論理型プログラム言語における述語特性宣言とその最適化技法

矢野稔裕† 瀧口伸雄† 小谷善行†

Prolog では，変数，配列，リストやフラグなどの扱いが異なる諸対象を，述語の動的書換えという単一手段で表現する手法がよく用いられる．これを均一的に処理してしまうのが，Prolog などの論理型言語における非効率の一因である．そこで，個々の述語のプログラム中での扱いがわかれば，おのおの別な形にコンパイルして，効率的なコードを生成できるはずである．われわれは，プログラマが明示的に述語の扱いを記述する方式を提案する．その記述形式を「宣言構文」と呼び，それを解釈してコンパイラは，最適化に利用する情報を受け取る．本手法の設計に当たり，第一に節がプログラムとして実行される際の処理系における述語の扱われ方に応じて，宣言すべき対象を決定し，宣言構文を設計した．それらは述語の特性記述と，動的述語の型の準備的定義である．第二にその情報をもとにした．従来の手法にない最適化技法を検討した．動的述語自体の実行とその述語の操作処理に対して適用可能な最適化技法を開発した．たとえば，述語に対し大域変数的な単純な利用のみを行うことを宣言すれば，処理系は単なる代入操作に還元したコードを生成する．それらの最適化技法を WAM 命令セットの拡張として設計し，効率評価を行った．その結果，本研究の最適化技法を施した動的述語で，実行効率や追加削除効率は従来比で数倍の性能向上が見込めることがわかった．

A Method of Optimization by Property Declaration of Predicates in a Logical Programming Language

Toshihiro Yano,† Nobuo Takiguchi† and Yoshiyuki Kotani†

Each object in Prolog such as a variable, an array or a list, should be dealt with in different methods. However, all objects are described in the single way that is to rewrite clauses dynamically, unless it can be represented as an argument of some predicate. One of the reasons why performance of Prolog programs is not good is that such objects are processed uniquely in the Prolog processing system. Thus, the system may be able to generate efficient codes, and to compile each predicate to various styles, if it knows how to deal with the predicates. We propose a method called "declaration statements" by which a programmer describes how to use predicates explicitly, and informs it to the Prolog compiler. Declaration statements are added to Prolog syntax, and generate information for optimization. Here, depending upon how to have dealt with the predicates, the declaration syntax is designed by determining what should be declared. In addition, we discuss newly enabled oatimization techniques by this declaration. Thus we develop applicable optimization technique for running and rewriting dynamic predicates. For example, a single assignment code is generated, if one declares the usage of a global variable to a predicate. These optimization techniques are designed as extensions for WAM instruction set. Then, we evaluate their performances. As a result, optimized predicates are estimated to have several times faster speed on running, asserting and retracting than ordinary codes.

1. はじめに

大きい Prolog プログラムを観察すると，述語の扱われ方にさまざまなものがあることがわかる．たとえば，もっとも簡単なものとして，引数のない述語で，一つの事実節が assert されているかどうかによって，フラグの役割を果たす場合がある．また，配列と同様の使われ方をする述語がある．その場合，内容が書き換わるものも書き換わらないものもある．

このように，Prolog では，手続き，配列，一次元リスト，フラグなどに相当する，扱われ方が異なる諸対象を，述語の動的書換えという均一な手段で表現している．動的書換えとは，節操作述語の assert や retract で実行時に述語を操作することを指す．もちろんこうした対象を述語の引数中でリストや複合項として扱うこともあるが，限界があり，多くの処理を述語の動的書換えに負っている．実際，動的述語（動的に書き換えられる述語）は，実用的プログラムでは少なからず使用されている．例えば知識処理分野では，

† 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
Department of Computer Science, Tokyo University of Agriculture and Technology

本質的に不可欠な機能となっている．すなわち，意味ネットワークやフレーム，プロダクションシステムにおけるワーキングメモリなどの個別的知識は，Prologの事実節（単位節）で定義され，推論過程で変更，追加，削除が動的に行われる．

