人　工　知　能　80－9
記　号　処　理　63－9
（1992. 1. 17）

Prologのデータベース操作述語の最適化について

矢野稔裕　瀧口伸雄　小谷善行
東京農工大学工学部電子情報工学科
コンピュータサイエンスコース

Prologのデータベース述語などは、そのプログラム中での利用形態がわかれば、非常に効率的なプログラムコードを生成できるはずである。我々はその利用形態を、コンパイラに対して、「宣言構文」を記述することによって行う、という方式をとる。宣言構文はProlog言語に追加され、それを解釈することで処理系は、最適化に利用できる新たな情報をプログラマから受け取ることが可能になる。

ここでは第一に、述語の扱われ方を分類する。第二に宣言すべき対象を決定し、宣言構文を設計する。第三にその情報を元にして新しく可能になる最適化技法を検討する。具体的には、動的述語とその操作に対する最適化が新たに可能になる。それらの最適化技法はWAM命令セットの拡張として設計される。

Optimization for clause database operation of Prolog

Toshihiro YANO, Nobuo TAKIGUCHI, Yoshiyuki KOTANI
Department of Computer Science
Tokyo University of Agriculture and Technology
2-24-16 Nakacho Koganei Tokyo 184 Japan

It may be possible to generate very efficient compiled codes from Prolog database predicates, if the compiler knows the usage of the predicates. We employ a method of Prolog program optimization by giving "declaration syntax" for describing the usage. It is added in Prolog syntax, and enables compilers to receive additional information for optimization, by interpreting the syntax.

Here, first we classify how predicates are dealt with, second, we design the declaration syntax by determining what should be declared. And finally we discuss newly enabled optimization techniques by the information. Dynamic predicates and operations on them can be optimized newly. The optimization techinques are designed as an extention for WAM instruction set.

1. はじめに

Prolog の動的述語、すなわちデータベース操作述語 assert や retract によって実行時に操作される可能性のある述語は、実用的プログラムでは少なからず使用されており、Prolog における大域変数的な役目や、プログラムを見やすくする役目を果たしている。また、例えば知識処理分野では、本質的に不可欠な機能となっている。

しかしながら、動的述語に関して最適化を施すことはあまり考えられていない。Prolog が第一義的に論理型言語であって、動的に述語を操作することは“不純”であり、あまり着目されなかったことも理由の一つかもしれない。

商用の処理系では独自に効率のよい大域データ専用述語などの拡張を行って、節データベース操作の負担を減らそうとしている。しかしそのような拡張述語を使用すると、DEC-10 Prolog に準じたプログラムであれば比較的互換性が高いという Prolog の利点の一つを捨てることになる。

本稿では、まず Prolog 言語に追加する新たな宣言構文に関して述べる。これは既存の処理系との互換性を保っている。この宣言構文を解釈できる処理系は、最適化に利用できる新たな情報をプログラマから受け取ることが可能になる。その宣言を用いたいくつかの最適化技法を検討し、WAM 命令セット[1] の拡張として示す。

2. 宣言構文

Prolog 言語に新たに次の宣言構文を追加した言語仕様を考えた。これは DEC-10 Prolog に準じた多くの処理系の上位互換の仕様となる。

・述語特性宣言
・未然定義宣言

これらの宣言を行うことによって、プログラマは処理系に対して最適化に役立つ情報を与えることが可能になる。これは、処理系から強要されるわけではなく、宣言のない述語には従来の手法が適用されるだろう。

2.1. 述語特性宣言

述語特性宣言は、述語 property によって記述され、プログラムで定義される述語、および実行時に登録される述語の特性を処理系に伝える。この宣言で、引数の入出力方向の情報、引数の型宣言に相当する情報、動的述語に関する特性などを記述する。処理系はこれらの情報を利用して効果的なコンパイルドコードの生成を行う。ただし、処理系が単に無視することもありうる。その構文を BNF によって図2.1 に示す。

この述語特性宣言によって表せる内容について次に説明する。

2.1.1. 仮引数名

述語の引数に対する仮引数名を表記する。これは、property/2 の第 2 引数に記述できる述語単位の特性リストの中で引数を名前で指定したり、処理系からのメッセージを読みやすくするのに使われる。Prolog では処理系の読める形式で引数名を表記できなかったが、これによって可能になる。

