スライス解析の制御依存解析パートにおける支配関係を求めるアルゴリズムの調査と実装

知識工学部　阿部裕介

2012 年 9 月 25 日

1　概要

スライス解析は, コンパイラ技術に由来するソースコードの解析手法で, デバッグ, ソースコード診断, リバースエンジニアリングにおいて重要な役割を担う要素技術である. スライシング基準と呼ばれるユーザが指定したソースコード内のある文のある変数に対し, 実行可能でなるべく小さい, しかも元のプログラムと同じ出力を返すサブプログラムを求める [7]. スライス解析は大雑把に捉えれば, 字句解析・構文解析を経て中間言語表現に落としたソースコードを, さらにフローグラフと呼ばれるグラフ構造に変換した対象について行われ, 以下の 3 パートからなる.

$$\text{スライス解析} = \text{データ依存解析} + \text{制御依存解析}\ (+\ \text{スティッキー依存解析}^{*1}).$$

弊社で開発中のスライス解析モジュールのボトルネックは, 制御依存解析パートの支配関係節ならびに支配木を求める部分であったため, 今回, 支配関係を求めるアルゴリズムについて調査を行い, 新たに採用したアルゴリズムを Haskell で実装し, 実行時間を計測した.

2　フローグラフのノード間の支配関係を求めるアルゴリズム

- Aho-Ullman$\cdots$ 古典的でわかりやすい [1].
- Lengauer-Tarjan$\cdots$ 一般的なコンパイラの制御依存解析にもっともよく使われているらしいが, 実装はやや複雑 [3],[2].
- Harel$\cdots$ フローグラフのノード数 n に対して, 線形時間での処理を可能にした画期的な結果. 実装はとても複雑. [4] は Harel の方法をよりシンプルにしたもの. 支配関係を求めるには, n ノードを少なくとも一度は見なければならず, 計算量的にはこれが最善である.
- Cooper-森 [*2] $\cdots$ Aho-Ullman の改良版で, シンプルで速いのが特長 [5],[6].

Haskell での実装のしやすさ, 計算量, アルゴリズム自体のわかりやすさを鑑みて検討を重ねた結果, Cooper-森アルゴリズムを用いて支配関係を求める部分を書き直すこととした. また Cooper-森アルゴリズムにおい

[*1] 開発中のスライス解析モジュールにおいて, ある指定した文たちを消さずに残す処理をこう呼んでいる.

[*2] この呼称は一般的ではない. 本来このアルゴリズムは森が Cooper らより先に発見して Web 上で公開していたものの, Cooper らがおそらくは森とは独立に発見し、先に論文として発表してしまったらしい. こうした経緯から Cooper-森と呼ぶことにした.

て必要となる, 直接支配節を収めることになる配列 idom に相当するデータ構造は, Containers パッケージの Data.IntMap を用いた.

2.1 Cooper-森アルゴリズムについて

Cooper によれば “Our simple iterative technique is faster than the Lengauer-Tarjan algorithm.” とある [5]. 森も「実際のフローグラフに適用してみると多くの場合 Lengauer-Tarjan よりも高速である」と書いている [6].

計算量は, フローグラフのノードを n としたとき, Aho-Ullman が $O(n^2)$, Lengauer-Tarjan が $O(n \cdot \log(n))$ および改良された経路圧縮版では $O(n \cdot \alpha(n))$ (α は Ackermann 関数 $A(n,n)$ の逆関数で, 極度に遅く増加[*3]) である. Cooper-森は, 病的に作成されたフローグラフでは Lengauer-Tarjan より遅いが, 現実のコードから派生するフローグラフでは, Lengauer-Tarjan より大抵速く, その計算量は, Lengauer-Tarjan 同様, ほぼ線形時間での処理が期待できる.

2.2 Cooper-森アルゴリズムの正当性の証明

以下に Cooper-森アルゴリズムの擬似コードを示す. Cooper と森のアルゴリズムはよく似ているが, 配列 idom の初期化の有無, 更新順序の記述, 補助関数内でのノード比較の仕方が異なる. ここでは森のアルゴリズムに準拠した擬似コードを提示することにした.