この述語動的書換えという単一手段に依存していることがPrologなどの論理型プログラム言語の効率の悪さのひとつの要因になっている．今日まで，論理型プログラム言語の効率化の研究は多い．しかし静的述語を対象とするものが中心であった[6)~12)]．その上に動的に変化する述語のさまざまな扱われ方に対応する効率化は論じられていない．

逆にいうと，個々の述語のプログラム中での扱われ方がわかれば，おのおの別な形にコンパイルして，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．動的述語を含め，述語の扱われ方は，プログラムの実行中，ほとんど例外なく固定的であるので，それが意味をもつ可能性は高い．

しかし，このような動的述語に関して最適化を施すことはあまり考慮されなかった．第一の理由には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がその理念の外にあったという点がある．しかし上記のように動的述語は機能として必須である．

第二の（そして最大の）理由はその最適化が現状のままでは困難であることである．これは，Prologの言語仕様では，プログラムを構成する「節」がプログラムが扱うデータ構造「項」と本質的に同一であることに起因する．Prologの効率化はWAMモデル[1)]に代表されるように，データでもプログラムでもある「節」を，単なるプログラムである命令列にコンパイルすることで実現するという手法に基づく．そうすると，実際のプログラムにおいてどんな形式のどんなコンパイル対象物を生成すべきかは，コンパイラはあらかじめプログラムから予期できない．結局，効率的なコードの生成は不可能である．

そこでわれわれは，コンパイラがその推論をするのではなく，逆に，プログラマが積極的にその種の情報を明示するという方向に発想を転換した[3)]．そうした作業はプログラマにとっては負担と思われるかもしれない．しかしプログラマはそうした情報をすべて知っているはずであり，知的な手間は少ない．むしろ「整った」表現形式の一部として利用できると考える．

動的述語の情報を明示すれば，動的述語を最適化することに対して非常に強い武器を持つことになる．ある述語がいつも一定の形にassertされるならば，あらかじめそれをコンパイルしておくことも可能になる．

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[4),5)]．これはプログラマ側から処理系に情報を伝えるという意味ではわれわれの考えと一致している．しかし単に拡張述語を使用するのでは，DEC-10 Prologに準じたプログラムの互換性の高さというPrologの利点の一つを捨てることになる．述語のモード宣言や述語のdynamic/static宣言の考え方は，われわれの手法に近い．ただしモード宣言は述語の引数の方向性を，dynamic/static宣言は動的述語かどうかを，処理系に示しているだけである．われわれはそれらも含め，多様な動的述語の扱われ方をシステムに伝えることを考える．

われわれは，まずProlog言語に追加する新たな形式として，「宣言構文」というものを提案し，それに上記の情報を表現させることをめざす．これは既存の処理系との互換性を保つものとする．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．さらに，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. 宣言構文とその機能

Prolog言語に新たに二つの宣言構文を追加した言語仕様を設計した．すなわち，第一のものは，述語の性質を宣言する「述語特性宣言」であり，多様な種類が用意される．第二に，動的に定義される述語の形を前もって示す「未然定義宣言」がある．

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．このことは処理系から強要されるわけではないため，宣言のない，あるいは不足したプログラムでは，処理系は従来通りの手法によって処理する方式とする．

二つの宣言構文とも，ソースプログラム中の節定義の形式として表記する．こうすることで，この宣言構文を解釈できる処理系は，それを宣言として解釈し，解釈できない処理系では通常の節の定義として解釈し，実行されない無駄な節として定義されるだけである．

したがって，これらの宣言を書き加えたプログラムであってもDEC-10 Prologに準じた多くの処理系で問題なく読み込み，処理される．

2.1 述語特性宣言

述語特性宣言は，プログラムで定義する述語や実行時に登録する述語の特性を処理系に伝える．この宣言で，動的述語に関する特性を記述する．また，引数の入出力方向，引数の型宣言に相当する情報も扱う．述語特性宣言の構文を付録に示す．このなかで，われわれが提起し設計した宣言特性を以下に説明する(モード宣言等の，特に新規性のないものの説明は省略した)．