2.1.2. 引数の型

引数の型を表記できる。この情報でインデキシングの効率化が行えるようになる。

複数の型名をリストにすることで、受け渡しの起こる可能性のあるすべての型を表明できる。このとき、並べた型しか受け渡しが起きないと解釈される。term は任意の型を意味する。

2.1.3. 動的述語宣言

動的述語であるか静的述語であるかを処理系に明示する。static または dynamic で指定するが、動的述語であることが前提である特性を同時に指定したときは dynamic は省略できる。

2.1.4. 呼出し形式 (call)

述語の呼出しが、プログラムに定義された述語の本体において、サブゴールとして明示されたものだけである (explicit) か、そうでない

```
<述語特性宣言> ::=
    property( <pred> ) |
    property( <pred>, <prop> ) |
    property( <pred>, [ <proplist> ] )
<proplist> ::=
    <prop> | <prop>, <proplist>
<pred> ::= <onepred> | [ <predlist> ]
<predlist> ::=
    <onepred> | <onepred>, <predlist>
<onepred> ::= <name> |
    <name> / <int> |
    <name> ( <arglist> )
<arglist> ::= <arg> | <arg>, <arglist>
<arg> ::= <io> |
    <io> ( <name> ) |
    <io> ( <name>, <type> ) |
    <io> ( <name>, <type>, <prop> ) |
    <io> ( <name>, <type>, [<proplist>] )
<io> ::= in | out | io
<type> ::= <onetype> | [ <onetypelist> ]
<onetypelist> ::=
    <onetype> | <onetype>, <onetypelist>
<onetype> ::=
    integer |
    integer( [ <intlist> ] ) |
    integer( <int>, <int> ) |
    atom |
    atom( <enumlist> ) |
    struct |
    struct( <name> ) |
    struct( <name> / <int> ) |
    struct( <name> ( <arglist> ) ) |
    list |
    list( [ <arglist> ] ) |
    list( [ <arglist> | _ ] ) |
    list( [ <arglist> | <arg> ] ) |
    float |
    term
<prop> ::=
    <switch> static |
    <switch> dynamic |
    excl |
    excl = <enumlist> |
    call = explicit |
    call = implicit |
    priority = space |
    priority = time |
    max = <int> |
    order = bag |
    order = stack |
    order = queue |
    interval = normal |
    interval = short |
    interval = long |
    <switch> unit_clause |
    <switch> known |
    <switch> unknown
<switch> ::= + | -
<intlist> ::= <int> | <int>, <int>
<enumlist> ::= <name> | [ <namelist> ]
<namelist> ::= <name> | <name>, <namelist>
<name> ::= <アトム印字名>
<int> ::= <整数値表記>
```

図2.1. 述語特性宣言

か（implicit）を表明する。

前者の場合には、処理系がプログラムを静的に解析して得られる情報の項目が増えると同時に、プログラマの宣言による表明との矛盾のチェック可能な項目も増加してプログラマを助ける。

後者の場合は、変数を介して間接的に呼び出されること、例えば

…, P=..[pred, arg], call(P), …

のような呼出しや、動的に定義された節から呼び出される場合である。このようなときには、適用できない最適化技法がある。

なお動的述語の場合は、操作述語の引数として述語が明示されるか否かをも意味する。すなわちexplicit の場合、assert や retract の引数が複合項で、abolish の引数が述語名と整数値であることを保証する。

2.1.5. 優先順位（priority）

述語の最適化が記憶の節約を指向すべき（space）か、処理速度の向上を指向すべき（time）かを表明する。

2.1.6. 排他性（excl）

述語特性リストに単独で使われたとき、動的述語を構成する節に、同じものの重複して定義される瞬間がありうるか否かを表す。

引数の特性リストに使われたときか、述語特性リストで仮引数リストを伴って使われたとき、

それらの引数に関して、同じ入力値で複数の節と単一化成功する可能性がないことを表明する。

2.1.7. 節の最大数（max）

動的述語が、プログラム実行中に高々いくつの節によって構成されるかを表す。

これが与えられたとき、静的に記憶割り付け可能となって実行時の効率に貢献できる場合がある。また、これが 1 の時に特に可能になる最適化手法もある。

2.1.8. 登録順序（order）

節を登録する際の規則性に関して表明する。

(1) bag

asserta, assertz を使用して登録しないことを表明する。

(2) stack

asserta のみによって登録し、最後に登録した節から順番に消去することを表明する。

(3) queue

assertz のみによって登録し、最初に登録した節から順番に消去することを表明する。

2.1.9. 述語操作頻度（interval）

動的述語の操作の頻度を表明する。特に頻繁に登録や消去が発生する述語（short）か、滅多に発生しない（long）かを指定する。処理系は前者の場合、述語操作自体の効率を重視し、後者の場合操作された述語の実行効率を重視する。デフォルトではどちらでもない（normal）。

