Committed-Choice 型言語 Fleng における静的粒度最適化

荒木拓也† 田中英彦†

Committed-Choice 型言語 Fleng は，データフロー同期の機構を用いることにより，容易に大量の並列性を抽出することが可能である．しかし，実行の粒度が非常に細かいため，同期やゴール（Fleng における実行の単位）の起動といった，細粒度実行に由来するオーバヘッドが大きい．並列度を低下させない程度にプログラムの粒度を大きくすることができれば，オーバヘッドを低減することができる．しかし，Fleng においてプログラムの意味を変えずに粒度を大きくすることは容易ではない．複数のゴールを 1 つのゴールに融合するゴール融合を行えば，プログラムの粒度を大きくできるが，安易なゴール融合はデッドロックを招き，プログラムの意味を変えてしまうからである．このデッドロックが起こるのは，融合により循環的な依存関係を作り出してしまうことが原因である．循環的な依存関係は，融合するゴール間に間接的な依存関係が存在する場合に作られる．本研究では，プログラムのデータフローを大域的に解析し，融合するゴール間に間接的な依存関係がないことを保証することによって，デッドロックを起こさないでゴール融合を行う手法の提案を行った．また，この手法をコンパイラに実装し，並列計算機上で評価を行った．プロセッサ台数にかかわらず，小規模のプログラムで数倍，実用規模のプログラムで 1.2 倍程度の速度向上を達成した．

A Static Granularity Optimization Method of a Committed-Choice Language Fleng

TAKUYA ARAKI† and HIDEHIKO TANAKA†

A committed-choice language Fleng can extract much parallelism easily from any programs using dataflow synchronization. However, there is large overhead such as synchronization and goal (a unit of Fleng computation) invocation, because the granularity of execution is very fine. If granularity of a program is coarsened, such overhead can be reduced; but it is not easy to make granularity coarse. When several goals are fused into one goal, granularity of a program can be coarsened, but this may cause deadlock and change behavior of the program. This deadlock is caused by cyclic dependency made by the goal fusion. Cyclic dependency is made when there is indirect dependency between fused goals. In this paper, we propose a goal fusion algorithm which analyzes global dataflow of a program and ensures that there is no indirect dependency between fused goals. We implemented this algorithm and evaluated it on a parallel computer. The evaluation shows that enough speedup can be attained by this method.

1. はじめに

Committed-Choice 型言語 Fleng[5]は，単一代入変数と，データフロー同期の機構を用いることにより，ゴール間にわたる並列性を含め，容易に大量の並列性を抽出することが可能である．

しかし，実行の粒度が非常に細かいため，ゴールのフォーク，ゴール間のコンテクストスイッチ，同期，通信といった，並列処理に由来するオーバヘッドが大きくなる．したがって，このオーバヘッドの削減が，効率的な実装の鍵となる．

オーバヘッドの主な原因は，実行の粒度が細か過ぎることにある．したがって，並列度を低下させない程度にプログラムの粒度を大きくすることができれば，オーバヘッドを低減することができる．しかし，Fleng において，粒度をプログラムの意味を変えずに大きくすることは容易ではない．安易にプログラムを変更すると，デッドロックを引き起こすためである．

本研究は，Committed-Choice 型言語 Fleng において，プログラムの意味を変えずにその粒度を最適化することを目的とし，アルゴリズムの提案，実装，評価を行った．

本稿の構成は以下のとおりである．2 章では Fleng について説明する．3 章では，本研究で用いる粗粒度化手法であるゴール融合について，その問題点と解決

† 東京大学工学系研究科
School of Engineering, The University of Tokyo

法を示す．4章では本手法の実装と評価について述べる．5章では関連研究について述べる．

2. Committed-Choice 型言語 Fleng

Fleng は論理型言語を祖先とする並列記号処理言語である．シンタックスは Prolog のものとよく似ているが，バックトラックを行わないという点で，セマンティクスは大きく異なる．Committed-Choice 型言語には，他に GHC[10]，KL1[3] などがある．

Fleng は，

- すべてのゴールを並列に実行する
- 単一代入変数を用いたデータフロー同期をとる

ことにより，細粒度，高並列なプログラムの実行が可能である．この点が Fleng の大きな特徴になっている．以下に例をあげながら説明する．

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

このプログラムは R = (A + 1) * 2 を実行するものである．":-" の左側をヘッド，右側をボディといい，全体で**定義節**と呼ぶ．Fleng のプログラムはこのような定義節の集まりである．Fleng における計算の単位はゴールと呼ばれる．このプログラムの場合，foo(A,R) という初期ゴールが与えられると，add(A,1,B)，mul(B,2,R) という 2 つのゴールに書き換えらる．この書き換えの操作を**リダクション**という．書き換えられた add と mul はそれぞれが並列に実行される．しかし，この場合，mul の方は B の値が決定するまで実行することはできない．このような場合，add の実行が終了し B の値が決まるまで，mul は実行を中断（**サスペンド**）し，add の実行が終了し B の値が決定すると，実行を再開（**アクティベイト**）する．この機構を矛盾なく実現するため，変数は単一代入であり，書き換えることはできない．変数は値が決まっていないか決まっているかの 2 つの状態を持ち，一度決まってしまえば，その値が変わることはない．変数の値を決めることを**具体化**するという．