それらは，property/2 述語を用いて表現する．第1引数が対象の述語名で，第2引数は，述語全体に関する情報を記述する．

なお，最後に未然定着宣言の説明を加える．

(1) 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か (explicit)，否か (implicit) を表明する．

前者の場合，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，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ム作成を支援できる．後者は，変数を介した間接的呼出し (たとえば，…, P=.. [Pred, arg], call(P), … といった形) や，動的に定義された節からの呼出しを指す．次の宣言例で，述語 pred/0 は明示的な呼出しだけで呼ばれる．

```
property(pred, call=explicit).
```

(2) 更新形式 (modify)

動的述語を操作する組込み述語 (assert など) の引数として，述語が明示されているか (explicit)，そうでないか (implicit) を表明する．前者の場合，述語操作呼出し部分に，コンパイラがインタプリタと共用である組込み述語の呼出しコードではなく，その動的述語に特化して生成されたコードの呼出し，あるいはそのコードのインライン展開が行える場合がある．後者は，変数を介して間接的に呼び出される場合や，動的に定義された節から操作される場合であり，インライン展開はできない．次の宣言例では，述語 pred/0 の操作は明示的に行われる．

```
property(pred, modify=explicit).
```

(3) 節の最大数 (max)

プログラム実行中に，動的述語がたかだかいくつの節によって構成されるかを表す．

節の最大数が与えられたとき，記憶割付けを静的に行うことが可能となって実行時の効率に貢献できる．また，節の最大数が1のときには，特に有効な意味をもつ．すなわち，そのような述語は決定的な動作しか起きないため，非決定性実行コードの生成が不要となる．

(4) 登録順序 (order)

節を登録する際の規則性に関して表明する．これは述語定義が一次元リストとして用いられるとき，スタックや待行列であるかどうかをあらかじめ宣言するもので，次のパラメータを定めた．

① bag : assert の処理をもっとも効率的なように処理系にまかせる．

② stack : asserta/1 だけによって登録し，retract/1 によって削除するときは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/1 だけによって登録し，retract/1 によって削除するときは最初に登録した節から順番に消去することを表明する．

(5) 単位節 (unit_clause)

動的述語が単位節だけで構成されることを表明する．これがプログラマによって保証されたとき，特別な最適化を行える場合がある．次に宣言例を示す．

```
property(pred/3, unit_clause).
```

(6) 引数の型

引数の型を表記する．整数 (integer)，記号アトム (atom)，実数 (float)，複合項 (struct)，リスト (list) あるいは任意 (term) のどれかを指定する．

整数型のとき，値の下限値と上限値によって範囲を記述できる．また，とりうるすべての値がわかっていればリストにして列挙することもできる．記号アトムのとき，とりうるすべての値がわかっていればリストにして列挙できる．複合項 (またはリスト) に関しては，その引数 (または要素) を再帰的に記述できる．

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，並べた型だけを受け渡しすると解釈される．

次の宣言例では，述語 pred/1 は −10 から 10 までの整数か，“switch” として記号アトム ‘on’ または ‘off’ の入力される複合項 sw/1 をとる入力専用引数 “arg” をもつことを表す．ここで入力専用引数とは，完全にインスタンシエイトされた引数だけが渡されうることを意味する．

```
property(pred(in(arg, [integer(−10, 10),
        struct(sw(in(switch, atom([on, off])))])))).
```

一般に，なるべく引数の型や値が限定されているほうが，最適化に有効な情報となる．

型の情報は，例えば，入力専用で型が単一に限定されている引数がインデクシング引数として処理されるとき，その述語のコンパイルドコードでは，型判別用の分岐命令 switch_on_term 命令を生成しなくてよい，などの効果がある．