最適化優先順位が同時に指定され

[priority(time),interval(long)]

の場合、処理系は、述語レベルのコンパイラを実行時に起動し、インデキシングコードの再生成を行う。なお

[priority(time),interval(short)]

の場合は、処理系を悩ませるかもしれない。

2.1.10. 単位節

動的述語が単位節だけで構成されることを表明する。

2.1.11. 既知述語（known）

動的述語を構成しうる節が、プログラム中にすべて表されていることを表明する。つまり、通常の述語定義と次に述べる未然定義宣言の利用で、すべての節が静的に判明することを意味する。

2.2. 未然定義宣言

未然定義宣言は、プログラムロード直後には定義されていないが、実行時に定義されうる節をあらかじめ表明するものである。述語 preassert/1 の引数に節を示して記述する。

```
preassert((condition(red):-stop)).
```

この宣言によって、コンパイル時に前もって節のコードを生成したり、インデキシングのコードを生成しておくことが可能になる。

3. 新たな最適化技法

拡張された宣言構文によってプログラマから適切な情報を受け取ることができるようになると、

- 最適なインデキシング引数の選定
- 無駄なコード生成の抑制

などの、従来からモード宣言や動的述語の指定によって可能であったものに加えて新たに、

- 配列化インデキシング
- 未然のコード生成
- 動的インデキシング

などが可能になる。

3.1 配列化インデキシング

WAM のインデキシングでは、整数データと記号アトムは定数として分類され、同じハッシュ表を用いて入口番地を得ている。

しかし整数と記号アトムの内部表現の違いを利用できる場合がある。アトムの内部表現は一般的に規則性の低い整数値となることが多い。これは記号表の管理の方法にもよるが、例えば

ハッシュ法を用いて管理しているとき、印字名のハッシュ関数値をそのアトムの内部表現値とするような実現法がとられる場合である。よって、インデキシングの対象となる節頭部の引数が記号アトムであるとき、それらは不連続な整数値群となっている。

一方、節頭部の引数に整数値が現れるときは、それらの値が隣接していることが少なくない。特に、手続き型言語での大域変数の配列に相当する利用形態で、連続し重複のない整数列がよく現れる。

```
array(1,…).
array(2,…).
array(3,…).
  …
```

導入された宣言構文を用いてプログラマがこの特徴を次のように記述したとき、処理系はこの引数によるインデキシングをより効率よく処理できる。

```
property(array(
    in(index,integer,excl),…),
    [static]).
```

すなわちハッシュ関数を使用せず単なる配列アクセスによって分岐番地を得ることができる。これでハッシュ関数の計算、衝突時の処理が不要となる。これを実現するため、拡張 WAM 命令

```
switch_on_integer size,table,base
```

を用意した。

これは、引数レジスタの整数値を i とすると、i - base > size のときバックトラックし、そうでなければ i - base を添字として表 table を引いて分岐する。

インデキシング対象の引数に記号アトムと整数が混在する場合は、次のように宣言する。

```
property(array(
    in(index,[integer,atom],excl),…),
    [static]).
```

このとき WAM インデキシング命令 switch_on_term の替わりに、整数の場合の分岐先を追加した拡張命令を使用する。これで整数の場合には高速化され、アトムの場合にはヒット率が向上する。

なお、頭部に現れる整数の順序は無関係である。また、完全に詰まっている必要もなく、多少の欠番のあるときにもこの手法が適用される。

4. 動的述語の最適化

ここでは、DEC-10 Prolog に準じた処理系に備わる次の組込述語群

assert/1　　asserta/1　　assertz/1
retract/1　　abolish/2

と、先に述べた宣言構文による記述とでどのような最適化が可能になるか述べる。

4.1. 動的インデキシング

WAM のインデキシングコードは、コンパイル時に述語を静的に解析して生成する。動的述語に対して適用するには、実行時に述語コンパイラを起動することになってしまい、一般には負荷の大きい処理である。