分岐は次のように表す．

```
foo(true,R):- R = 1.
foo(false,R):- R = 0.
```

このプログラムは第一引数が true ならば R = 1，false ならば R = 0 を実行するプログラムである．この場合，さきほどと同様に，第一引数が決まるまでサスペンドし，値が決まったところでアクティベイトされ，その値によって分岐を行う．ただし，true や false のように小文字で始まるものはシンボルを表す．変数は大文字で始まる．

また，算術演算は

```
add(#A,#B,R):- compute(+,A,B,R).
```

のように定義されている．ここで "#" のついた変数は具体化されるまで待つことを表す．また，"compute" はサスペンドせずに実行し，ほぼアセンブリ言語の add 命令にコンパイルされる．したがって，compute を用いる際は # によって値が具体化されていることを保証しなければならない．

compute と = はサスペンドせずに実行可能なため，ゴールのフォークではなく，リダクション時に直接実行される．また，複数の compute，= があった場合は，記述順に逐次に実行される．

3. ゴール融合

本研究では，プログラムの粒度を大きくする手法として，ゴール融合を用いる．ゴール融合とは，複数のゴールを 1 つのゴールにまとめることによって粒度を大きくする手法である．以下にゴール融合の手法について述べる．

3.1 ゴール融合の問題点

ゴール融合を行う場合は，それを適用することによってプログラムの意味が変化しないことを保証しなければならない．この判定はそれほど容易なことではない．次に例をあげる．

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
add(#A,#B,R):- compute(+,A,B,R).
mul(#A,#B,R):- compute(*,A,B,R).
```

このプログラムは，R = U + U，S = V * V を実行する．このプログラムにおいて，foo から呼び出されている add と mul を融合し，

