依存関係の局所性を利用した プログラム依存グラフの効率的な構築法

大畑文明　西松顯　井上克郎

大阪大学大学院基礎工学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3

プログラム依存グラフ (PDG) とはプログラム文間の依存関係を表す有向グラフであり, プログラムスライスに利用される. これまで提案されている多くの PDG 構築法は, スライスの正確性向上を目標としてきた. しかし, 大規模ソフトウェアにスライスを適用する場合においては, その正確性だけでなく抽出の効率を考慮する必要がある. 本研究では, プログラム文間に存在する依存関係の局所性に着目し, 依存情報が文単位で保持される従来手法に対し, より大きな粒度で保持させることで効率向上を行ない, 正確性にも留意した PDG 構築手法を提案する.

Efficient Construction Method of Program Dependence Graph Using Dependency Locality

Fumiaki Ohata, Akira Nishimatsu and Katsuro Inoue

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan

In this paper, we propose an efficient constructing algorithm of program dependence graph. Program dependence graph(PDG) is used for calculating program slice. To construct PDG, we need dependency analysis. In traditional dependency analyses, dependency information is held by each statement in the source program. In our algorithm, dependency information is shared by several statements. In order not to degrade the accuracy of the resulting slice, we use dependency locality with statements.

1 まえがき

プログラムのデバッグ支援に有効な方法としてプログラムスライスがある．その有効性は我々が行なった実験で確認されており [6], 現在では, デバッグ支援だけでなくテスト, 保守, プログラム合成, プログラム理解などにも利用されている．

プログラムスライス [9] とは, プログラム P 中のある文 s に対して, s で参照しているある変数 v の内容に影響を与えうる文の集合 Q のことである．スライス抽出の方法の 1 つに, プログラム文間の依存関係を表す有向グラフであるプログラム依存グラフ (PDG) を利用する手法がある．

しかし, プログラミング言語の高級化に伴い依存関係解析アルゴリズムは複雑化し, 大規模ソフトウェアに対する PDG 構築には多くの時間コスト, 空間コストがかかるようになった．例えば, 約 28000 行の C プログラムの PDG 構築に約 2 時間を要するという報告がされている [10]．そこで, スライス (PDG) の正確性と依存関係解析に要するコストとのトレードオフを考慮する事が必要になってきた [2]．ここでいう正確性とは, スライスに含むべきものは含まれているという前提において, スライスに含まれるべきでない文をどれだけ排除できるかの程度である．正確性の向上によりスライスサイズは減少し, 正確性の低下によりスライスサイズは増加する．

本研究では, プログラム文間に存在する依存関係の局所性に着目し, 依存情報が文単位で保持される従来手法に対し, より大きな粒度 (連続文, ブロック, 繰り返し文 等) で依存情報を保持させることで時間コスト, 空間コストを削減する PDG 構築手法を提案する．

以降, 2. ではプログラムスライスについて述べる．3. では従来手法の問題点を述べ, 提案を行なう．4. では節点集約について述べる．5. では提案手法の評価を行なう．6. では関連研究について述べる．最後に, 7. でまとめと今後の課題について述べる．

2 プログラムスライス

スライス計算の手順は [4] に示されている．その過程を図 1 に示し, 以下各過程を簡単に述べる．

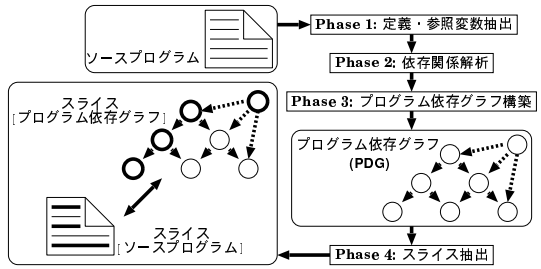


図 1: スライス計算過程

Phase 1: 定義, 参照変数 抽出

プログラムの各文で定義, 参照される変数を抽出する．図 2 にサンプルプログラムおよび各文の定義, 参照変数を示す．

```
1:   b = 5;
2:   a = b + b;
3:   if(a > 0)
     {
4:     c = a;
5:     d = b;
     }
```

文	定義	参照
1	b	
2	a	b
3		a
4	c	a
5	d	b

図 2: サンプルプログラム および 各文の定義, 参照変数

Phase 2: 依存関係解析

プログラム文間のデータ依存関係, 制御依存関係を計算する．文 s から文 t に対して変数 w に関するデータ依存関係 **DD**(s, w, t) があるとは, ある変数 w が存在して, 文 s における変数 w の定義が変数 w を参照している文 t に到達する場合をいう．文 s から文 t に対して制御依存関係 **CD**(s, t) があるとは, 文 s が分岐命令で文 t がその分岐文内に直接含まれている場合, あるいは, 文 s がループ命令で文 t がそのループ文内に直接含まれている場合をいう．表 1 に図 2 におけるデータ依存関係, 制御依存関係を示す．