（7）引数の特性

引数の特性を指定する機能である．このなかで重要であるのは，引数の排他性（excl）である．排他性の引数とは，その述語において，入力された引数値に対して一つの節としか単一化の成功しない引数である．

排他性引数に値が入力されて呼び出されると，その述語は決定的動作になり一つの節だけが実行される．したがって排他性引数が入力専用引数なら，決定的動作しか起きない述語である．このことがわかれば，処理系は非決定性動作用のコンパイルコード（try_me_else 系命令や try 系命令）を生成しないで済む．

次の宣言例では，述語 pred/1 は排他性の整数型入力専用引数 “arg” をもつことを表す．

```
property(pred(in(arg, integer, excl))).
```

（8）その他の述語特性宣言

このほかにもさまざまな述語特性宣言を設計した．最適化優先順位（priority）は，述語の最適化が記憶の節約を指向すべきか，処理速度の向上を指向すべきかを表明する．述語操作頻度（interval）は，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か，滅多に発生しないかを指定する．known/unknown 宣言は，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か（known），そうでないか（unknown）を表明する．また既存の Prolog 処理系にみられる，dynamic/static 宣言も設計に含まれる．

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宣言構文である．述語 preassert/1 の引数に節を示して記述する．次に宣言例を示す．

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

未然定義宣言の効果的な利用として，known 宣言との組み合わせがあげられる．known 宣言はある特定の述語がすべて既知であることを示す．この二つの宣言により，コンパイル時に前もって節のコードを生成しておくことができ，また可能ならばインデクシングのコードをあらかじめ定義でき，最適化処理を施すことができる．

3. 宣言の効用

宣言構文を設計する目的は，処理系の行う最適化に役立てることである．これに加え，この宣言構文は同時に次のような好ましい副作用をもたらす．もちろん，宣言構文は一般の手続き型言語における宣言と同様の性質をもつ．つまり，宣言は充分注意深く行われなくてはならず，誤った宣言を与えたプログラムをコンパイルすることが致命的な結果を導く可能性があるのは，通常の手続き型言語のコンパイラなどと同様である．

まず，プログラムの可続性の向上が期待できる．プログラムは処理系に有益と思われる情報を，形式化された表記で記述するようになる．これらの情報は，人間がプログラムを読む場合にも当然有益な情報である．

可読性の向上には，コメント構文による注釈も有効である．しかし，コメント構文にはなんら形式が存在しないため，気まぐれな形式によって曖昧さや不正確さが付加される危険性がある．形式化されていることによって，簡潔で正確な表現となる．これはプログラム書式の標準化にもつながる．

また，一般にコメント構文によってプログラムに注釈を与えることは，処理系はそれを単に無視するため，プログラムの内容と注釈の内容の食い違いが発生する危険性がある．形式化された構文を使った処理系による検査は，その危険性を抑える助けになる．

処理系によって解釈可能であると，プログラムから与えられた情報と，宣言以外のプログラムから静的あるいは動的に得られる情報との突き合わせが可能になる．これはプログラムのバグの早期発見に貢献できる．

なお，新たに宣言可能になった情報のほとんどは，Prolog プログラムがプログラム作成時に意識している事項である．これをプログラム完成後に追加するのは苦痛かもしれないが，作成中であれば負担は少ない．さらにつけ加えれば，プログラマが宣言に対して，array のような別名を与えることは可能であり，それにより宣言の簡便性を向上させることができる．また，このようなよく使われる宣言をライブラリとして提供する方法もありうる．

4. 配列化インデクシング

ここで，宣言構文に対応した最適化技法について検討する．

本研究の結果が適用される処理系として，汎用計算機上のインタプリタ処理系を核として，WAM に準じた命令セットを生成するコンパイラをもつシステムを考える．WAMコードはエミュレータで実行されるとしても専用プロセッサで実行されるとしてもよい．

拡張された宣言構文によってプログラムから適切な情報を受け取れるようになると，最適なインデクシング引数の選定や，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