```
foo(U,V,R,S):- add_mul(U,V,R,S).
add_mul(#U,#V,R,S):-
  compute(+,U,U,R), compute(*,V,V,S).
```

とすることはできない．

foo(1,Tmp,Tmp,S) という呼び出しを行った場合，元のプログラムでは，add(1,1,Tmp) を実行することにより Tmp = 2 を出力し，さらに mul(Tmp,Tmp,S) を実行することにより，S = 4 という結果を返す．しかし，変更後のプログラムでは add_mul(1,Tmp,Tmp,S) が Tmp の値が決まるのを待ち続けるためデッドロックする．したがってゴールを融合することによってプログラムの意味を変えてしまうので，この変換は誤りである．ゴール融合を行う場合は，このようなプログラムの意味を変えてしまう変換を避けなければならない．

このプログラムにおいて，add と mul の融合がデッドロックを引き起こすのはなぜかを考えてみよう．foo(1,Tmp,Tmp,S) という呼び出しを行った場合

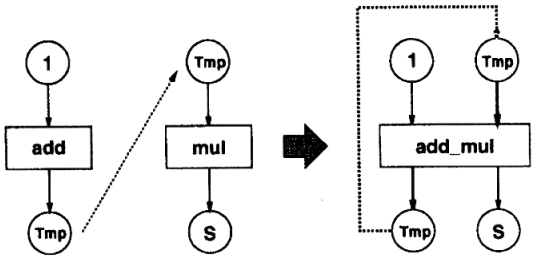

図1 データフローグラフ
Fig. 1 Dataflow graph of original goals and the fused goal.

の，元のプログラムのデータフローグラフを**図1**の左側に，融合した場合のデータフローグラフを図1の右側に示す．

融合前のデータフローグラフを見ると，呼び出し側の変数 **Tmp** を通じて **mul** が **add** に依存していることが分かる．この依存関係は点線で表している．融合後のデータフローグラフを見ると，この依存関係を通じて **Tmp** から **Tmp** という**サイクリックな依存関係**が存在する．サイクリックな依存関係があると，自分の出力に依存するわけであるから，デッドロックを引き起こす．もとのプログラムでは，サイクリックな依存関係は存在しないため，デッドロックが起こらない．

3.2 ゴール融合のアルゴリズム

デッドロックの原因はサイクリックな依存関係であるから，ゴール融合を行う際には，サイクリックな依存関係を作らないようにすればよい．したがって基本アルゴリズムは次のようになる．

> **"ある2つのゴールが融合可能である条件は，その2つのゴールの間に間接的な依存関係が存在しないことである"**

これは，2つのゴール間に間接的な依存関係が存在する場合は，融合すると必ずサイクリックな依存関係を作り出してしまうためである．

例を用いて説明する．**図2**のようなデータフローグラフで表されるプログラムを考える．

円は変数，四角はゴールを表す．矢印は依存関係を表す．この場合のように実線で表した依存関係は直接的な依存関係を表す．すなわち，ゴールから変数への矢印は，そのゴールのリダクション時に（**compute** あるいは = によって），変数が具体化されることを表す．また，変数からゴールへの矢印は，その変数が具体化されないとそのゴールはリダクションできないことを表す．

このプログラムにおいて，ゴール f と g が融合できるかどうかを考えると，g は h を介して f に間接的に

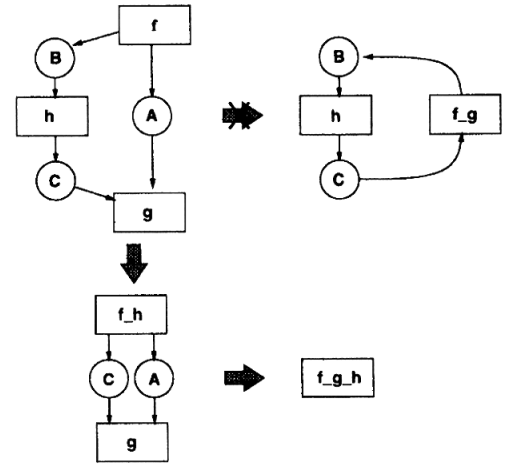

図2 ゴール融合のアルゴリズム
Fig. 2 Goal fusion algorithm.

依存しているため，f と g は融合することはできない．実際融合した場合，サイクリックな依存関係を作ってしまう．このようなサイクリックな依存関係は，2つのゴールが間接的に依存する限り，必ず生ずる．融合によって，同じゴールから出て同じゴールに入る依存を作るからである．

次にゴール f と h が融合できるかを考えると，h は f に依存しているが，これは直接的な依存関係であるため，融合可能である．なぜならば，直接的な依存関係は融合によって解消され，融合されたゴールの中に取り込まれるからである．f と h を融合した f_h と g は先ほどと同じように融合可能である．結局 f，g，h は1つのゴールに融合可能である．

3.3 依存関係の解析

前節では，直接的な依存関係で構成されたデータフローグラフを例として用いた．しかし，間接的な依存関係を作るのはこのような例だけではない．たとえば最初に融合できない例としてあげたプログラムの場合は，ヘッドの引数の間で間接的な依存関係を作っている．

ゴールの融合が可能かどうかを判定するには，2つのゴール間に間接的な依存関係が存在しないことを保証しなければならない．これは誤った変換をしないよう，安全に行う必要がある．かといって，存在しない依存関係まで存在するとしてしまうと，可能な融合も不可能であると判定してしまう．すなわち，依存関係の解析の正確さが，ゴール融合の性能に大きな影響を与える．

本節では依存関係の解析法について述べる．

3.3.1 モード推論

Flengは関数型言語などと異なり，変数が入力か出力かシンタックスからは判断できない．依存関係を解析するには変数の入出力を知らなければならないが，入力か出力か判断できない変数については，安全側に倒して判断しなければならず，依存関係を解析する際に性能を低下させてしまう．ある引数が入力か出力かを知るには，モード推論を行う必要がある．

文献11)ではwell-moded なGHCプログラムについて構造データも含めたモード推論を行っているが，本研究で行ったモード推論は，変数単位の単純なモード推論である．また，モード推論の考え方は文献11)のものを用いている．

アルゴリズムは以下のようになる．まず定義節から直接入出力を行うことが分かるゴールを検出する．これと後述する定義節間解析によりいくつかのゴールの入出力モードが分かっているものとする．この状態で，以下の推論を行う．

(1) ある変数があるゴールの出力なら，その変数を共有している他のゴールでは入力である．
(2) ある変数が1つを除いたすべてのゴールで入力なら，残りの1つのゴールでは出力である．

図で表すと**図3**のようになる．図の左側では，変数Aはゴールfの出力である．したがって，Aを共有するゴールg，hにおいてはAは入力であると推論できる．また，図の右側では変数Aはゴールg，hの入力である．Aを共有しているゴールは他にfしかないので，Aはfの出力であると推論できる．

推論できなかった変数は，入力である可能性も出力である可能性もあるとして扱う必要がある．

3.3.2 依存関係の検出

プログラム中で，依存関係を作りうるものについて，以下に検出法を述べる．この検出は，安全側に倒して行う必要がある．すなわち，依存関係が存在する可能性があれば，必ずしも存在しないものでも，依存関係があるとして扱う必要がある．

ゴールフォーク そのゴールのリダクション時にcomputeあるいは=によって変数が具体化される場合は，そのゴールからその変数へ直接的な依存関係が存在する．また，ある変数が具体化されないとそのゴールがリダクションできない場合は，その変数からそのゴールへ直接的な依存関係が存在する．

しかし，安全に依存関係を検出するにはそれだけではなく，すべての入力の可能性のある変数から，すべての出力の可能性のある変数へ依存関係があるとしなければならない．そのゴール自体では依存関係がなくても，そのゴールから呼ばれるサブゴールの中で依存関係があるかもしれないからである．

このように，可能性としては存在するが確実に存在するわけではない依存関係を，文献7)にならい**潜在的な依存関係**と呼ぶ．また，データフローグラフ上では破線で表す．潜在的な依存関係は，融合時に解消することができないので，間接的な依存関係として扱われる．

たとえば，

