# Committed-Choice 型言語 Fleng における静的粒度制御手法

荒木 拓也, 小池汎平, 田中英彦
東京大学工学部

## 1 はじめに

Committed-Choice 型言語 Fleng は, 単一代入変数によって同期をとりながらすべてのゴールを並列に実行することにより, 非定型的な問題においても並列度を最大限抽出して実行可能な言語である. しかし, ゴールを単位とする並列実行は粒度が小さ過ぎるため, 同期, ゴールの起動などのオーバーヘッドが大きい。

このオーバーヘッドを取り除くためには, 粒度を大きくする必要があるが, これはプログラムの意味を変えてしまう可能性がある.

本稿では, 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手法について述べる.

## 2 関連研究

同様の問題は Non-Strict な関数型言語でも存在する. 文献 [1] では, Non-Strict な関数型言語のプログラムを, 意味を変えることなく, 複数の逐次スレッドに分割するアルゴリズムが提案されている. 我々のアルゴリズムは基本的にはこれを採用するが, Fleng に適用するため幾つかの拡張, 変更を加えている.

## 3 Fleng

Fleng は Committed-Choice 型言語の一種であり, ほぼ GHC からガードゴールを取り除いたものに相当する. ガードの機構はヘッドユニフィケーションによってのみ実現されている.

純粋な Fleng のレベルでは, 粒度を大きくするための低レベルな記述ができない. このため, 我々は Fleng の上位互換で, Fleng の記述力を高めた言語 Fleng−− 用い, これを粒度を調整する際のターゲット言語としている.

Fleng−−では,

- Non-Variable Annotation:
  値がバインドされるまで待つ. #で表す.
- システム述語 compute:
  サスペンドせずに実行する.
- Non-Blocking の分岐:
  サスペンドせずに分岐する.

などが導入されている.

A static granularity control method of a committed-choice language Fleng

Takuya ARAKI, Hanpei KOIKE, Hidehiko TANAKA
Faculty of Engineering,the University of Tokyo

compute, Non blocking の分岐は値が決まっていることを保証して使う必要がある. Fleng−−で add を書くと, 次のようになる.

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

## 4 粒度制御手法

### 4.1 粒度を大きくする際の問題点

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

の add と mul を 1 つのゴールに融合して,

```
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).
```

として粒度を大きくすることはできない. なぜならば, `?- foo(1,Tmp,Tmp,S)` という呼び出しに対して, 最初のプログラムは答を返すが (Tmp = 2, S = 4), 変更後のプログラムは Tmp の値が決まらないのでデッドロックするからである.

Fleng プログラムの粒度を大きくする際には, このようなデッドロックをひきおこす変換を避けなければならない.

### 4.2 アルゴリズム

文献 [1] では, **潜在的な依存関係**という概念を用いてこの問題を解決するアルゴリズムを提案している.

上の述語 foo のデータフローグラフを書くと図 1のようになる (図では入出力変数を丸で囲んでいる). このデータフローグラフにおいて, add と mul が 1 つのゴールにできないのは R が V の, あるいは S が U の入力になる可能性があるためである. これを潜在的な依存関係と呼び, 点線で表す (図 1). 点線でつながれたゴール同

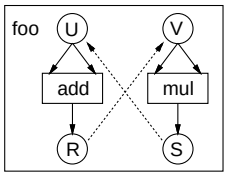


図 1: 1 つのゴールにできない例

士は 1 つのゴールにはまとめることができない.

反対に, 点線でつながっていなければ 1 つのゴールにまとめることが可能である. 例えば

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

のデータフローグラフは図 2のようになる. S はすでに U, V に依存しているので, S → U, V という潜在的な依存関係はあり得ない. したがって, これらは 1 つのゴールにまとめることが可能である.

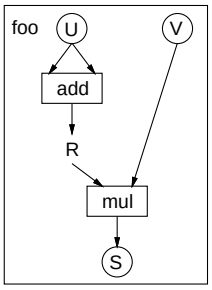


図 2: 1 つのゴールにできる例

### 4.3 ゴールの扱い

文献 [1] では, データフローグラフのノードに現れるもの (算術演算等) のみを扱っているが, Fleng において一般のゴールを対象とすると,

- 間接的な入力変数
- 間接的な出力変数
- モードの分からない変数

の扱いが問題となる. 間接的な入力/出力変数とは, そのゴールから呼び出されるサブゴールで入力/出力される変数を表す.

**間接的な入力変数**　入力を待たなければならないのは, 実際にはサブゴールであるから, データフローグラフ上では, つながっていないものと等価に扱う.

**間接的な出力変数**　ゴールが実行されてから値が決まるまで遅延が存在する. したがって, 出力を直接使うことができないので 1 つのゴールにまとめることはできない. この依存関係を**間接的な依存関係**と呼び, 波線で表す. 例えば

```
foo(U,V,S):- bar(U,U,R),mul(R,V,S).
bar(#A,#B,R):- compute(+,A,B,C),add(1,C,R).
```

の場合, 図 3のようになる.

間接的な依存関係は, 依存関係があることは示すので, 潜在的な依存関係を除去するために使うことは可能である. 例えば図 3の場合, S → U という潜在的な依存関係はあり得ない.

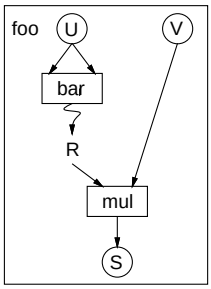


図 3: 間接的な出力変数

**モードの分からない変数**　モードが分からない変数を引数とするゴールを呼び出す場合は, それらの変数が潜在的な依存関係を作る可能性がある. 例えば,

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

を考えると, bar の中で, R = V, S = U が行なわれる可能性がある. したがって, 変数 R, S は foo にローカルな変数であるが, R と V, S と U の間に潜在的な依存関係を作るので, add と mul は 1 つのゴールにすることは出来ない (図 4).

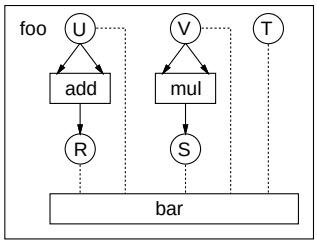


図 4: モードの分からない変数

## 5 まとめ

関数型言語において提案されている方法をもとに, Fleng プログラムの粒度を静的に大きくする手法を提案した. 粒度と並列度はトレードオフの関係にあるため, 実際に粒度を大きくする場合は最適な粒度を見つける必要がある. 今後, このトレードオフをとりながら粒度制御を行なうプログラムを実装し, 評価を行なう予定である.

## 参考文献

[1] K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation constraint partitioning – a new algorithm for partitioning non-strict programs into sequential threads. In *POPL '95*, pp. 259–271. ACM, 1995.