表 1: 図 2 のデータ依存関係, 制御依存関係

データ依存関係	DD(1, b, 2), DD(2, a, 3) DD(1, b, 5), DD(2, a, 4)
制御依存関係	CD(3, 4), CD(3, 5)

Phase 3: プログラム依存グラフ 構築

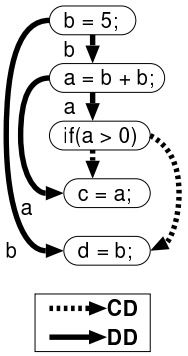


図 3: 図 2 の PDG

依存関係解析結果を元にプログラム依存グラフを構築する．**プログラム依存グラフ (Program Dependence Graph, 略して PDG)** とは, プログラムに存在する文を節点, 文間のデータ依存関係, 制御依存関係を辺で表現した有向グラフである．データ依存辺には実際に依存している変数名がその名前として与えられる．辺が節点 V_s から節点 V_t に引かれている場合, 節点 V_t が節点 V_s に依存していることになる．図 3 に図 2 の PDG を示す．

Phase 4: スライス 抽出

スライスを抽出する．スライス抽出の出発点を**スライス基準**といい, 文 s の変数 x であれば**スライス基準** (s, x) と表す．スライス基準 (s, x) のスライスは, s に対応した PDG 節点 V_s から逆方向に制御依存辺および x に関するデータ依存辺を経て推移的に到達可能な節点集合に対応する文となる．図 4 に図 2 のスライス基準 (文 5, b) のスライス (太枠部) を示す．

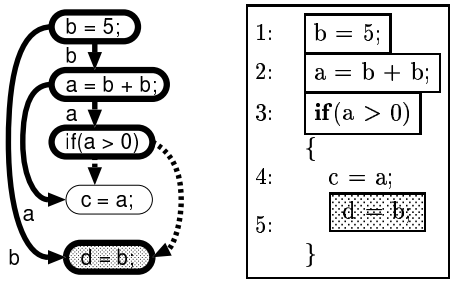


図 4: 図 2 のスライス基準 (文 5, b) のスライス

3 提案

3.1 従来手法の問題点

通常 Phase 2: 依存関係解析 で手間がかかるため, 依存関係解析アルゴリズムを計算量の少ないものに変更することでコストを削減し, 正確性とのトレードオフを考えていた [2, 3]. しかし, 異なる依存関係解析アルゴリズムにより構築された PDG の相互利用は難しく, 最悪の場合, アルゴリズム変更のために一から再解析して PDG を構築しなければならない. また, 異なる依存関係解析アルゴリズム間での PDG の相互利用を考慮するには, アルゴリズム変更に対応させるために PDG 構造が複雑になる.

3.2 提案する手法

従来のコスト削減手法が依存関係解析アルゴリズムに依存していたのに対し, 今回我々は**節点集約**を用いたコスト削減手法を提案する. 節点集約とは, プログラム文と PDG 節点が 1 対 1 で対応していたものを多対 1 にするもので, 複数の文を 1 つの節点で表現する. この節点集約を用いることにより,

- 情報の共有による, 情報量 (空間コスト) 削減
- 計算対象の削減による, 計算量 (時間コスト) 削減

が可能となる. また, 既存の依存関係解析アルゴリズムとの共用,PDG 構築後の正確性の変更も容易である.

4 節点集約

4.1 節点集約

今回提案する節点集約は, 3.2 節で述べたように, 依存関係解析アルゴリズムとの共用,PDG 構築後の正確性の変更 を容易にするため, 以下の方針をとる.

- 依存関係解析と独立させるため, 集約を依存関係解析前に行なう
- 節点集約 (分解) を局所的解析のみで実現するため, 集約対象を連続文またはプログラム構造のまとまりに限定する

集約を行なう上で最も重要なことは, 集約によりスライスに含まれるべき文が含まれなくなるのを防ぐことである. そのため, 集約の際には集約対象節点に関するすべての依存関係を集約節点に置き換える必要がある.

具体的な節点集約の例を図 5 に示す. 文 5〜文 8 の**if** 文全体を集約した結果, 文 5〜文 8 に関する依存関係は集約節点にまとめられ (集約対象となった文間の依存関係は取り除かれる), PDG の節点数 (10→7) および辺数 (13→6) が減少する.