（1）配列化インデクシング
（2）未然のコード生成
（3）動的インデクシング

などの最適化技法が実現可能になる．これらの手法をWAM モデルの拡張の形で示す．すなわち，try_link など 19 個の拡張命令を追加して行う．（1）をこの節で，（2）と（3）を次の節で論じる．

WAM のインデクシングでは，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性質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．したがって，インデク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．

4.1 整数引数による配列化インデクシング

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．例えば次のように，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は，連続し重複のない整数列がよく現れる．

```
a(1,…).  a(2,…).  …  a(10,…).
```

導入された宣言構文を用いて，プログラマがこの述語の特徴を次のように記述したとする．

```
property(a(in(index, integer(1, 10), excl),…),
        static).
```

このとき，処理系はこの引数によるインデクシングをより効率よく処理できるようになる．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処理にコンパイルできる．

これによって次のような利点がある．

（1）ハッシュ関数の計算が不要
（2）ハッシュ関数値の衝突時の処理が不要
（3）使用される表はハッシュ関数に与えられるキーの保持が不要で，分岐先番地の並びだけでよい．

これを実現するため，拡張 WAM 命令 switch_on_integer を新設した．これは引数レジスタの値からベース値を引いた数の番地へ飛ぶもので，switch_on_constant の代わりに使う．

4.2 型の細分化

インデクシング対象の引数に記号アトムと整数が混在する場合を考える．例えば次のように宣言された述語である．

```
property(array(in(index,
        [integer(1, 4), atom], excl),…), static).
```

このとき WAM インデクシング命令 switch_on_term の代わりに，整数の場合の分岐先を追加した拡張命令 switch_on_term_i を使用する．これを用いれば，インデクシング引数が整数の場合には，分岐先で配列化インデクシングが適用されることで高速化され，記号アトムの場合には，通常のインデクシング命令 switch_on_constant のもつハッシュ表から整数の場合の要素が除外されていることで，表のアクセスのヒット率が向上する．

なお，インデクシング引数である整数値の順序は無関係である．また，連続値である必要となく，多少の欠番のあるときにも配列化インデクシングが使える．

5. 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み述語だけによって節データベースを操作していると仮定する．

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

5.1 動的な WAM コード操作の負荷軽減

WAM のインデクシングコードは，コンパイル時に述語を静的に解析して生成することを前提にしている．動的述語に対してインデクシングを適用するには，プログラム実行時に述語単位コンパイラを起動することになってしまい，一般に手間のかかる処理である．それでも，実行時に再コンパイルすることが割の合う操作であれば，それを行えるべきである．このとき，次のような処理を実行時に行う必要がでてくる．

（1）ハッシュ表の操作：節が追加される際，空き

がなくなったときハッシュ表を拡張する操作，つまり一般には記憶再割付けが起こる．

（2）try 系命令の操作：WAM モデルでは，インデクシング対象引数が同じ定数である複数の節を登録するときは，ハッシュ命令と節コードに間に try 系命令の並びを生成して非決定的な実行を処理している．しかし try 系命令は連続した記憶領域を要求するので，このような節の追加は一般に実行時の記憶再割付けが起こる．

WAM のインデクシング命令群は実行時の組替え操作を特に想定していない[1]ため，これらの命令の操作に伴う記憶の再割付けによって，WAM 命令領域が虫食い状態となる．これは記憶利用効率の低下や，ごみ集め処理の起動周期を短縮するなどの悪影響を及ぼす．

そこで，適切な情報を処理系に与えて，動的述語に対するインデクシング適用の負荷を軽くすることを考える．例えば，アトムの対応表を構成する次の述語 table/2 を考える．

```
property(table(in(key, atom), io(data, atom)),
        [dynamic, max=10]).
```

まず，この宣言によって節の最大数 10 が与えられているので，処理系はインデクシング用のハッシュ表の大きさを適切に決定して，あらかじめ記憶割付けしておくことを静的に行えるようになる．