```
foo(A,R):-
  add(1,A,T1), bar(T1,T2), mul(T2,2,R).
```

のようなプログラムの場合，barのT1は入力でT2は出力と推論される．したがって，T1からT2という依存関係が存在するとしなければならない．この場合，addとmulはbarのT1とT2を通じて間接的に依存する可能性があるので，融合できないことが分かる（**図4**）．

barの定義によっては，T1からT2という依存関係がない場合もある（たとえばT1の値を捨てて，T2を定数に具体化するような場合）．しかし，一般には安全に融合を行うため，依存関係があるとしなければならない．barの定義が分かっていて，T1からT2という依存関係がないことが分かる場合は，後述する定義節間解析で検出する．

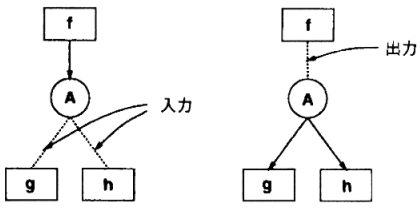

図3 モード推論の例
Fig. 3 Example of mode inference.

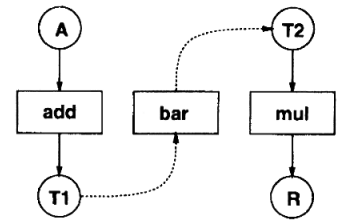

図4 ゴールフォークによる依存関係
Fig. 4 Dependency made by goal fork.

3.3.1 モード推論

Flengは関数型言語などと異なり，変数が入力か出力かシンタックスからは判断できない．依存関係を解析するには変数の入出力を知らなければならないが，入力か出力か判断できない変数については，安全側に倒して判断しなければならず，依存関係を解析する際に性能を低下させてしまう．ある引数が入力か出力かを知るには，モード推論を行う必要がある．

文献11)ではwell-moded なGHCプログラムについて構造データも含めたモード推論を行っているが，本研究で行ったモード推論は，変数単位の単純なモード推論である．また，モード推論の考え方は文献11)のものを用いている．

アルゴリズムは以下のようになる．まず定義節から直接入出力を行うことが分かるゴールを検出する．これと後述する定義節間解析によりいくつかのゴールの入出力モードが分かっているものとする．この状態で，以下の推論を行う．

(1) ある変数があるゴールの出力なら，その変数を共有している他のゴールでは入力である．
(2) ある変数が1つを除いたすべてのゴールで入力なら，残りの1つのゴールでは出力である．

図で表すと**図3**のようになる．図の左側では，変数Aはゴールfの出力である．したがって，Aを共有するゴールg，hにおいてはAは入力であると推論できる．また，図の右側では変数Aはゴールg，hの入力である．Aを共有しているゴールは他にfしかないので，Aはfの出力であると推論できる．

推論できなかった変数は，入力である可能性も出力である可能性もあるとして扱う必要がある．

3.3.2 依存関係の検出

プログラム中で，依存関係を作りうるものについて，以下に検出法を述べる．この検出は，安全側に倒して行う必要がある．すなわち，依存関係が存在する可能性があれば，必ずしも存在しないものでも，依存関係があるとして扱う必要がある．

ゴールフォーク そのゴールのリダクション時にcomputeあるいは=によって変数が具体化される場合は，そのゴールからその変数へ直接的な依存関係が存在する．また，ある変数が具体化されないとそのゴールがリダクションできない場合は，その変数からそのゴールへ直接的な依存関係が存在する．

しかし，安全に依存関係を検出するにはそれだけではなく，すべての入力の可能性のある変数から，すべての出力の可能性のある変数へ依存関係があるとしなければならない．そのゴール自体では依存関係がなくても，そのゴールから呼ばれるサブゴールの中で依存関係があるかもしれないからである．

このように，可能性としては存在するが確実に存在するわけではない依存関係を，文献7)にならい**潜在的な依存関係**と呼ぶ．また，データフローグラフ上では破線で表す．潜在的な依存関係は，融合時に解消することができないので，間接的な依存関係として扱われる．

たとえば，

```
foo(A,R):-
  add(1,A,T1), bar(T1,T2), mul(T2,2,R).
```

のようなプログラムの場合，barのT1は入力でT2は出力と推論される．したがって，T1からT2という依存関係が存在するとしなければならない．この場合，addとmulはbarのT1とT2を通じて間接的に依存する可能性があるので，融合できないことが分かる（**図4**）．

barの定義によっては，T1からT2という依存関係がない場合もある（たとえばT1の値を捨てて，T2を定数に具体化するような場合）．しかし，一般には安全に融合を行うため，依存関係があるとしなければならない．barの定義が分かっていて，T1からT2という依存関係がないことが分かる場合は，後述する定義節間解析で検出する．

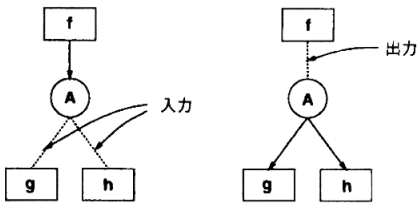

図3 モード推論の例
Fig. 3 Example of mode inference.

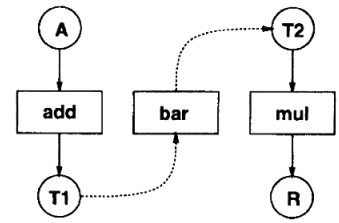

図4 ゴールフォークによる依存関係
Fig. 4 Dependency made by goal fork.

ユニファイ　変数とシンボル，数字などとのユニファイの場合は依存関係は考えなくてよい．変数どうしの場合は，お互いに依存関係があるとすればよい．たとえばA = Bの場合はAはBに，BはAに依存しているとすればよい．

これらに対して変数と構造データのユニファイの場合は注意が必要である．たとえば，A = [B|C]というユニファイの場合（[B|C]はcarがB，cdrがCのリストを表す），本質的にはこれらの変数A，B，Cの間には何の依存関係も存在しない．しかし，仮想的にAからB，Cへ，B，CからAへという依存関係が存在するとしないと，本来存在する依存関係を検出できないことがある．たとえば，