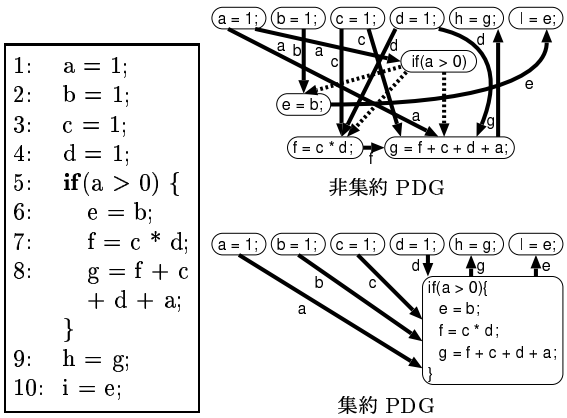


図 5: 節点集約例

次に, 節点集約とスライスの関係について述べる. 以降, 非集約 PDG によるスライスを非集約スライス, 集約 PDG によるスライスを集約スライスと呼ぶ.

例として, 図 5 のスライス基準 (文 9, g) に対するスライス抽出を非集約 PDG(図 6), 文 5〜文 8 を集約した PDG(図 7), 文 7〜文 8 を集約した PDG(図 8) に対して行う.

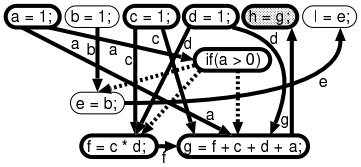


図 6: 図 5 のスライス基準 (文 9, g) のスライス

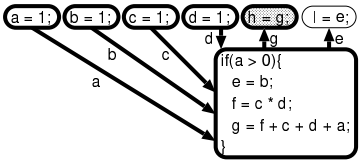


図 7: 図 5 のスライス基準 (文 9, g) のスライス

図 7 では文 2, 文 6 が集約スライスに含まれスライスが不正確になっているが, 図 8 では非集約スライスと同じスライスが抽出されている. 集約節点内の文は依存情報を共有するため, 図 7 のように単純に連続した文を集約するだけではスライスが著しく不正確になる可能性があり, 集約の可否を決定する基準となるものが必要となる.

そこで本研究では, 集約対象文が依存している文が共通していること (**依存関係の局所性**) を集約判定基準とした. 図 8 では, 文 7, 文 8 が依存している文 (依存関係) が共通しているために正確性の低下が抑制されている.

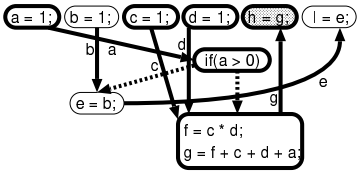


図 8: 図 5 のスライス基準 (文 9, g) のスライス

4.2 依存関係の局所性

集約判定に依存関係の局所性を用いることは 4.1 節で述べたが, 本節では, 局所性の把握を容易にするため局所性を有する依存関係を以下の 3 つに分類する. 以降, それぞれについて述べる.

(1) 共通参照変数に関する依存関係
(2) 定義+参照変数に関する依存関係
(3) 支配変数に関する依存関係

(1) 共通参照変数に関する依存関係

共通参照変数とは, 集約対象文 s_1,s_2 において $^{\exists}t$[exist(DD(t, a, s_1)) & exist(DD(t, a, s_2))][¶1]を満たす変数 a をいう. 図 9 に, 文 7, 文 8 の共通参照変数に関する依存グラフを示す.

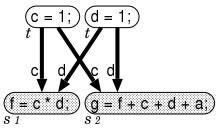


図 9: 共通参照変数に関する依存グラフ

(2) 定義+参照変数に関する依存関係

定義+参照変数とは, 集約対象文 s_1,s_2 において exist(DD(s_1, a, s_2)) を満たす変数 a をいう. 定義+参照変数が存在する場合, $^{\forall}t^{\forall}v$[exist(DD(t, v, s_1)) → im-exist(DD(t, v, s_2))][¶2]が成り立つ. 図 10 に, 文 7, 文 8 の定義+参照変数に関する依存グラフを示す. 細破線が示すデータ依存関係は, 定義+参照変数 f の存在による間接的な依存関係である.

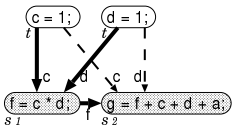


図 10: 定義+参照変数に関する依存グラフ

(3) 支配変数に関する依存関係

支配変数とは, 条件節 s_0 の参照変数 a をいう. 集約対象文 s_1,s_2 が分岐節, 繰り返し節に存在する場合, exist(CD(s_0, s_1)) & exist(CD(s_0, s_2)) であるから, $^{\forall}t$[exist(DD(t, a, s_0)) → im-exist(DD(t, a, s_1)) & im-exist(DD(t, a, s_2))] が成り立つ. 図 11 に, 文 7, 文 8 の支配変数に関する依存グラフを示す. 細破線が示すデータ依存関係は, 支配変数 a の存在による間接的な依存関係である.

