C1-2 Committed-Choice 型言語 FLENG−− 上の最適化コンパイル

The optimizing compilation of Committed-Choice language FLENG−−

下山 健, 小池 汎平, 田中 英彦
Takeshi Shimoyama, Hanpei Koike, and Hidehiko Tanaka
{ken,koike,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部
Faculty of Engineering, The University of Tokyo

本稿では、FLENG に効率良い並列処理やメモリ管理、実行の高速化を指示するアノテーション等を取り入れた FLENG の低水準記述言語、FLENG−− を紹介し、FLENG から中間言語やプロセッサのネイティブコードにコンパイルする際の最適化のうち、SRA(Single Reference Annotaion) と If-Then-Else 構文、Guard 構文を利用した最適化について紹介する。

1 はじめに

我々は現在、並列推論エンジン PIE64[1] の開発を進めている。PIE64 上では大規模な知識処理を行なうための言語として、Committed-Choice 型言語 FLENG[2]、およびその上位言語 FLENG++[3] を採用している。FLENG では、ゴールの実行がその論理的な真理値とは無関係に行なわれるため、ゴール間の論理的関係やガードがなく、実装が容易になっている。FLENG++ は、FLENG にオブジェクト指向のパラダイムを採り入れた言語である。FLENG++ は FLENG++ コンパイラによって FLENG または FLENG の中間言語にコンパイルされる。

PIE64 の各 IU(Inference Unit) には FLENG の実行を直接受け持つ推論プロセッサ UNIREDII[5] と、IU 全体を管理する汎用プロセッサ SPARC と、ネットワークの管理を行なう NIP (Network Interface Processor) があり、協調動作を行なっている。PIE64 から高い性能を引き出すためには、FLENG を実行する UNIRED IIのネイティブコードに FLENG あるいは FLENG++ を効率よくコンパイルしなくてはいけない。この際の最適化処理は PIE の実性能を決定する重要な要因となる。しかし、最適化処理によっては、その効果がプログラムの動的特性によるものや、並列実行時に初めて効果が発揮されるものなどがあり、最適化を行なうプログラムには柔軟性が要求される。そこで、FLENG に最適化のための情報を付加した言語、FLENG−− を定義することにより、プログラムから最適化のための情報を収集する部分とそれらの情報を実際のコードに具体化する部分を処理系から分離することができるようになった。こうすることにより、処理系が柔軟になり、また、多くの実験的な最適化を行なうことが比較的容易となった。本稿では、FLENG−− に付加された最適化情報のいくつかとそれらへの対応について概説する。

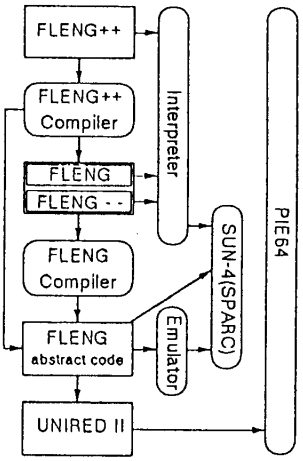

図 1: FLENG 言語処理系と PIE64

2 低水準記述言語 FLENG−−

FLENG には、その低水準記述言語として FLENG−− が用意されている。この言語は FLENG にいくつかのアノテーションや宣言文などを追加したものである。これらは効率良い並列処理やメモリ管理、実行の高速化を処理系やコンパイラに指示するものであり、プログラムの記述力を向上させるものである。FLENG−− は、FLENG のスーパーセットとなっており、追加されるアノテーション等を除いたり、簡単な変換を施すことにより FLENG の言語仕様内で、結果としては等価な FLENG のプログラムに変換することができる。これらのアノテーションや宣言は当初はユーザが細心の注意と時には好奇心を持って付けられるものである。ユーザからこれらの指示が与えられることにより、FLENG−− の処理系やコンパイラは、プログラムから最適化のための情報を収集する必要がなくなる。最終的には、これらのアノテーションや宣言はプリプロセッサによって自動生成されるものである。FLENG−− で追加されたアノテーションのうちいくつかをここに紹介する。

Active Unify Annotation (!)