しかし適切な情報を与えれば、動的述語にインデキシングを適用できる場合がある。例えば、アトムと項の対応表を構成する次の宣言を考える。

```
property(table(in(key,atom),
                io(data,term)),
    [dynamic,max=N,
    call=explicit]).
```

節の最大数 N （実際は整数値が表記される）が与えられているので、処理系はインデキシング用のハッシュ表の大きさを適切に決定して記憶割付しておくことを静的に行える。

この場合入力引数の型が一つに指定されているので、ハッシュ表は switch_on_constant 用だけでよく、switch_on_term 命令と節コードの try_me_else 系命令での連結も必要ない。

節の追加は命令

```
entry_constant n,hashtable
```

によって行われる。WAM では、キーの同じ複数の節を登録するときは、ハッシュ表と節コード

の間に try 系命令の並びをおいて非決定的に実行させている。try 系命令は連続した記憶領域を要求するので、実行時に再割り付けが頻繁に起こるとき、記憶の効率的な管理に不利である。そこで新たに

```
try_link    Lme,Lelse
retry_link Lme,Lelse
trust_link Lme,fail
```

を使う。これらは Lelse を代替節アドレスとして選択点に書き込んで Lme を実行する。

入力引数に excl が指定してあれば同じキーは現れないので、これらの命令の割付は発生せず決定的な実行だけになる。

なお、特性 unit_clause の追加指定されたアリティ 1 の動的述語ならば、次の専用命令を使ってハッシュ表を小さくできる。

```
lookup_constant n,hashtable
```

これは大きさ n の表 hashtable を引いて見つかれば次の命令に進み、そうでなければバックトラックする。この場合、次の命令とは proceed となる。

配列化インデキシングが適用できる場合は、実行時のハッシュ表の操作は行う必要はない。

なお、クローズドハッシュ法を使うときには空きと消去済の区別が必要だが、実現にはデータタグ用ビットを利用できる。

4.2. 整数型変数

次の宣言は、整数型の引数を持つ単位節が高々一つ定義されるような動的述語であることを示す。

```
property(var(io(var,integer)),
    [dynamic,max=1,unit_clause]).
```

このような単純な動的述語は図4.2 のようにコンパイルできる。

ここで命令

```
get_global_integer var,Vn
```

は var 番地の整数値の内容とレジスタ Vn との単一化を行う。

```
put_global_integer var,Vn
```

```
Var:  get_global_integer Int_var,A1
      proceed
Int_var:  <integer_value>
assert_var: asserta_var: assertz_var:
      put_global_integer Int_var,A1
retract_var: abolish_var:
      reset_exist Int_var
```

図4.2. 述語 var のコード

は var 番地にレジスタ Vn の整数値を書き込む。

```
reset_exist var
```

は番地 var のタグ用ビットを操作して、現在節が定義されていないことを示す。

この述語の操作は、すべて単純な代入処理に帰着させることができる。

なお、assert_var 以降のラベル群は、各組込述語の中から、特別な最適化の施された述語の場合に分岐して来る。

4.3 未然定義宣言の利用

実行時に登録される節を前もってプログラマが知っている場合がある。そのような節が未然定義宣言によって処理系に伝えられたとき、あらかじめインデキシングコードを生成しておくことが可能になる。また、実行時には、あらかじめ割り付けられた領域だけを操作するので、塵集め処理の起動に影響しない。特に繰り返し処理の内側にあって頻繁に呼び出される場合などは効果がある。

4.3.1. 状態変数的な場合

次の動的述語は三つの節のうち高々どれか一つが有効になり、状態変数のように使われている。

```
property(stat(in(color,atom)),
    [dynamic,max=1,known]).
stat(blue).
preassert(stat(yellow)).
preassert(stat(red)).
```

このとき図4.3.1 のようにコンパイルされる。

```
Stat:   switch_on_status Var,Table
Var:    <integer_value>
Table:  L1
        L2
          …
L1:     <clause1>
L2:     <clause2>
          …
Assert_stat: Asserta_stat: Assertz_stat:
        call Get_id_stat,N
        put_global_integer A1,Var
        proceed
Retract_stat: Abolish_stat:
        reset_exist Var
        preceed
Get_id_stat:
          …
```