```
foo(...):- A = [B|C], bar(A,X), ...
bar([B|C],X):- add(B,C,X).
```

のような定義を考えてみよう．fooにおいて，barというゴールフォークによってAからXへの依存関係は検出できる．しかし，barはAのcarとcdrを取り出してaddに渡しているので，fooにおいて，Bから X，Cから Xという依存関係も存在する．これはBからA，CからAという依存関係が仮想的にあるという扱いにしないと検出できない．

また，次のような例を考える．

```
foo:-
  bar(B,C,A), A = [X|Y], baz(X,Y,Z)
bar(B,C,A):-
  add(B,C,X), mul(B,C,Y), A = [X|Y].
baz(X,Y,Z):- sub(X,Y,Z).
```

fooにおいて，ゴールbarにより，B，CからAへの依存関係は検出でき，ゴールbazにおいて，X，YからZへの依存関係は検出できる．ここで，fooのA = [X|Y]は，リストのAからcarとcdrを取り出しているので，fooにおいて，XはBに，YはCに依存している．この依存関係はX，YがAに仮想的に依存しているという扱いにしないと検出できない．ただし，プログラムがwell-modedである場合は，すでに具体化されたAに対してA = [X|Y]を実行してはならないので，この依存関係は考慮しなくてもよい．

以上をまとめると，構造データに対するユニファイがあった場合，その構造データを経由した依存関係の存在を考慮しなければならない（図5）．

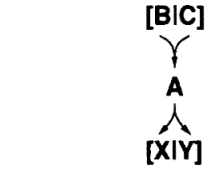

図5　構造データを介した依存関係
Fig. 5　Dependency through structured data.

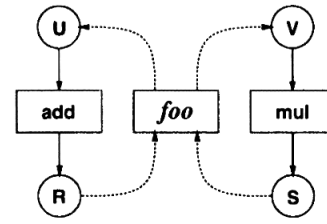

図6　ヘッドを介した依存関係
Fig. 6　Dependency through head.

ヘッド　最初にあげた融合できない例のように，ヘッドの引数どうしは潜在的な依存関係を作りうる．これは，依存関係を作るのが呼び出し側か被呼び出し側かが違うだけで，モードの分からないボディーゴールとまったく同じ状況である．したがって，モード推論，依存関係の解析において，ヘッドはボディーゴールとまったく等価に扱う．この場合，ヘッドのモードは外から見た場合のモードとちょうど反対になる．

たとえば，融合できない例として最初にあげた，次のプログラムを考える．

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
```

ヘッドを含めてこのプログラムのデータフローグラフを書くと図6のようになる．ヘッドは斜体で表す．この場合，ヘッドfooの入出力モードを推論すると，R，Sが入力，U，Vが出力になる．このモードはプログラムを外側から見たときとは逆になる．

このデータフローグラフを見ると，RからV，SからUといった依存関係が存在する可能性があることが分かるため，addとmulは融合できないことが分かる．

3.3.3　矛盾する依存関係の排除

前項で存在しうる依存関係の求め方を示したが，その中にはありえない依存関係も存在する．たとえば，

```
foo:- bar(Y,X), add(X,X,Y).
```

のbarにおいて，YからXという依存関係は存在しえない．なぜならばaddによってYはXに依存していることが分かるので，もしYからXという依存関係が存在すれば，デッドロックするからである．与えられたプログラムにバグがなく，正常に動作するものであると仮定すると，このような依存関係は存在しない

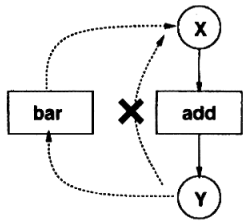

図7 矛盾する依存関係の排除
Fig. 7 Removal of potential dependency which is contradicted by certain dependency.

(図7).

本研究では，文献7) に述べられている方法を用い，

"確実な依存関係に矛盾する潜在的な依存関係は排除する"

という方針をとる．潜在的な依存関係が確実な依存関係と矛盾する場合は，潜在的な依存関係の方が，実は存在しないことが保証できるわけである．

3.4 定義節間解析

これまでは定義節内での解析について述べた．本節では定義節間での情報の伝搬について述べる．

情報を受け渡す方向には，被呼び出し側から呼び出し側（Bottom-Up）と呼び出し側から被呼び出し側（Top-Down）の2種類があり，定義節間で受け渡す情報には入出力モードと変数間の依存関係の2種類がある．依存関係は，矛盾する依存関係を排除するため，確実な依存関係と潜在的な依存関係に分けて渡す．分岐する方向によって情報が異なる場合は安全側に倒して情報を渡す．

解析はトップゴールを与え，そこから呼び出されるゴールをたどることによって行う．これは，深さ優先で呼び出し木をたどればよい．たどったゴールは記憶しておき，再帰があった場合はそこで解析をやめる．