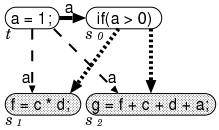
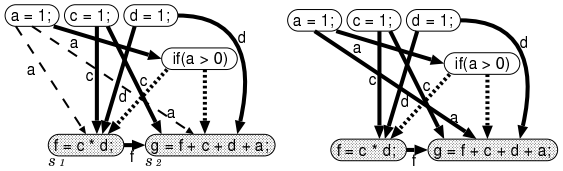


図 11: 支配変数に関する依存グラフ

4.3 集約の判定

本節では, 集約対象文の集約可否の判定方法について述べる.

連続文 s_1,s_2 の集約においては, PDG に存在する 2 文に関する依存グラフが, 2 文の共通参照変数, 定義+参照変数, 支配変数に関する依存グラフをまとめたグラフの部分グラフである場合に集約可能となる. 図 12(a) に図 9, 図 10, 図 11 をまとめたグラフ, 図 12(b) に PDG に存在する文 7, 文 8 に関する部分依存グラフを示す. 図 12(b) が図 12(a) の部分グラフであることから, 文 7, 文 8 は集約可能と判定される.



(a) 依存関係の局所性に基づく 依存グラフ　(b) PDG の部分依存グラフ

図 12: 集約の判定

4.4 集約アルゴリズム

本節では, PDG を利用しない集約の判定方法 および 集約可能と判定した場合の節点集約について述べる.

集約は依存関係解析前に行なうため, PDG(依存関係情報) を利用した判定を行なうことはできない. しかし, 集約判定に必要となる要素は集約対象文の定義変数, 参照変数, 支配変数であり, また連続した文のみを集約対象としているため, 判定は変数集合に対する演算に置き換えることができる. 4.3 節では連続文における集約の判定を示したが, 集約の判定および節点集約はプログラム構造に依存するため,

- 連続文 (AlgorithmSerial)
- ブロック (AlgorithmBlock)
- 分岐文 (AlgorithmSelection)
- 分岐文 (else なし)(AlgorithmSelection')
- 繰り返し文 (AlgorithmIteration)

に分けてそのアルゴリズムを定義した. 各アルゴリズムは前提, 入力, 判定, 集約, 出力, 関連に分けられており, 以下ではAlgorithmSelection' を

¶1 exist(A) は, 依存関係 A が存在することを表す.

¶2 im-exist(B) は, 2 つの依存関係によって間接的な依存関係 B が存在することを表す. 例えば, exist(DD(s, a, t)) & exist(DD(t, b, u)) から im-exist(DD(s, a, u)) が導き出される.

例に用いて述べる. その他詳細は付録として添付した.

諸定義

S:	節点 (集約節点も含む)
CTL(S):	S を支配する変数集合
USE(S):	S で参照される変数集合
DEF(S):	S で必ず定義する変数集合
poDEF(S):	S で定義される可能性のある変数集合 (DEF(S) ⊂ poDEF(S))
limit:	集約許容値 (集約の程度を決定する値. limit = 0 で, 集約対象となる文が依存している文が一致している場合にのみ集約される (図 8 参照))

アルゴリズム

前提: then 節は 1 節点 (つまり, 単一文節点または集約節点) でなければならない[¶3]
入力: 条件節節点 A, then 節節点 B
出力: 集約節点 S(集約可能と判定した場合)
判定: USE(A),USE(B) のいずれかが空集合の場合は無条件で集約可能と判定,
USE(A),USE(B) のいずれも空集合でない場合は $|\ \mathrm{USE(B)} - \mathrm{USE(A)}\ | \leq \mathrm{limit}$ のとき集約可能と判定[¶4]
集約: USE(S) = USE(A)∪USE(B)
DEF(S) = ∅, poDEF(S) = poDEF(B)
関連: 条件節の参照変数は分岐節の支配変数

4.5 支配表

提案する集約アルゴリズムにより, スライスの正確性低下を抑えた集約を行なうことができる. しかし, 集約節点で参照のみされる変数をスライス基準とした場合, 得られるスライスの正確性が著しく低下する可能性がある. 集約節点では制御依存に関する情報は失われるため, 定義変数は当然であるが参照変数に関しても, すべての参照変数に依存していると解釈される. 例えば, 図 13 では集約スライスのサイズが 2→8 に増大している.