次に，この述語においてインデクシングのキーが重複したとき，try 系命令に代わって try_link，retry_link，trust_link の三つの命令を使用する．

これらの try_link 系命令は，ポインタによりリンク構成を形成する．そのため削除などの動的操作の手間が軽減される．たとえば，try_link 命令は

```
try_link Lme, Lelse
```

という形をしており，1) 新たな選択点を造り，Lelse を代替節番地として書き込み，2) Lme に飛ぶ．

この述語の場合，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 命令や，節コードの try_me_else 系命令による連結も必要ない．

さらに，入力引数が排他的であれば（つまり excl が指定してあれば）同じキーは現れないので，switch_on_constant 命令も生成されず決定的な実行になる．

5.2 単位節のコード縮小

特性 unit_clause を指定してある動的述語ならば，命令 lookup_constant でハッシュ表を小さくできる．

この場合，述語を構成するすべての節が単位節であるため，節本体のコードは proceed 命令だけである．lookup_constant 命令を使用して，各節のための proceed 命令を一つで済ませられる．

loopup_constant 命令は主として次のこと確認して次の命令に進む（確認できない場合はバックトラックする，以下の命令も同様）．

（1）引数レジスタの定数値からハッシュ関数値を計算しハッシュ表を引く．

（2）結果の定数が引数レジスタのものと一致すれば，次の命令に進む．

なお，ハッシュ法のアルゴリズムはクローズドハッシュ法[14]を使うことを想定しているが，必要な空きエントリと消去済みエントリの区別には，データタグ用ビットを採用する．この命令は静的な述語に使える．

5.3 範囲のわかる整数引数

次の述語のように，引数が範囲のわかる整数に限定されるような場合では，配列化インデクシングの適用が可能である．

```
property(man(in(id, integer(1, 99), excl)),
        [dynamic, unit_clause]).
```

このとき lookup_constant 命令の代わりに lookup_integer を用いる．

この命令の参照する表にはキーも分岐アドレスも格納されず，節が定義されているか，未定義かを表す二値情報だけの配列である．この命令により，引数レジスタにある整数が範囲を逸脱しておらず，表にその定義済みマークがついていれば次の命令に進む．

5.4 整数型変数

次の宣言は，整数型の引数をもつ単位節がたかだか一つ定義されるような動的述語であることを示す．これは通常の手続に型言語の大域的整数変数に対応する．

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

このような単統な動的述語 v は**図 1** のようにコンパイルできる．

ここで使用されている設設の命令 get_global_integer，put_global_integer，reset_exist は，特定番地の整数値と節の存在状態を示すフラグとを扱う．このフラグを「存在フラグ」と呼ぶことにする．このフラグは大域整数変数セルのタグを利用している．整数タグの付いているときその節は存在し，そうでないとき節

```
V:        get_global_integer Value_v, A1
          proceed
Value_v: 〈整数セル〉
Op_v:     Assert_v; 動的述語操作分岐表
          Asserta_v
          Assertz_v
          Retract_v
          Abolish_v
Assert_v:
Asserta_v:
Assertz_v:
          put_global_integer Value_v, A1
          proceed
Retract_v:
Abolish_v:
          reset_Value_v
          proceed
```

図 1 整数変数的な述語のコード
Fig. 1 A code of a predicate as an integer variable.

```
S:        switch_on_status Value_s, Table_s
value_s: 〈整数セル〉
Table_s: L1
          L2
          L3
L1:       〈第1節のコード〉
L2:       〈第2節のコード〉
L3:       〈第3節のコード〉
Op_s: Assert_s; 動的述語操作分岐表
          :
Assert_s: Asserta_s: Assertz_s:
          allocate
          call Get_id_s, 0
          put_global_integer Value_s, A1
          deallocate
          proceed
Retract_s: Abolish_s:
          reset_exist Value_s
          proceed
Get_id_s: lookup_id 3. Table_id
          proceed
Table_id: 〈ハッシュ表〉
```

図 2 文字型変数的な述語のコード
Fig. 2 A code of a predicate as a string-type variable.