図4.3.1. 述語 stat のコード

コンパイル時に、述語を構成する節に通し番号が付けられ、その順序でそれぞれの節コードの入口番地を並べた配列が作られる。命令

```
switch_on_status Var,Table
```

によって、Var に格納されている整数値を添字として表 Table を引いて分岐する。

この述語では非決定的実行のためのコードは不要である。

get_id_stat には、ちょうど次のようなプログラム

```
get_id_stat(stat(blue),1).
get_id_stat(stat(yellow),2).
get_id_stat(stat(red),3).
```

がコンパイルされた場合のコードが置かれ、節の認識番号を返す。

4.3.2. 集合的な場合

次のように同時に複数の節が定義されうるときを考える。

```
property(set(io(lang,atom,excl)),
    [dynamic,known]).
set(prolog).
set(smalltalk).
set(c).
preassert(set(awk)).
```

このとき、動的述語 set のコードは図4.3.2 のように生成される。

```
Set:    <indexing_code>
NDet:   execute Tlb+0
Last:   execute Tlb+2
Tlb:    try_link    C1,Tlb+1
        retry_link  C2,Tlb+2
        trust_link  C3,fail
        nop_link    C4,fail
C1Det:  if_exist Tlb+0
C1:     <clause1>
C2Det:  if_exist Tlb+1
C2:     <clause2>
          …
Assert_set: Assertz_set:
        call Get_id_set,N
        add_last Last,Tlb
        proceed
Asserta_set:
        call Get_id_set,N
        add_first NDet,Tlb
        proceed
Retract_set:
        call Get_id_set,N
        del NDet,Tlb
        proceed
Abolish_set:
        clear Tlb,N,NDet
        proceed
Get_id_set:
          …
```

図4.3.2. 述語 set のコード

あらかじめインデキシングコードが生成され、そこからの分岐先である節コードには、命令

if_exist addr

が前置きされる。これは番地 addr に命令 nop_link があればバックトラックし、そうでなければ次に進んで節を実行する。

try_me_else 系命令は使用されず、先に述べた try_link 系命令と nop_link をならべて配置した try_link ブロックが形成される。命令

nop_link Lme,Lelse

はダミーであり、対応する節が現在無効になっていることを示す。

インデキシング引数が変数で呼び出されたとき、番地 NDet を経由して try_link ブロック内に分岐して非決定的に実行される。

NDet に置かれた execute 命令は asserta が呼ばれて先頭に節が追加されるときに、命令 add_first によってオペランドが操作される。この次に置かれた execute 命令はダミーであり、assertz の処理を行う命令 add_last が、現在の最後の節を線形探索せずに得るためのポインタである。

述語 abolish の処理は、clear 命令によって try_link ブロックを nop_link で埋める処理になる。

なお try_link ブロック内の第 1 オペランドは固定であり操作しない。

この手法によって、動的にインデキシングコードを修正することなくインデキシング可能となり、同時に非決定的処理の性能は静的な場合と同等である。述語の操作は、小さな領域内のリンク構造の更新だけで済む。

5. まとめ

Prolog 言語に追加する新たな宣言構文を設計した。これを追加した言語仕様は

・DEC-10 Prolog に準拠した処理系の上位互換言語である

・従来の Prolog プログラムにはコメントとしてしか表現できなかった情報を、処理系に解釈できるように形式化した

という特徴をもつ。

これによって

・既存のプログラムに適切な宣言を追加することで、従来不可能であった最適化が適用可能になる。特別な述語を使用しないので、プログラムの互換性を保ったまま処理できる

・プログラムの可読性、保守性、信頼性の向上に貢献できる。

また、配列化インデキシングなどの新たに可能になる最適化技法について、WAM 命令セットの拡張の形で設計した。特に動的述語に関しては、例えば整数大域変数のように利用しているときに、その意図をプログラマが適切な宣言で処理系に与えると、単純な代入操作に還元できることを示した。

参考文献

[1] Warren, D.H.D.: An Abstract Prolog Instruction Set, SRI International Technical Note, No. 309 (1983)

[2] T.P. Dobry: A High Performance Architechture for Prolog, Kluwer Academic Publishers (1990)