この問題の解決方法として,

- スライス基準となった集約節点のみを再解析する
- 集約前の節点情報を保持しておく

などが考えられるが, 本研究では, 各集約節点に各参照変数がどの参照変数に依存しているか (集約前, どの変数に支配されていたか) を記憶する**支配表 (control table)** を用いた. 支配表は n 行 n 列の表であり, n は各集約節点の参照変数の数である. この方法を用いることでコスト増加を最小限に抑えることができる[¶5]. 支配表は集約時に更新され, 集約節点の参照のみされる変数 a がスライス基準となった場合に, 支配表から変数 a が真に依存している参照変数を抽出することができる. 図 14 は, 図 13 の支配表およびそれに基づき抽出された集約節点の変数 a のスライスである. 集約節点の変数 a はそれ自身にのみ依存していることが支配表から導き出され, スライスサイズは 8→5 に減少した.

[¶3]本アルゴリズムでは, 入力節点は単一文節点または集約節点でなければならない.

[¶4]各プログラム構造に対して一般式 1 つでは困難なため (著しく正確性が低下するもの, 集約が中途で終るものがある), 入力節点の変数集合の状態で場合分けを行なっている.

[¶5]支配表は集約節点でのみ用いられ, 表作成による空間コスト増加は, 節点削除によるコスト減少と比べて十分に小さい.

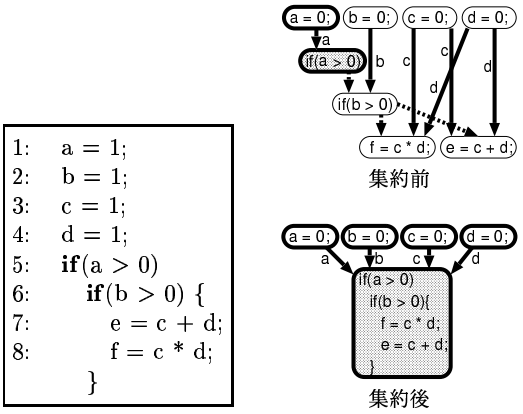


図 13: 集約節点からのスライス

被支配 \ 支配	a	b	c	d
a	○	○		
b		○	○	○
c			○	
d				○

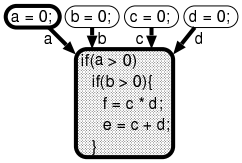


図 14: 支配表と集約節点からのスライス

5 評価・考察

5.1 実験

実装は我々が開発したシステム [7] に対する機能追加の形で行なった. [7] は Pascal のサブセットを対象言語としポインタ, レコード型は取り扱っていない. また, アルゴリズムは [8] による. 比較実験は, 以下の 4 種類 PDG を対象に行なった.

N: 集約なし (従来手法)
$\mathbf{L}_0$: 集約あり (集約許容値 limit = 0)
$\mathbf{L}_1$: 集約あり (limit = 1)
$\mathbf{L}_2$: 集約あり (limit = 2)

テスト対象プログラムを表 2 に, PDG 節点数を表 3 に, PDG 辺数を表 4 に, PDG 構築時間 (集約プロセスを含む) を表 5 に, 平均スライスサイズを表 6 に示す.

表 2: テストプログラム

	行	手続き	内容
P_1	249	14	π の計算
P_2	449	30	小プログラムの集合
P_3	429	18	酒屋問題
P_4	434	18	酒屋問題

表 3: PDG 節点数 [個]

	集約なし	集約あり		
		limit:0	limit:1	limit:2
P_1	128	104	96	86
		(−18.8%)	(−25.0%)	(−32.8%)
P_2	243	199	187	177
		(−18.1%)	(−23.0%)	(−27.2%)
P_3	211	166	153	136
		(−21.3%)	(−27.5%)	(−35.5%)
P_4	226	157	141	124
		(−30.5%)	(−37.6%)	(−45.1%)

表 4: PDG 辺数 [本]

	集約なし	集約あり		
		limit:0	limit:1	limit:2
P_1	525	443	433	394
		(−15.6%)	(−21.2%)	(−25.0%)
P_2	1092	980	951	912
		(−10.3%)	(−12.9%)	(−16.5%)
P_3	1487	1387	1336	1290
		(−6.7%)	(−10.2%)	(−13.2%)
P_4	1823	1662	1590	1525
		(−8.8%)	(−12.8%)	(−16.3%)

表 5: 解析時間 [s]

	集約なし	集約あり		
		limit:0	limit:1	limit:2
P_1	0.05	0.04	0.03	0.03
		(−20.0%)	(−40.0%)	(−40.0%)
P_2	0.18	0.15	0.13	0.13
		(−16.7%)	(−27.8%)	(−27.8%)
P_3	0.2	0.18	0.17	0.17
		(−10.0%)	(−15.0%)	(−15.0%)
P_4	0.32	0.26	0.24	0.23
		(−18.8%)	(−25.0%)	(−28.0%)