被呼び出し側から呼び出し側（Bottom-Up） 次のような例を考える．

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
bar:- foo(U,V,R,S),...
```

foo で解析した情報を bar に伝えることを考える．foo の入出力モードは先ほど述べたモード推論を行うことによって知ることができる．ただし，モード推論によって得られるモードはボディー側から見たモードのため，反転する必要がある．この場合，U, V が入力，R, S が出力である．

次に依存関係であるが，これはヘッドを除いて上で述べた依存関係の解析を行えばよい．得られる依存関係が foo の中での依存関係である．ここでは，U から R，V から S という確実な依存関係が得られる．それ以外の潜在的な依存関係はない．伝搬の様子を図8に示す．

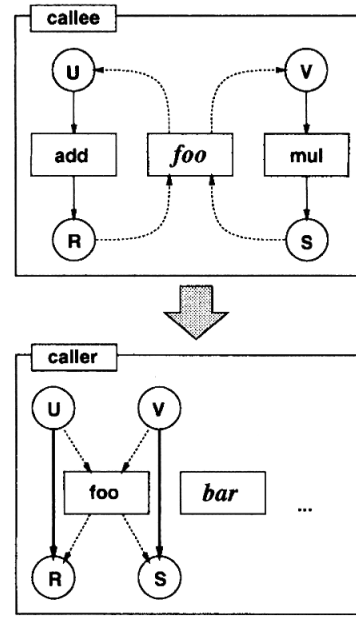

図8 定義節間解析
Fig. 8 Interclause analysis.

また，ここで注目すべきは，依存関係の伝搬によって，U から S，V から R という依存関係が存在しないことが分かる点である．モード情報のみではこれは分からない．

呼び出し側から被呼び出し側（Top-Down） 本研究における実装では，呼び出し側から被呼び出し側への情報の受渡しは複雑さを嫌って採用していない．被呼び出し側が複数の場所から呼ばれる場合は，複数の場所で得られる情報を安全にまとめて伝えなければならず，また，プログラムの外部から呼ばれる可能性がある場合は情報を伝えることはできないからである．このような複雑さはあるが，手法は本質的に被呼び出し側から呼び出し側への情報の受渡しと同じである．

4. 実装と評価

4.1 実　　装

コンパイラのプリプロセッサとして実装した．Fleng プログラムを入力とし，粒度を大きくした Fleng プログラムを出力する．約6000行の Fleng プログラムである．また，分岐するゴールも融合できるようにするため，ゴール内で分岐を扱えるよう，コンパイラを拡張した．この分岐は (Cond --> Then ; Else) のように表す．この分岐は Cond を評価するために必要な

表 1 ベンチマークプログラム
Table 1 Benchmark programs.

プログラム名	実行内容	行数	入力サイズ	バイナリサイズ（バイト）		コンパイル時間	
				オリジナル	最適化後	オリジナル	最適化時
qu	N-Queens 問題	55 行	9-Queens	8925	7918	3.58 秒	7.04 秒
primes	素数生成	39 行	2000 までの素数	7076	7367	2.87 秒	5.00 秒
qsort	クイックソート	31 行	10000 要素	4365	5373	2.55 秒	14.1 秒
fme	Fleng マクロ展開	1175 行	マクロ展開前の qu（43 行）	194110	170054	135 秒	539 秒

変数が具体化されていることを仮定しており，手続き型言語と同様，単なる条件分岐命令にコンパイルされる．

また，サスペンドしないゴールはできるだけインライン展開するようにした．

以下に変換例として，絶対値を求めるプログラムをあげる．

```
abs(A,R):-
  greater(A,0,IsGt),abs1(IsGt,A,R).
abs1(true,A,R):- R = A.
abs1(false,A,R):- sub(0,A,R).
```

このプログラムを処理すると以下のようになる．