このアノテーションは、定義節のヘッドにつけることが

できる。このアノテーションがついている項のヘッドユニフィケーションは常に成功し、コミット後に実際のアクティブユニフィケーションが行なわれる。例えば以下の2つのプログラムは全く同一である。

```
append([H|T],X,![H|Y]) :- append(T,X,Y).
append([H|T],X,Y) :- Y=[H|Y1], append(T,X,Y1).
```

Bind-to-non-variable Annotation (#)

このアノテーションは、定義節のヘッドの変数につく。このアノテーションがついている変数は、変数以外に束縛されなければその定義節はコミットされない。ただし、その変数がグランドまで落ちている必要はない。

Single Reference Annotation (‘)

このアノテーションは、定義節のヘッドの引数の構造体につけることができる。このアノテーションがつけられた構造体は、ゴール以外からの参照がないことが仮定される。このアノテーションにより以下で述べるような最適化を行なうことができる。以下にSRAを使ったAppendのプログラムを示す。

```
append(‘[H|#T],X,![H|Y]):-append(T,X,Y).
append([],X,!X).
```

FLENG−− にはこれらのアノテーションの他にも、以下のようなアノテーションが追加される予定である。

- セルの寿命に関する情報を与え、ガベージコレクションの戦略に影響するアノテーション
- サスペンド、アクティベイトが多く発生する定義節に関して、サスペンションリストの構造を変えてコミットできるようになるまでアクティベイトを遅延させるようにする宣言
- 複数の定義節のヘッドユニフィケーションに関して、その順番を制御するアノテーション

3 SRAによる最適化

単一のゴールから参照されている構造体は、ヘッドユニフィケーション、及びボディリダクション後はゴミとなるので即時に回収することができる。再利用の方法にはフリーリストで管理する等色々方法があるが、現在のコンパイラはSRAがつけられた定義節内で静的にわかる範囲において再利用を行なっている。この方法では、再利用する構造体を指すセルの内容をテンポラリレジスタに保存するだけで良く、フリーリスト等で回収するのに比べて動的なオーバーヘッドがなくなる。再利用できる場合は、新たにヒープからメモリを取る必要がなくなるため定義節によっては、新たにメモリを消費せず、GCの起動チェックを行なう必要がなくなる場合もある。SRAは構造体の単一参照性、つまり、再利用の可能性を表しているに過ぎなく、どのように再利用するかはコンパイラが判断しなくてはいけない。コンパイラの判断基準としては以下の二つのものが考えられる。

- 再利用できるメモリ領域の大きさ(メモリ低消費化)
- 再利用するメモリに書かれているデータ自体の再利用の可能性(速度の向上)

複数のSRAが定義節内に存在する場合や再利用する構造体が複数存在する場合、コンパイラはこれらの基準で最も適切な組合せを選択しなくてはいけない。

FLENGのように単一代入則を前提とした言語では、各定義節で処理すべき変数を多く持ち回ることが多く、プログラミングテクニックとしてそれらを一つの構造体にまとめる手法が取られることがある。このような場合、この構造体は一種の状態変数と考えることができる。したがって、プログラムの実行が進み、状態が変わる場合、構造体の中の一部分だけを書き換えてその他の部分は元のままである構造体をボディで作り出すような定義節を頻繁に実行しなくてはいけないような状態は容易に考えられる。このような場合、この構造体にSRAをつけることにより、メモリ消費と実行速度の両面で最適化の効果は大きいと考えられる。

一方、このSRAはFLENGの単一代入則を破って破壊的代入を行なうものなのでユーザは以下に述べるように細心の注意を持って使用しなくてはいけない。正しくSRAを付加すればプログラムの意味自体が変化することない。

3.1 構造体定数とSRA

プログラム中に記述されているグランドに落ちている構造体を構造体定数と呼ぶが、これらは従来のWAMベースの処理系では、ボディリダクション時にプログラムで明示的に生成される。しかし、この構造体定数はコンパイル時に内容が決定できるので、構造体定数をヒープ以外の場所にコンパイル時に作成し、使用する時にそれに対するポインタを返すようにすれば、構造体を作るための命令、時間を省くと共に、メモリ消費を抑えることができる。しかし、前述のSRAを使用したプログラムの場合、SRAのついた変数が構造体定数をアクセスし、内容を破壊してしまう可能性がある。構造体定数領域へのライト操作をハード的に検出してトラップなどにより動的に対応する方法も考えられるが、ハードウェアのサポートが必要となる。プログラムのグローバル解析によりこれを静的に検出することは可能ではあるが、現段階としては、SRAの自動生成と同様、これを行なわず、ユーザが細心の注意を持って処理系に知らせることにしている。SRAを表すマークには、’(バッククオート)を使用している。従来の仕様では、原理的にこのマークは定義節のヘッドにしか現れない。一方、最適化に使用できる構造体定数はボディに現れるものだけである。そこで、SRAにより破壊代入されるおそれのある構造体定数にもバッククオートをつけることにより区別できるようにする。処理系はアノテーションのついていない構造体定数に関してのみ従来通りの最適化を施すことができる。