PentiumII-300MHz-128MB(Linux)

表 6: 平均スライスサイズ [文]

	集約なし	集約あり		
		limit:0	limit:1	limit:2
P_2	69	70.67	70.67	77.22
		(+2.42%)	(+2.42%)	(+12.64%)
P_3	143.22	145	147.44	148.28
		(+1.24%)	(+2.95%)	(+3.53%)
P_4	168.67	170.83	170.83	170.83
		(+1.28%)	(+1.28%)	(+1.28%)

5.2 考察

空間コスト

PDG 節点数 (表 3) は 20〜40%, PDG 辺数 (表 4) は 10〜20%削減された. 節点と比べて辺の削減量が少ないが, これは集約の影響を受けない辺 (関数間の依存関係を表す辺 等) が存在するためである.

時間コスト

解析時間 (表 5) は 20〜40%削減された. L_0〜L_1 と比べて, L_1〜L_2 ではさらに集約が行なわれている (表 3, 表 4 参照) にも関わらず解析時間の減少が確認されなかった. この理由として, 以下の 2 点が考えられる.

- 依存関係解析は 関数内解析, 関数間解析 に分けられるが, 集約により時間コスト削減が得られるのは主に関数内解析のため, 集約がすすむことで関数内解析時間が減少し, 解析時間に占める関数内解析の割合が関数間解析より下回った
- 集約により時間コストが削減されるのは,
  - 集約により情報が共有され (例えば, 同じ依存情報を持っている節点が集約されると情報の絶対量は半減する), 解析に必要なデータ量が減る
  - 節点数の減少で, 解析に必要な計算回数が減る

  の 2 点によるものであるが, 過剰な集約は情報共有の効果を減少させる

集約プロセス自身にかかる時間コストに関して, 集約対象は連続した文に限定しているため, 集約プロセスはソースファイル先頭からの 1 回の走査で終了する. 実際, 集約に要する時間はいずれのプログラムに対しても 10[ms] 以下であった.

正確性

非集約スライスと比較して, 集約スライスのサイズは多少大きくなる. 相互に依存する文のみ集約した場合は同じであるが, 本手法はその条件を緩め同じ依存関係を持つ文を集約しているためである. 図 15 はその例を示したものであるが, 集約によりスライス基準 ('**printf**("circ:%d\n", circ)', circ) に対するスライスサイズが大きくなっているのが分かる. しかし, 同じ依存関係を持つ節点を集約しているため, スライスの正確性が大きく低下することはなく, スライスサイズの増加は集約節点でのみ生じる. そのため, 表 6 に示すように L_0 では 1〜2%(さらに, L_1 でも 1〜3%) 程度で抑えられているのが分かる. さらに, 非集約スライスのサイズが大きいものほど集約スライスのサイズ増加が少なくなる傾向があることを確認した.

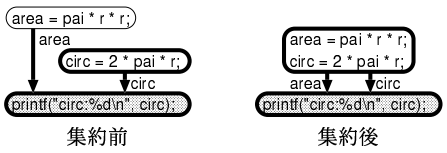


図 15: 集約における問題

P_2 の L_2 でスライスが大きく不正確になったのは, 集約によりスライスサイズが 6→134 となったものが存在したためである. それを除き再計算すると +2.35%となった. このようなスライスサイズの大幅な増加は, 集約許容値 limit を 1 以上にする場合避けられない問題であるが, 今回の実験ではこのスライスでのみ生じた.

6 関連研究

節点集約を用いたコスト削減手法としては, [10] で適用されているものがある.

[10] では, 相互に依存する節点のみ集約するため正確性の低下はない. コスト効率に関しては, 対象

言語, 依存解析アルゴリズム, テストプログラムが異なるため直接の比較はできないが, より大規模なプログラムに対して集約の効果が高いことを確認している. しかし, 集約は依存関係解析フェーズの内部で行なわれており, 異なる依存関係解析アルゴリズム間での PDG の共有は考慮されていない.

7 まとめと今後の課題

依存関係の局所性を利用した節点集約による, 正確性の低下を抑えたコスト削減手法を提案し, その有効性を実験により評価した. Pascal サブセット言語の 500 行程度のプログラムに対し, スライスが 2%程度増加したものの, 時間コスト 25〜35%, 空間コスト 20〜30%の削減が見られた. また, 集約なしのスライスサイズが大きいものほど集約による正確性の低下が少なくなる傾向があることを確認した. 今回定義したアルゴリズムは, 条件節での変数定義 (副作用のある条件節), ポインタ変数を扱っていないが, それらへの適応も可能である. 本手法は依存関係解析の対象である節点を集約するため, より複雑な依存関係解析が必要となる高級言語, 大規模プログラムではさらに有効である.