```
abs(A,B):- C = 0, greater_abs1(A,C,B).
greater_abs1(#A,#B,C) :-
  compute(>,A,B,D),
  ((D == true)-->
    C = A
  ;
    E = 0, compute(-,E,A,C)
  ).
```

この変換により，2 つのゴールフォークで構成されていた元のプログラムが 1 つのゴールフォークで実現され，ゴール起動とゴール間同期のオーバヘッドが削減されている．

4.2 評価条件

評価は並列推論エンジン PIE64[1)]の上で行った．PIE64 は Fleng の高速実行を目的に設計された並列計算機である．要素プロセッサ数は 64 台で分散共有メモリマシンである．プロセッサはマルチコンテクスト処理をサポートする．また，相互結合網は自動負荷分散の機能を持つ．実行時間の測定値は 3 回実行した値の平均値を用いた．ガーベジコレクションの時間は含まない．

また，粒度最適化部自身はプログラムの粒度を指定できるように実装しているが，今回の評価では粒度をできるだけ大きくするよう変換した．これは，関数型言語 Id のコンパイラでも指摘されているように[7)]，できるだけ大きな粒度にしても最適な粒度より細かい粒度にしかならなかったためである．

コンパイル時間は，ワークステーション（Sun Ultra 1）上で実行したときのものである．

4.3 評価

ベンチマークプログラムとして，qu, primes, qsort, fme を選んだ．qu は N-Queens 問題を解くプログラムである．primes はエラトステネスのふるいを用いた素数生成プログラムである．qsort はクイックソートプログラムである．fme は Fleng のマクロを展開するプログラムであり，実用規模のプログラムである．qu 以外のプログラムが持つ並列度は比較的少ない．

表 1 に，評価に使ったプログラムとコンパイルした後のバイナリサイズ，コンパイル時間を示す．

図 9～**図 12** に粒度最適化による速度向上を示す．横軸はプロセッサ台数，縦軸は粒度最適化を行わなかったときの 1 台のときの速度を 1 とした相対速度である．

qu 粒度最適化プログラムを通すことにより，コンパイルしたバイナリのサイズが減少した．これは，ゴール融合やインライン展開により無駄なコードが削除されたためであると考えられる．

図 9 に粒度最適化による速度向上を示す．プロセッサ台数にかかわらず，全領域で 9 倍以上の速度向上を示している．

primes コンパイルしたバイナリのサイズは若干増加した．増加分は，新たに生成された定義節によるものと思われる．

図 10 に粒度最適化による速度向上を示す．プロセッサ台数にかかわらず，全領域で 3～4 倍程度の速度向上を示している．

qsort コンパイルしたバイナリのサイズは若干増加した．primes と同様，増加分は，新たに生成された定義節によるものと思われる．

図 11 に粒度最適化による速度向上を示す．プロセッサ台数にかかわらず，全領域で 2 倍程度の速度向上を示している．

fme 実用規模のプログラムとして fme を評価した．fme は Fleng のマクロを展開するプログラムである．ここでは，入力としてマクロ展開前の qu

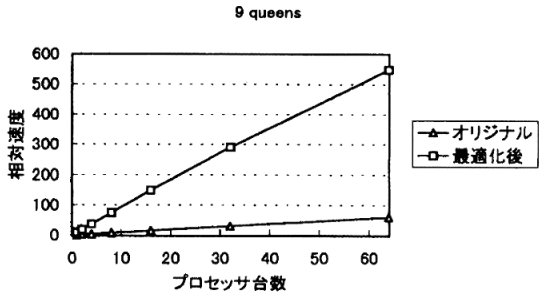

図9 9 queens の速度向上
Fig. 9 Speedup of 9 queens.

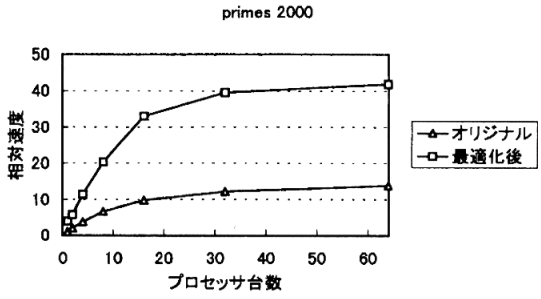

図10 primes の速度向上
Fig. 10 Speedup of primes.

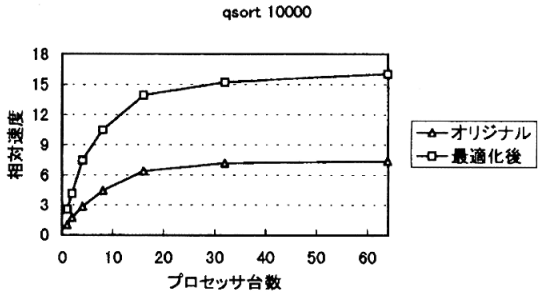

図11 qsort の速度向上
Fig. 11 Speedup of qsort.

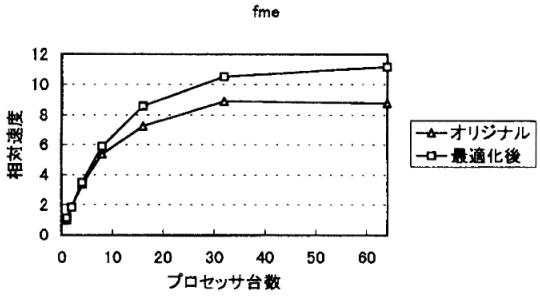

図12 fme の速度向上
Fig. 12 Speedup of fme.

(43行) を用いた．ただし，PIE64 とホスト計算機との通信時間を省くため，実際の入出力は行わなかった．すなわち，入力は定義節の形でプログラムに埋め込み (粒度最適化部には通さない)，出力は行わないようにした．

また，粒度最適化部は，定義節の大きさが大きいと処理に時間がかかるため，今回は一定義節内に変数が一定 (20個) 以上あると処理を行わないようにした．

コンパイルしたバイナリのサイズは減少した．図12に粒度最適化による速度向上を示す．最適化により，1.2倍程度の速度向上が見られた．小規模なプログラムほどではないが，実用規模のプログラムでも効果があることが示された．

以上のように，小規模のプログラムでは数倍，実用規模のプログラムでも1.2倍程度の速度向上を達成した．ゴール融合によりあらたに定義節が増え，コードサイズが増大することが予想されたが，増加量はわずかで減少する場合もあった．粒度最適化を行うことによって，コンパイル時間は2倍～6倍程度になった．若干時間がかかっているが，大域的な解析をしていることを考えると許容できる範囲であると考える．

5. 関連研究

Arvind が文献2) で述べたように，Id のような Lenient な関数型言語と Committed-Choice 型言語の実行モデルはよく似ており，本研究で行ったような静的粒度最適化は，関数型言語の分野でもさかんに研究されている[6)~9)]．関数型言語におけるスレッド生成の最新のアルゴリズムは，Separation Constraint Partitioning[7)] (以後 SCP) であり，本研究も大きな影響を受けている．

表現は異なるが本研究の基本アルゴリズムは SCP と本質的には同じものであると考えられる．しかし，SCP は入力としてデータフローグラフを前提とし，inlet/outlet アノテーションの存在を仮定しているので，Fleng のような入出力がプログラムのシンタックスから判定できない言語では，直接は適用できない．また，本研究で提案したアルゴリズムは，融合を途中でやめても正しい結果が得られるという点で，SCP と異なる．

また，本研究では Fleng プログラムのソースコードレベルで融合を行っているため，インライン展開などの他のソースコードレベルでの最適化を同時に行うことも可能である．今回の実装でも，融合によりインライン展開可能になった部分は積極的にインライン展開している．

Committed-Choice 型言語における粒度の最適化に関する研究としては，Massy と Tick の研究[4)]があげ

られる.

B.C. Massey と E. Tick は，対象プログラムを "feedback free" と呼ばれるクラスのものに限定することで，プログラムの逐次化を行っている．feedback free とは，「ヘッドを通して依存関係を生じないこと」を表す．たとえば，`foo(U,V,R,S):- add(U,U,R), mul(V,V,S)`．というプログラムで`foo(1,Tmp,Tmp,S)`のような呼び出しは禁止される．

