合も。

□ 例:

4.3 帰納変数の削除

□ 強さ軽減の結果、いくつかの帰納変数が使われなくなることも。またループ終了判定にのみ使われることも。

□ いずれも削除できる。

4.4 判定の書き換え

□ k が「ほとんど不要」とは、その変数がループ不変式に対する比較のためのみに使われ、同じファミリの不要でない変数がある場合。

□ そのときは比較を適宜書き換えれば不要にできる。

□ 制約: 変形時にきっちり割り切れないことも。またループ不変式の符号が不明で比較方向が分からないことも。

□ 例:

5 配列の範囲検査

□ うーん…

6 ループ展開

□ 本体が小さく、ループ制御の比率が高い場合に。

□ 帰納変数を毎回増やさずまとめて扱うことも。

□ 半端な周回は出口で処理する。