また, 非集約スライスと集約スライスには次のような違いがある.

- **非集約スライス**... 変数 a を定義する文 s〜変数 a を参照する文 t の論理的関係 (依存関係) に基づく
- **集約スライス**... 論理的関係 + 変数 a を参照する文 s〜変数 a を参照する文 t の意味的関係 (依存関係の局所性) に基づく

非集約スライスと比較して集約スライスのサイズが大きくなる (正確性が低下する) のは, 意味的関係に基づくスライスを含むためである. このことから, 本手法で得られる集約スライスがプログラム理解に有効であると考えている.

今後の課題としては,

- ポインタを含む言語への実装
- 大規模プログラムに対する評価
- プログラム理解に対する有効性の評価

を考えている.

謝辞 本研究は, 一部文部省科学研究費補助金特定領域研究 (A)(2)(課題番号:10139223) の補助を受けている.

参考文献

[1] Aho, A.V., Sethi, S. and Ullman, J.D. "Compilers : Principles, Techniques, and Tools", Addison-Weseley, 1986.

[2] Atkison, D. C. and Griswold, W. G. "The Design of Whole-Program Analysis Tools". In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, pages 16–27, 1996.

[3] Fiutem, R. , Tonella, P. , Antoniol, G. and Merlo, E. "Variable Precision Reaching Definitions Analysis for Software Maintenance". In *Proceedings of the Euromicro Working Conf. on Soft. Maintenance and Reengineering*, pages 60–67, Berlin, Germany, 1997.

[4] Horwitz, S. and Reps, T. "The Use of Program Dependence Graphs in Software Engineering". In *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, pages 392–411, 1992.

[5] Horwitz, S., Reps, T., and Binkley, D. "Interprocedural slicing using dependence graphs". In *ACM Transactions on Programming Languages and Systems 12, 1*, pages 26–60, 1990.

[6] 西江, 神谷, 楠本, 井上 "プログラムスライスに基づくデバッグ支援ツールの実験的評価", ソフトウェアシンポジウム 97 予稿集, pages 142–147, 1997.

[7] 佐藤, 飯田, 井上 "プログラムの依存解析に基づくデバッグ支援ツールの試作", 情報処理学会論文誌, Vol. 37, No.4, pages 536–545, 1996.

[8] 植田, 練, 井上, 鳥居 "再帰を含むプログラムのスライス計算法", 電子情報通信学会論文誌, Vol. J78-D-I, No.1, pages 11–22, 1995.

[9] Weiser, M. "Program Slicing". In *Proceedings of the 5th International Conference on Software Engineering*, San Diego, California, pages 439–449, 1981.

[10] The Wisconsin Program-Slicing Tool 1.0, Reference Manual. Computer Sciences Department, University of Wisconsin-Madison, 1997.

A 集約アルゴリズム

以下,

- 連続文 (AlgorithmSerial)
- ブロック (AlgorithmBlock)
- 分岐文 (AlgorithmSelection)
- 分岐文 (else なし)(AlgorithmSelection')
- 繰り返し文 (AlgorithmIteration)

の集約アルゴリズムを順に示す. 判定欄の記号は

∅: 空集合
*****: 任意
−: 非空集合

を表しており, 上から集合の状態の組合せと同じもの検索し, 合致した行の右端の判定に従う.

○: 無条件に集約
×: 集約しない
(n): 表下の式 (n) で集約判定

アルゴリズム AlgorithmSerial
前提: 節点 A, 節点 B: 集約節点 もしくは 単一文節点

入力: 節点 A, 節点 B

判定:

USE (A)	po DEF (A)	USE (B)	po DEF (B)	
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	○
$\emptyset$	$\emptyset$	$*$	$*$	×
$*$	$*$	$\emptyset$	$\emptyset$	×
$\emptyset$	$-$	$\emptyset$	$-$	○
$\emptyset$	$-$	$-$	$\emptyset$	×
$*$	$*$	$*$	$*$	(1)

$|\ \mathrm{USE(A)} \cup \mathrm{USE(B)} - \mathrm{USE(A)} \cap \mathrm{USE(B)} - \mathrm{CTL(S)} - \mathrm{poDEF(A)} \cap \mathrm{USE(B)}\ | \leq \mathrm{limit} \ldots (1)$