Cooper-森アルゴリズム

```
idom = ある深さ優先探索木をつくり, 初期値として各ノード v の親ノードを指す配列. 最終的には各 v の直接支配節がそれぞれ収まる.

while (ある idom[v] に変化がある)
    for (v in V)  /* ノード集合 V は reverse postorder で整列 */
       for (p in Pred(v)) /* Pred(v) は v の predecessor の集合 */
          idom[v] = intersect(p,idom[v])
```

補助関数 intersect

```
intersect(x,y) = while (x != y)
                     if (dfnum(x) > dfnum(y))
                         x = idom[x] /* 更新途上にある配列 idom を参照することに注意 */
                     else (dfnum(x) < dfnum(y))
                         y = idom[y]
                 return x

dfnum(v) = ノード v の depth-first number
```

[*3] 定義より $\alpha(A(4,4)) = 4$ であるが, $A(4,4)$ で既に $A(4,4) = 2^{2^{2^{65536}}} - 3$ という破天荒な大きさの数となり, 現実的な大きさの n では $\alpha(n) \leq 4$ で抑えられる. よって, 経路圧縮版 Lengauer-Tarjan アルゴリズムの計算量は, 事実上線形時間と見なしてよい.

2.2.1 配列 idom の更新順序に関して

配列 idom の更新順序について Cooper[5] は reverse postorder と, 森 [6] は postorder と書いている. この食い違いの理由に (筆者同様) 悩む人がいるかもしれないので自身の見解を述べる. 当初, Cooper も森も同じアルゴリズムなのかと思って読んでいたが, 補助関数内でのノード比較方法が森では depth-first number 順を, Cooper では postorder を用いて全く異なっており, その違いに起因しているように思われる. [*4]

先に提示した擬似コードでは配列 idom の更新順序は森のものとは異なり reverse postorder とした. Cooper-森アルゴリズムは Aho-Ullman アルゴリズムと似ていることは確かであり, その Aho-Ullman アルゴリズムでは効率よく計算するために reverse postorder で更新している. これはもし後退辺がない場合は reverse postorder が topological sort と一致し, すべてのノードの successor が後にあらわれることで更新の影響を効率よく伝播させることができるという理由による.

なお森 [6] では depth-first number 順を深さ優先探索木の reverse postorder と一致としている. ドラゴンブック [1] とタイガーブック [2] で, フローグラフの depth-first number の定義には微妙に食い違いがあり, 本レポートでは [2] に掲載されている depth-first number の定義を採用した. この場合, depth-first number はある深さ優先探索でフローグラフの各ノードを訪れる順番そのままであり, 深さ優先探索木を左から子ノードを追加する規約で作成した際に, 各ノードを preorder で辿ったものと一致する. 一方 [1] での depth-first number は先の規約で作成した深さ優先探索木の reverse postorder に一致する. 森はこちらの見方でアルゴリズムを記述していると思われる. 少なくとも森のアルゴリズムでは配列 idom の更新順序は効率 (何回の更新で収束するか) には影響するが, 正しく直接支配節を計算するというアルゴリズムの正当性には影響しないと考えてよい.

2.2.2 証明の方針と準備

Cooper-森アルゴリズムが正しく直接支配節を計算することを示す[*5] には, 以下の 1-4 が言えればよい. ある良い性質を持つ適切な初期値 idom[v] たちから出発し, それぞれの dfnum(idom[v]) は更新毎に減少していくものの, 2 で存在が保証されている更新不変配列を飛び越えたりしないこと (3 に相当), 最後に収束した段階では 4 で idom[v] が v を支配することから, 任意の v について idom[v]=idom(v) が言える.

1. アルゴリズムの停止性 (これは 3 で配列 idom の各要素の dfnum(idom[v]) が更新の度に減少していくことから言える).
2. 配列 idom の初期値として各 v の直接支配節 idom(v) を入れた場合, 配列 idom は更新で不変である.
3. 初期状態の配列 idom_init は次の二つの良い性質を持っている.
   - root ではない任意の v について, idom_init[v] は v の真の祖先である.
   - root ではない任意の v とその任意の predecessor p について, 以下の系列を考える.

$$\text{idom\_init[v]}, \text{idom\_init[idom\_init[v]]}, \cdots, \text{root}$$

$$\text{p}, \text{idom\_init[p]}, \text{idom\_init[idom\_[p]]}, \cdots, \text{root}$$

このとき両系列のどちらにも v の直接支配節 idom(v) がどこかにあらわれる. とくに一つ目の系

[*4] しかしながら, 森 [2] の記述「後行順に訪問する」が「逆後行順に訪問する」の誤植の可能性もあるかもしれない.
[*5] Cooper[5], 森 [6] のどちらにも正当化の証明の詳細は書かれていないので, 本レポートではなるべく詳しい記述を試みた.

列上に idom(v) があらわれることから任意の v について,

$$\text{idom\_init[v]} \geq \text{idom(v)}.$$

ノードの比較は depth-first number の大小で比較している.

この性質が更新によって保たれることが帰納法で言える. ここで擬似コード最終行での 1 回の更新前後での配列 idom の変化を, 更新前を idom_old, 更新後を idom_new で表すことにする. idom_old が idom_init が満たす先の二性質を満たすならば idom_new もこれらの性質を満たす. したがって任意の v について以下が成り立つことになる.

$$\text{idom\_old[v]} \geq \text{idom\_new[v]} \geq \text{idom(v)}.$$

4. 擬似コードで定義した初期状態から出発した更新途上にある配列 idom において, ある v の idom[v] が v を支配していないならば, idom[v] はいずれより小さい値に更新される.