3.2 未定義変数の構造体への埋め込みとSRA

PIE64上のFLENG処理系は、処理の高速化と省メモリのために未定義変数の構造体内部への埋め込みをサポートしている。これは、構造体の内部に存在の一意性が要求されるUNDEFのタグを持ったセルを直接書き込むものである。未定義変数を構造体へ埋め込む場合、この構造体を再利用するためには、再利用する側の構造体の未定義変数の実体が再利用される側の未定義変数の実体と同じでない場合は、埋め込まれた未定義変数が束縛されなくてはいけない。したがって構造体内の変数には安全のためにBind-to-non-variableアノテーションをつけなければいけない。もちろん、未定義変数の構造体への埋め込みを指示しなければ、このアノテーションをつける必要はなくなる。

4 マクロの導入

FLENGにはガードがなく、ゴールの実行はそれらの論理的な真理値とは無関係に行われる。ゴール間に論理的な関係が必要なときは、ユーザが共有変数によって、明示的に記述しなければいけない。このことは、処理系に対する負担を軽くするが、一方ユーザには多くの負担を強いる結果となる。たとえば、FLENGにおいて、条件分岐を行うときは、ユーザがデータの依存関係に注意しながら条件を判断する述語を呼び出してその論理値を計算し、さらにその結果をヘッドのマッチング規則に還元して、条件分岐を記述しなくてはいけない。しかし、このように記述されたプログラムは可読性が悪く、またプログラム自体の記述力も低いと言わざるを得ない。そこで、FLENGではプログラムの記述力と可読性を高めるために以下にあげる2種類のマクロが用意されている。(図2)

- PROLOG-like If-Then-Else
- GHC-like Guarded-Command

これらのマクロは条件部には、システムで定められたある一定のゴールのみ使用することができる。前者が条件部を逐次に判断するのに対し、後者は条件部を並列に判断するところが違う。このようなマクロを導入することにより、FLENGをFlatGHC的にとらえてプログラムすることができるようになった。また、さらにこの表現は任意にネストすることが可能であり、条件判断をより直接的に表現でき、記述力を大きく向上させることができる。

4.1 マクロの最適化

FLENGのインタプリタでは、これらのマクロはプリプロセッサによって図2のように自動的に展開されて実行される。FLENGコンパイラではこのマクロを展開せずに最適化の情報として利用している。コンパイラによって最適化される部分は以下の部分である。

- 条件部には限られた比較論理演算等しか記述できないので、これらをシステム述語のゴールとしてフォークする事なく直接コード上にコンパイルする。
- 条件部のどれかが成立するような状態になってから、はじめてその定義節をコミットし、新たなゴールをフォークするようにする。どの条件も成立しないときは強制的にサスペンドさせる。

GHC-like guarded-command

```
Head :- Cond1 | Goal1 ;
        :
        Condn | Goaln.
```

Prolog-like If-Then-Else

```
Head :- Cond -> TrueGoals ;
                FalseGoals.
```

Before Macro Expanding

```
a(A,B):-a > 1 | c(A);
        b > 1 | d(B).
b(A,B):-a > 1 -> c(A);
        b > 1 -> d(B).
```

After Macro Expanding

```
a(A,B):-gt(A,1,C),gt(B,1,D),a0(C,D,A,B).
a0(false,false,_,_).
a0(true,_,A,_):-c(A).
a0(_,true,_,B):-d(B).
b(A,B):-gt(A,1,C),b0(C,A,B).
b0(true,A,_):-b(A).
b0(false,_,A):-gt(A,1,C),b1(C,A).
b1(true,A):-c(A).
b1(false,A).
```

図2: マクロ構文と展開例

前者に関しては、従来は各条件部を判断するゴールを生成していたため、変数が束縛されていない状態でも、一部のゴールがサスペンドした状態でマクロ部分の実行を中断することが可能であった。しかし、最適化によりマクロ自体を一つのゴールとして扱うため、変数が束縛されていない状態でも条件判断をする必要がでてきた。そこで各条件部の論理値に未定義を加え、以下の4値とし、それぞれand,or,not論理演算に関して、表3のような論理演算表を対応させた。こうすることにより、変数の束縛順序に結果が影響されず、また条件によってはand,orの一部を調べるだけで結果を決定する最適化を行なうことができる。
一方後者の最適化を適用するには以下の条件が必要となる。

True 条件が成立した場合
False 条件が成立しなかった場合
Undef 変数が束縛されておらず決定できない場合
Error 変数が条件判断できない型に束縛していた場合