は未定義であることを示す.

get_global_integer 命令は，その二つの引数（番地とレジスタ）の各内容で単一化を行う. put_global_integer 命令は引数のレジスタの整数値を番地に書き込む. 同時に存在フラグを立てる. reset_exit 命令は引数にある番地の存在フラグを降ろし，現在節が未定義であることを示す.

このように，述語自身の動作および述語に対する動的操作は，すべて単純な代入処理に帰着できる.

図1中の Assert_v 以降のラベル群は，ラベルが示す各節操作組込み述語の中で，述語 v がこの最適化の施されたコンパイルド述語であることを記号表で確認したときの分岐先番地である. つまり記号表には，節ソース項へのポインタ，コンパイルドコードへのポインタに加えて，最適化された述語操作コードの分岐表へのポインタが格納される. これがラベル Op_v を指し，これを経由して各操作コードが実行される.

5.5 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ムが知っている場合がある. そのような節を未然定義宣言によって処理系に伝えたとき，インデクシングコード生成が前もって可能になる. 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，ごみ集め処理の起動に影響しない. 特に繰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.

5.5.1 状態変数的な場合

次の動的述語 s は，三つの節のうちたかだかどれか一つが有効になる状態変数を表現するように使われており，述語を構成しうるすべての節が既知である.

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

このような述語は，すべての節のコンパイルドコードをあらかじめ生成しておき，各節に対して節識別番号を割り振っておく. 実行時には，現在有効になっている節への分岐に，節識別番号による配列化インデクシングを施せる. コンパイルドコードを**図2**に示す. これは図1にハッシュ表操作を加えたものに相当する.

この述語では最大で一つの節しか存在し得ないので非決定的実行のためのコードは生成不要である.

命令 switch_on_status では，引数レジスタの値を i とすると，引数にあるテーブルの番地の分岐表の i 番目の要素の指す番地に飛ぶ. lookup_id は

1) 引数レジスタの複合項の第 1 引数の定数値からハッシュ関数値を計算しハッシュ表を引き，
2) 結果の定数と一致すれば，節識別番号を引数レジスタに格納する.

5.5.2 集合的な場合

次の述語 set は，排他的な記号アトムを入出力引数にもつ既知な動的述語である.

```
property(set(io(lang, atom, excl)),
        [dynamic, known]).
```

```
Set:        〈インデクシングコード〉
            ; 決定的実行のとき C1 Det, … へ
            ; 非決定的実行のとき NDet へ
NDet:       execute Tlb+0   ; try_link 命令を指す
Last:       Tbl+2
Tlb:        try_link C1, Tlb+1   ; try_link ブロック
            retry_link C2, Tlb+2
            trust_link C3, fail
            none_link C4, fail
C1 Det:     if_exist Tlb+0
C1:         〈第1節〉
C2 Det:     if_exist Tlb+1
C2:         〈第2節〉
              …
Op_set:       …
Asserta_set: allocate
            call Get_id_set, 0
            add_first NDet, Tlb
            deallocate
            proceed
Assert_set: Assertz_set:   …
          :
          :
Get_id_set:   …
```

図 3 集合的述語のコード
Fig. 3 A code of a predicate as a set.

```
set(prolog).    set(smalltalk).
set(c).         preassert(set(awk)).
```

述語を構成しうる節はあらかじめすべてわかっており，実行時にはそれらのうちいくつかの節が同時に定義された状態となりうる．このとき，動的述語 set のコードは図3のように生成できる．

述語 set を構成するすべての節のコンパイルドコードがあらかじめ生成される．述語の入口番地には，すべての節が定義されている状態を想定したインデクシングコードが生成される．