Proposition 2.2.1　root ではない任意の v について, idom[v] が v の真の祖先であるとき, 任意の x, y について intersect(x, y) は x と y の共通祖先である.

Proof. 仮定より関数 intersect は祖先関係を保持したまま, 深さ優先探索木上を root 側に登っていき, depth-first number が一致したノードを返すので上の命題が成り立つ. なお, Cooper-森アルゴリズムでの配列 idom の初期状態は上の命題の仮定の条件を満たす.

Proposition 2.2.2　v の predecessor の集合 Pred(v)=$\{p_1, \cdots, p_n\}$ について, v の直接支配節 idom(v) は $p_1, \cdots, p_n$ の共通祖先である.

Proof. ある p ∈Pred(v) が idom(v) の子孫ではないとする. このとき root から p へ至る idom(v) を経由しない深さ優先探索木上の経路が存在しなくてはならないが, これは root から idom(v) を経由しないで v に至る経路が存在することを意味し, 矛盾する.

2.2.3　2 の証明

仮定より, 任意の v について idom[v]=idom(v) が成り立っていることに注意する. 任意の p ∈Pred(v) について, intersect(p, idom[v]) = idom[v] が言えればよい. Proposition 2.2.2 より idom[v] は p の祖先であり idom[v] ≤ p. よって等号不成立ならば intersect(p, idom[v]) = intersect(idom[p], idom[v]) であるが, ここで

$$\text{idom[p], idom(idom[p])}, \cdots, \text{root}$$

という系列を考えると, それぞれの depth-first number はどこかで idom[v] に一致する. なぜなら idom(v) は p を支配するので上の系列のどこかにあらわれる. したがって, intersect(p, idom[v]) = idom[v].

2.2.4　3 の証明

idom_old が先の二性質を満たすと仮定したとき, idom_new も二性質を満たすことが言えればよい. idom_old[v] ≥ idom_new[v] は, Proposition 2.2.1 より idom_new[v] が idom_old[v] と Pred(v) の共通祖先であることから出る. これより一番目の性質である, root ではない任意の v について, idom_new[v] は v の真の祖先であることが言えた.

ある v の更新 (idom_new[v] ← intersect(p, idom_old[v]) の結果としての idom_new を考え, これが二番目の性質を満たすことを示す.

以下の系列を考える.

$$\text{idom\_old[v]}, \text{idom\_old[idom\_old[v]]}, \cdots, \text{root}$$

$$\text{p}, \text{idom\_old[p]}, \text{idom\_old[idom\_old[p]]}, \cdots, \text{root}$$

帰納法の仮定により, 両系列のどちらにも idom(v) が共通にあらわれている. したがって

$$\text{idom\_new[v]} = \text{intersect(p, idom\_old[v])} \geq \text{idom(v)}$$

でなければならない.

ところで idom_new[v] は, ここでの p の旧系列のどこかに登場していたはずである (例えばであるが, idom_new[v] = idom_old[p] のように). さらにいま v 以外の要素 w については idom_new[w] = idom_old[w] であるから, 系列

$$\text{idom\_new[v]}, \text{idom\_new[idom\_new[v]]}, \cdots, \text{root}$$

は下の系列

$$\text{p}, \text{idom\_new[p]}(= \text{idom\_old[p]}), \text{idom\_new[idom\_new[p]]}(= \text{idom\_old[idom\_old[p]]}), \cdots, \text{root}$$

の部分系列になっていることになり, 結果として両系列のどちらにも idom(v) が共通にあらわれていることになる. なおここでの p は更新時の特定の p であったが, 他の任意の v の predecessor p' についても帰納法の仮定より p' の新系列中に idom(v) があらわれることが言える.

2.2.5 4 の証明

3 の結果より, idom[v] ≥ idom(v) としてよい (さらに idom[v] は Pred(v) の要素たちの共通祖先としてもよい). よって idom(v) より depth-first number が大きい idom[v] は, いずれ更新によって, depth-first number がより小さい値に更新されることが言えればよい.

idom[v] > idom(v) とする. もしすべての v の predecessor を idom[v] が支配するならば, idom[v] が v を支配することになってしまうので, ある p ∈ Pred(v) が存在して, この p は idom[v] に支配されていない. すなわち idom(v) から idom[v] を経由しないで p に至る経路が存在する. この経路は深さ優先探索木上の辺だけでできていることはできず, idom[v] を祖先とするこの経路上のノード q で, Pred(q) の要素たちの共通祖先の depth-first number が idom[v] より小さいものが存在する. これより idom[v] は, いずれ更新でいまの idom[v] の depth-first number より小さい値に更新されることになる.