A or B

A \ B	True	False	Undef	Error
True	True	True	True	True
False	True	False	Undef	Error
Undef	True	Undef	Undef	Undef
Error	True	Error	Undef	Error

A and B

A \ B	True	False	Undef	Error
True	True	False	Undef	Error
False	False	False	False	False
Undef	Undef	False	Undef	Undef
Error	Error	False	Undef	Error

A	not A
True	False
False	True
Undef	Undef
Error	Error

図3: 論理演算表

分割前

```
a(A,B) :- (A > 0 -> b(A,C); c(A,C)), d(C,B).
```

分割後

```
a(A,B) :- a0(A,C), d(C,B).
a0(A,C) :- A > 0 -> b(A,C); c(A,C).
```

図 4: マクロ構文の分割例

- マクロの同レベルのボディゴールによって、具体化される変数が条件部にないこと。
- ガードマクロのゴールにさらにマクロがないこと。

もし、マクロと同レベルのボディゴールが存在する場合、マクロを一つのボディゴールとして、他のゴールと共にフォークするようにする。つまり、対象となる定義節をマクロのみをボディゴールとして持つ定義節に分離することによって、マクロ部分に対して最適化が適用できるようにしている。この例を図4に示す。一方、後者の場合、上位のマクロの条件が成立した後では、条件判断の対象は下位のマクロの条件に絞られなくてはいけない。上位のマクロがガードマクロの場合、条件が並列に判断されるので、条件部にある変数の束縛順序が条件分岐に影響してくる。一度サスペンドしてしまうと、この情報は失われてしまうので、結局コミットという機構を使って、この情報を記憶しなくてはいけない。結局、ガードマクロのゴールにさらにマクロがある場合、これを明示的に別の定義節に分離しなくてはいけない。上位のマクロがIf-Then-Elseマクロの場合、条件は逐次判断されるので、変数の束縛順序は条件判断に影響しないので、この必要はない。
図5にこのマクロをFLENGの中間言語にコンパイルした例を示す。なお、この例は一部省略及び手直しを加えている。

```
loop(Cnt, X) :- Cnt > 0 -> folk(Cnt, X),
                           Cnt1 := Cnt - 1,
                           loop(Cnt1, X).

derefAndCheckINT     A1, T1, $SUSPEND, $FAIL
cmpAndBle            T1, 0, $FAIL
setStructure         folk/3, T2
store                T1, [T2 + 2]
store                A2, [T2 + 3]
enqueueGoal          T2
sub                  T1, 1, A1
execute              loop/2
```

図 5: マクロ構文のコンパイル例

5 おわりに

以上で、FLENGの低水準記述言語FLENG−− とその上に追加されたアノテーション及び、それを利用した最適化の方針について概要を述べた。FLENGコンパイラはFLENGをUNIRED Ⅱ等のネイティブコードに落とす前に、UNIRED Ⅱ用に設計されたFLENGの中間言語にコンパイルされる。マシン・インディペンデントな最適化処理の多くはこの中間言語の段階で行なわれ、最終的にこの中間言語から、レジスタ割り当ての最適化などの技法[6]を使って、マシン・ディペンデントな最適化が行なわれる。
現在、このコンパイラはUNIRED Ⅱの製作と並行して、製作されている。
なお、本研究は文部省特別推進研究No.62065002によるものである。

参考文献

[1] Koike,H.and Tanaka,H.: *"Multi-Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64"* Proc. of Fifth Generation Computer Systems, Tokyo, Japan, November 1988.

[2] Nilsson M. and Tanaka H.: *"FLENG Prolog - The Language which turns supercomputers into Prolog machines"* Proc. Logic Programming Conference, 1986, pp.209-216

[3] 中村, 小中, 田中: "並列論理型言語FLENGに基づいたオブジェクト指向言語FLENG++" 日本ソフトウェア科学会, オブジェクト指向計算に関するワークショップ ,WOOC '89, 1989.

[4] Warren,D.H.D.: *"An Abstract Prolog Instruction Set"* Technical Note 309, Artificial Intelligence Center, SRI, 1983.

[5] 島田, 下山, 清水, 小池, 田中: "推論プロセッサUNIREDⅡのアーキテクチャ" 情報処理学会計算機アーキテクチャ研究会 77-2, 1989年7月

[6] Frederick Chow and John Hennessy: *"Register Allocation by Priority-based Coloring"* , ACM SIGPLAN Notices, vol.19, no.6, June 1984, (Proceedings of SIGPLAN 84 Symposium on Compiler Construction), pp.222-232