集約: $\mathrm{USE(S)} = \mathrm{USE(A)} \cup (\mathrm{USE(B)} - \mathrm{DEF(A)})$
$\mathrm{DEF(S)} = \mathrm{DEF(A)} \cup \mathrm{DEF(B)}$
$\mathrm{poDEF(S)} = \mathrm{poDEF(A)} \cup \mathrm{poDEF(B)}$
出力: 集約 (連続文) 節点 S
関連: $\mathrm{CTL(S)} = \mathrm{CTL(A)} = \mathrm{CTL(B)}$

アルゴリズム AlgorithmBlock
前提: 節点 A: 集約節点 もしくは 単一文節点
入力: 節点 A
判定: ○
集約: $\mathrm{USE(S)} = \mathrm{USE(A)}$
$\mathrm{DEF(S)} = \mathrm{DEF(A)}$
$\mathrm{poDEF(S)} = \mathrm{poDEF(A)}$
出力: 集約 (ブロック) 節点 S
関連: $\mathrm{CTL(S)} = \mathrm{CTL(A)}$

アルゴリズム AlgorithmSelection
前提: then 節節点 B, else 節節点 C: 集約 (ブロック) 節点 もしくは 単一文節点
入力: 条件節節点 A, then 節節点 B, else 節節点 C

判定:

USE (A)	USE (B)	USE (C)	
$*$	$\emptyset$	$\emptyset$	○
$\emptyset$	$\emptyset$	$-$	○
$\emptyset$	$-$	$\emptyset$	○
$\emptyset$	$-$	$-$	(1)
$-$	$\emptyset$	$-$	(2)
$-$	$-$	$\emptyset$	(3)
$-$	$-$	$-$	(4)

$|\ \mathrm{USE(B)} \cup \mathrm{USE(C)} - \mathrm{USE(B)} \cap \mathrm{USE(C)}\ | \leq \mathrm{limit} \ldots (1)$
$|\ \mathrm{USE(C)} - \mathrm{USE(A)}\ | \leq \mathrm{limit} \ldots (2)$
$|\ \mathrm{USE(B)} - \mathrm{USE(A)}\ | \leq \mathrm{limit} \ldots (3)$
$|\ \mathrm{USE(B)} \cup \mathrm{USE(C)} - \mathrm{USE(A)}\ | \leq \mathrm{limit} \ldots (4)$

集約: $\mathrm{USE(S)} = \mathrm{USE(A)} \cup \mathrm{USE(B)} \cup \mathrm{USE(C)}$
$\mathrm{DEF(S)} = \mathrm{DEF(B)} \cap \mathrm{DEF(C)}$
$\mathrm{poDEF(S)} = \mathrm{poDEF(B)} \cup \mathrm{poDEF(C)}$
出力: 集約 (条件文) 節点 S
関連: $\mathrm{CTL(B)} = \mathrm{CTL(C)} = \mathrm{USE(A)}$

アルゴリズム AlgorithmSelection'
前提: then 節節点 B: 集約 (ブロック) 節点 もしくは 単一文節点
入力: 条件節節点 A, then 節節点 B

判定:

USE (A)	USE (B)	
$\emptyset$	$*$	○
$*$	$\emptyset$	○
$-$	$-$	(1)

$|\ \mathrm{USE(B)} - \mathrm{USE(A)}\ | \leq \mathrm{limit} \ldots (1)$

集約: $\mathrm{USE(S)} = \mathrm{USE(A)} \cup \mathrm{USE(B)}$
$\mathrm{DEF(S)} = \emptyset$
$\mathrm{poDEF(S)} = \mathrm{poDEF(B)}$
出力: 集約 (条件文) 節点 S
関連: $\mathrm{CTL(B)} = \mathrm{USE(A)}$

アルゴリズム AlgorithmIteration
前提: 繰り返し節節点 B: 集約 (ブロック) 節点 もしくは 単一文節点
入力: 条件節節点 A, 繰り返し節節点 B

判定:

USE (A)	USE (B)	po DEF (B)	
$\emptyset$	$*$	$*$	○
$-$	$\emptyset$	$*$	○
$-$	$-$	$*$	(1)

$\mathrm{poDEF(B)} \cap \mathrm{USE(A)} \neq \emptyset$ または
$|\ \mathrm{USE(B)} - \mathrm{USE(A)}\ | \leq \mathrm{limit} \ldots (1)$

集約: $\mathrm{USE(S)} = \mathrm{USE(A)} \cup \mathrm{USE(B)}$
$\mathrm{DEF(S)} = \emptyset$
$\mathrm{poDEF(S)} = \mathrm{poDEF(B)}$
出力: 集約 (繰り返し文) 節点 S
関連: $\mathrm{CTL(B)} = \mathrm{USE(A)}$