6Q-1

単一参照アノテーションを用いた論理型言語プログラムの最適化コンパイル

小池　汎平,　田中　英彦
(東京大学 工学部)

1　はじめに

論理型言語において、メモリ使用効率と実行速度を向上させるのための最適化用情報として単一参照アノテーション(SRA: Single Reference Annotation)を提案し、これによって可能となる処理の最適化について検討する。SRAは定義節頭部引数の構造データに付加するアノテーションであり、これを付加すると対応するゴール側構造データにゴール以外からの参照がないことが定義節のコンパイルの際に仮定される。この結果、参照のなくなった構造データ領域を即時回収して新たな構造データを生成する際に再利用できるほか、再利用方法を静的に決めてしまうことにより、元の構造データにもともと書き込まれているのと同じ値を書き込み直すという無駄なデータ転送処理を除去できるという最適化も可能となり、メモリ使用効率と実行速度の両面での性能向上が実現される。SRAによる最適化は、Warrenの抽象命令セットで導入されたレジスタ上のゴールオブジェクトに対する最適化の対象を構造データへ拡張し、処理効率を更に高めるものと見なすことが出来る。

2　単一参照アノテーション

単一参照アノテーション(SRA: Single Reference Annotation)は、論理型プログラムの定義節頭部引数の構造データに付加するアノテーションである。SRAを定義節頭部引数に付加すると、この引数と単一化されるゴール側引数の構造データに対してゴール以外からの参照がないことが定義節のコンパイルの際に仮定され、この情報を元に、3で述べるような最適化が施されて、メモリ使用効率と実行速度の両面での性能向上が図られる。SRAはユーザによって直接指定するほか、プログラムの大域的解析[3]、上位言語からのコンパイル[1]などによって付加される。この意味で、SRAはプログラムの大域的な特性を集約した情報と見なすことが出来る。SRAの使用例を以下に示す。SRAとして‘ (backquote)を用いている。

```
:- mode append(+,+,-).
append(‘[H|X],Y,[H|Z]) :- append(X,Y,Z).
```

プログラム1

```
:- mode p(+,-).
p(‘s(A,B,C,D),s(A1,B,C,D)) :- A1 is A - 1.
```

プログラム2

[0]Optimizing Compilation of Logic Programs using Single Reference Annotation
Hanpei KOIKE, Hidehiko TANAKA
the University of Tokyo

3　単一参照性を利用した実行処理の最適化

SRAを用いて構造データの単一参照性を静的に仮定することにより可能となる実行処理の最適化について考える。ここでは、論理型言語として引数の入出力モードが決まっているもの(モード宣言を併用したProlog、Parlogなど)を対象とする。

3.1　構造データ領域の再利用

単一参照性が仮定された構造データの領域は、単一化が済むとゴミになるので回収でき、新たな構造データの生成に再利用することができる。これにより、メモリ消費速度の低減、アクセスの局所性の向上等がもたらされる。

領域の再利用を同一クローズ内など静的に決定できる制御の範囲内に限定し、この中で生成する構造データのうちで最適化の効果のもっとも大きくなるものを再利用領域に生成するデータとすることを静的に決定すると、回収した再利用領域のアドレスは一時的にレジスタ等に格納しておくだけで済み、フリーリスト等で一般的に管理する必要はなくなる。これにより、動的なオーバヘッドがなくなるとともに、任意長のデータの再利用が容易にできる様になる。また、3.2で述べるように無駄なデータ転送の除去も可能となる。プログラム1及び2はいずれもこのようにクローズ内で回収再利用が行われる例である。

この方法を単独で用いる場合、同一クローズ内で再利用できる領域しか回収できない。このため、回収効率を高めるために、フリーリスト管理を併用することも考えられる。また、SRAを利用してユーザプログラムレベルでフリーリストによる回収領域の管理を記述し、クローズ間にわたる領域の再利用を図ることも可能である。

3.2　無駄なデータ転送処理の除去

定義節の実行において、どの構造データの領域をどの構造データの生成に再利用するかを静的に決定することにすると、新たに生成する構造データの要素のうち、元の構造データにもともと書き込まれているものを改めて書き直す必要はなくなる。また、これにともない書き込むはずだった値を読み出すことも不要になる場合も多い。このようにして、無駄なデータ転送処理を除去することができ、処理の高速化を図ることができる。

例えば、図のプログラム1で、第三引数のリストのcar部に変数Hの値を書き込む必要はなく、第一引数のリストのcar部を変数Hに読み出す必要もなくなる。また、プログラム2では、ストラクチャの長さ、ファンクタ名、変数B,C,Dの読み書きが不要となり、結局、必要な処理は、Aの値を読み出し、型チェックをして1を引き、(必要ならばトレイルをとってか