また，プログラムのモードは上田の方法[11]によりすべて決定できるものとする．このクラスのプログラムは "fully-moded" と呼ばれる．fully-moded は well-moded よりもさらに厳しい条件である．プログラムは well-moded であっても情報が足りなくてモードが決定しない場合があるからである．彼らはこのようなプログラムに限定することによりプログラムの逐次化を行っている．

しかし，彼らの課した条件は非常に厳しい．特に feedback free であるという条件は多くのプログラムの記述を不可能にする．彼ら自身があげている例としては，差分リストを使ったプログラムが feedback free ではない．

さらに，彼らの手法では並列化は考えられておらず，プログラムの逐次化のみを扱っている．

6. おわりに

本研究では，Committed-Choice 型言語 Fleng において，プログラムの意味を変えずにその粒度を最適化することを目的とし，アルゴリズムの提案，実装，評価を行った．

本研究では，プログラムの粒度を大きくするためにゴール融合という手法を用いた．ゴール融合が可能かどうかの判定を行うアルゴリズムを提案し，提案したアルゴリズムに基づき Fleng プログラムの粒度を最適化するプログラムを実装し，評価を行った．粒度最適化により，小規模なプログラムでは数倍，実用規模のプログラムでも 1.2 倍程度の高速化が達成された．

実用規模のプログラムで性能向上率が低いのは大域的な解析が十分に行われなかったためであると考えられる．文献 12) のような手法を用い，解析が困難な場合でも粒度を大きくすることが今後の課題としてあげられる．

参考文献

1) Araki, T., Hidaka, Y., Nakada, H., Koike, H. and Tanaka, H.: System Integration of the Parallel Inference Engine PIE64, *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, pp.64–76 (1994).
2) Arvind: Shared Problems in Compiling Functional and Logic Languages, *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, Invited Talk (1994).
3) Chikayama, T.: Operating System PIMOS and Kernel Language KL1, *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp.73–88 (1992).
4) Massey, B.C. and Tick, E.: Sequentialization of Parallel Logic Programs with Mode Analysis, *4th International Conference on Logic Programming and Automated Reasoning*, LNCS, Vol.698, pp.205–216, Springer-Verlag (1993).
5) Nilsson, M. and Tanaka, H.: Fleng Prolog – The Language which Turns Supercomputers into Prolog Machines, *Logic Programming '86*, LNCS, Vol.264, pp.170–179, Springer-Verlag (1989).
6) Schauser, K.E., Culler, D.E. and Eiken, T.: Compiler-controlled Multithreading for Lenient Parallel Languages, *FPCA '91*, LNCS, Vol.523, pp.50–72, Springer-Verlag (1991).
7) Schauser, K.E., Culler, D.E. and Goldstein, S.C.: Separation Constraint Partitioning – A New Algorithm for Partitioning Non-strict Programs into Sequential Threads, *POPL '95*, pp.259–271, ACM (1995).
8) Traub, K.R.: Compilation as Partitioning: A New Approach to Compiling Non-strict Functional Languages, *FPCA '89*, pp.75–88, ACM (1989).
9) Traub, K.R., Culler, D.E. and Schauser, K.E.: Global Analysis for Partitioning Non-strict Programs into Sequential Threads, *LFP '92*, pp.324–334, ACM (1992).
10) Ueda, K.: Guarded Horn Clauses, Technical Report, TR-103, ICOT (1985).
11) Ueda, K. and Morita, M.: A New Implmentation Technique for Flat GHC, Technical Report, TR-560, ICOT (1990).
12) 荒木拓也，田中英彦：Committed-Choice 型言語 Fleng のインライン展開による粒度制御手法，第 53 回情報処理学会全国大会，Vol.1, No.3E-8, pp.347–348 (1996).

(平成 8 年 9 月 13 日受付)
(平成 9 年 4 月 3 日採録)

荒木　拓也（学生会員）

1971年生．1994年東京大学工学部電気工学科卒業．1996年同大学院情報工学専攻修士課程修了．現在，同大学院博士課程に在学中．並列プログラミング言語の最適化の研究に従事．並列関数型言語，並列オブジェクト指向言語，並列化コンパイラなどに興味を持つ．

田中　英彦（正会員）

1943年生．1965年東京大学工学部電子工学科卒業．1970年同大学院博士課程修了．工学博士．同年東京大学工学部講師．1971年助教授，1978～1979年ニューヨーク市立大学客員教授，1987年教授現在に至る．計算機アーキテクチャ，並列処理，人工知能，自然言語処理，分散処理，CAD等に興味を持っている．「非ノイマンコンピュータ」，「情報通信システム」著，「計算機アーキテクチャ」，「VLSIコンピュータI，II」，「ソフトウエア指向アーキテクチャ」共著，New Generation Computing編集長，電子情報通信学会，人工知能学会，ソフトウエア科学会，IEEE，ACM，各会員．