図 1 ではこの証明の状況に対応するフローグラフの例を挙げた. ノードの番号は depth-first number を, 実線は深さ優先探索木上の辺を, 破線はフローグラフ上の辺をあらわす.

3 計測結果

domtree1 が Aho-Ullman アルゴリズムを用いて支配木作成までを行う (かつてスライス解析モジュールで用いていた) プログラム, domtree2 が今回新たに実装した Cooper-森アルゴリズムを用いて支配木作成を行うプログラムである. 入力としてのフローグラフには 2246 ノードを持つサンプルを使用した. [*6]

[*6] 実験環境は, ghc-7.4.2, メモリ 4GB, CPU として Intel(R) Core(TM)2 Quad CPU Q9650 @3GHz を使用した.

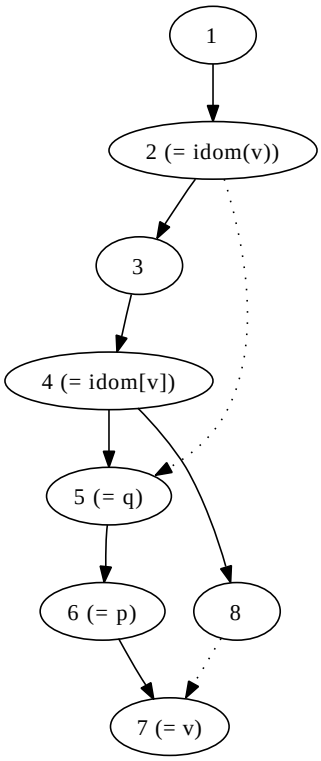

図 1　4 の証明に対応するフローグラフの例

domtree1 では 23 分ほどかかっていた処理時間が, domtree2 では約 2 秒になっており, アルゴリズムとデータ構造の置き換えによる効果が現れていることがわかる. [*7]

[*7] ただし Haskell のデータ構造の違いによる効果も相当にあると思われるので, アルゴリズムの良し悪しの単純な比較ではないことに重ねて注意されたい.

domtree1

```
$ time ./domtree1 +RTS -s
   1,446,770,080 bytes allocated in the heap
     182,702,248 bytes copied during GC
      22,064,200 bytes maximum residency (13 sample(s))
         328,984 bytes maximum slop
              52 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2773 colls,     0 par    0.09s    0.16s     0.0001s    0.0012s
  Gen  1        13 colls,     0 par    0.10s    0.12s     0.0091s    0.0331s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time  1379.51s  (1379.88s elapsed)
  GC      time    0.19s  (  0.28s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time  1379.70s  (1380.16s elapsed)

  %GC     time       0.0%  (0.0% elapsed)

  Alloc rate    1,048,756 bytes per MUT second

  Productivity 100.0% of total user, 100.0% of total elapsed


real 23m0.167s
user 22m59.710s
sys 0m0.260s
```

domtree2

```
$ time ./domtree2 +RTS -s
   1,831,445,824 bytes allocated in the heap
      98,682,376 bytes copied during GC
       4,593,880 bytes maximum residency (26 sample(s))
         101,976 bytes maximum slop
              14 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3370 colls,     0 par    0.05s    0.09s     0.0000s    0.0004s
  Gen  1        26 colls,     0 par    0.03s    0.04s     0.0014s    0.0068s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    1.83s  (  1.94s elapsed)
  GC      time    0.08s  (  0.13s elapsed)
  RP      time    0.00s  (  0.00s elapsed)
  PROF    time    0.00s  (  0.00s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.91s  (  2.07s elapsed)

  %GC     time       4.2%  (6.2% elapsed)

  Alloc rate    1,000,790,067 bytes per MUT second

  Productivity  95.8% of total user, 88.5% of total elapsed


real 0m2.070s
user 0m1.910s
sys 0m0.020s
```

参考文献

[1] A. V. Aho, Ravi Sethi, J. D. Ullman. *Compilers Principles, Techniques, and Tools*, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML*, 1997.

[3] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph, 1979.

[4] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time, 1999.

[5] Keith D. Cooper, Timothy J. Harvey and Ken Kennedy. A Simple, Fast Dominance Algorithm, 2001.

[6] 森公一郎. フローグラフの支配関係木を求めるアルゴリズムについての覚書, http://www005.upp.so-net.ne.jp/kmori/dominator/Dominator.html, 2002.

[7] 溝渕裕司, 中谷俊晴, 佐々政孝. コンパイラ・インフラストラクチャを用いた静的プログラムスライシングツール, 2003.