ら)同じ場所に書き戻すことだけになる。

4 WAMで実現された最適化手法との関係

Warrenの抽象命令セット(WAM)[1]は、論理型言語の様々な最適化手法を用いたコンパイルを可能とするもので、論理型言語実装の標準手法としての地位を確立している。特に、

- ゴールをレジスタ上に構成する。
- 無駄なレジスタ間転送命令を除去する。

の2点は、appendのように末尾再帰で短いループを繰り返すプログラムの処理速度の向上に大きく効いている。

これらの最適化は、SRAの導入によって可能となった最適化の特殊な場合と解釈することが可能である。つまり、ゴールオブジェクトは常に参照が単一なので、レジスタという特殊な領域を再利用して格納することができ、再利用の際にレジスタ間の無駄なデータ転送が除去されるのである。いいかえると、SRAの導入は、WAMの最適化手法の対象をゴールのみから一部の構造データにまで拡張し、処理効率を更に高めるものであると考えることができる。SRAによって可能となる構造データに対する最適化は、参照の単一性が静的に決定でき、なおかつ、同一クローズ内で再利用されなければならないという制約があるものの、レジスタ間データ転送と比べて手間の大きなメモリアクセス命令が除去できるようになるので、最適化できたときの効果はより大きなものであると考えられる。

5 命令セット

SRAを用いた最適化のためにWAMの抽象命令セットを拡張する。このために、構造データを処理するための命令であるget_list命令、get_structure命令、put_list命令、put_structure命令、各種unify命令を、引数の入出力に応じてreadモードまたはwriteモードいずれか、また、再利用を行うかに応じてreuseモードを組み合わせた専用命令に分ける。このうち、reuseモードの命令では再利用領域を指すレジスタを指定するためのオペランドを追加する。また、各unify命令に対しオフセットを追加し、構造データの必要な引数のみをランダムアクセスできるようにする。(従来、unify命令はポインタを更新しながら構造データの引数の順に実行されたが、ペナルティなしでポインタ+オフセットのアドレシングモードによるメモリアクセスができるプロセッサの場合、抽象命令のレベルでunify命令にオフセットを追加しておくことは、ポインタの更新が不要になる、命令の順序を入れ換えて最適化する可能性を広げることが出来る、等の点でも有利である。)

新たに追加される抽象命令は、

```
reuse_read_get_list areg,rreg
reuse_read_get_structure functor/arity,areg,rreg
reuse_write_get_list areg,rreg
reuse_write_get_structure functor/arity,areg,rreg
reuse_write_get_same_structure areg,rreg
reuse_put_list areg,rreg
reuse_put_structure functor/arity,areg,rreg
reuse_put_same_structure areg,rreg
reuse_read_unify_variable var,rreg,offset
reuse_write_unify_variable var,rreg,offset
areg - argument register
rreg - reuse area register
```

等である。(unify命令は一部のみ挙げた。)

6 コンパイル例

上記の抽象命令セットを用いて、プログラム1をコンパイルすると次のようになる。(unify命令のオフセットは、0でリストのcar部を、1でcdr部を指すものとしている。)

```
reuse_read_get_list a1,r4          % append('[H|
reuse_read_unify_variable a1,r4,1 % X],Y,
reuse_write_get_list a3,r4         % [H|
reuse_write_unify_variable a3,r4,1% Z]) :-
execute append/3                   % append(X,Y,Z).
```

これをネイティブコードに展開後、命令の入れ替え等によって更に最適化を施すと、最終的に破壊的書き換えによるリストの結合(lispのnconc)と同等の高速なコードが得られる。

7 おわりに

論理型言語において、メモリ使用効率と実行速度の向上を図る最適化のための付加情報として、単一参照アノテーション(SRA)を提案し、これによって可能となる最適化技法について検討した。今後の課題としては、

- SRAを有効利用するコンパイラの作成[2]
- 様々な種類のプログラムについてのSRAの有効性の確認とその効果の評価
- プログラムモジュールの大域的解析に基づくSRAの自動付加(Reference Count Inference)[3]
- 上位言語からのコンパイルにおけるSRAの利用可能性の検討[4]

等が挙げられる。

参考文献

[1] Warren, D.H.D., "An Abstruct Prolog Instruction Set", Tech. Note 309, SRI International, 1983.

[2] 下山, 島田, 小池, 田中, "FLENGコンパイラとその抽象化コード", 本大会.

[3] 小中, 田中, "Committed-Choice型言語FLENGのタイプ/モード推論", 本大会.

[4] 中村, 田中, "並列オブジェクト指向言語FLENG++の実装", 本大会.