インデクシング引数が変数でなく決定的な実行となるとき，一つの節が選ばれて実行されるが，その節コードには命令 if_exist が前置きされている．この命令は，特定の番地での命令 none_link の有無を見て，あればバックトラックする．なければ何もせず次の命令に進む．これによって，その節が実際には未定義のときに失敗させる．命令 none_link に対応する節が現在未定義であることを示し，実行されることはない．

述語が呼ばれたときインデクシング引数が変数であれば，非決定的な実行となる．それを実現するため，先に述べた try_link 系命令と none_link 命令，および後述する nop_link 命令を並べて配置した try_link ブロックが形成される．これらの命令はみな同じ命令長となっており，二つのオペランドを持っている．

第1オペランドは常に節のコードを指して固定であり，操作されることはない．第2オペランドは次に試行されるべき代替節を表しており，これによってつながれた try_link 系命令を実行することで，非決定的実行が行われる．節の登録や削除が実行時に発生したとき，オペコードと第2オペランドを書き換えて述語の構成を更新する．

先の if_exist 命令のオペランドは，この try_link ブロックのどれかの要素を指している．

この述語の動的操作について，asserta/1 を例にして説明する．asserta/1 が呼ばれてこの述語の先頭に節が追加されるとき，add_first 命令によって try_link ブロックが操作される．非決定性実行の際は execute 命令（単なるジャンプ命令）を経由して try_link ブロックに飛ぶ．execute 命令の飛び先を示すオペランドを先頭ポインタと呼ぶことにする．これが，追加される節に対応する try_link ブロックの要素を指すように更新される．新たに指された要素は none_link 命令から try_link 命令に書き換えられ，その第2オペランドはいままで execute 命令が指していた要素を指して，そのオペコードも更新される．この更新は，try_link 命令ならば retry_link 命令へ，nop_link 命令ならば trust_link 命令へと更新される．nop_link 命令は，節が一つだけ定義されているとき使われる命令で，単に第1オペランドの指す命令へのジャンプだけが行われる．登録しようとする節が最初の節であることは execute 命令の飛び先が失敗処理コードを指していることでわかるが，このときだけ nop_link 命令が書き込まれる．

このほか，assertz/1 や retract/1 などの動的操作も try_link ブロックを操作することなどにより同様のしくみで実現される．

本研究によって設計し WAM 命令セットに追加する命令を**表1**にまとめて示す．

5.5.3 最適化の効果

述語特性宣言と未然定義宣言を利用して，動的述語とその操作を単純な操作に還元する最適化手法は次のような特長をもつ．

（1）動的述語のコンパイルドコードを生成できるため，動的述語の実行はインタプリタによる実行に切り替える方式にくらべて高速である．決定性実行処理，決定性実行処理共に静的述語の場合と同等の性能にできる．

（2）動的述語でもインデクシング可能な場合があ

表 1 新設命令の一覧
Table 1 List of extend WAM commands.

add_first	先頭の節の追加
add_last	最後の節の追加
clear	すべての節の削除
del	節の削除
get_global_integer	大域整数変数との単一化
if_exist	節の存在の調査
lookup_constant	定数による表引き
lookup_id	節識別番号による表引き
lookup_integer	整数による表引き
lookup_integr_b	整数によるビット表引き
nop_link	ダミー命令
put_global_integer	大域整数変数への代入
reset_exist	節の無効化
retry_link	代替節再試行
switch_on_integer	配列化インデクシング
switch_on_status	大域整数変数での配列化インデクシング
switch_on_term_i	switch_on_term の改良
trust_link	最後の代替節の試行
try_link	最初の節の試行

り，動的述語のインデクシングをあきらめる方式より有利である．

（3）述語の操作が高速である．コンパイルドコードはあらかじめ生成されており，実行時にはまったく生成せず，特定番地のセルや try_link ブロックのリンク構造の操作だけで実現される．実行時にコンパイル処理する方式に比べ有利である．

（4）述語の操作がごみ集め処理の起動に影響しない．述語操作は小さな特定領域内の更新だけで済む．節ソース項の操作も不要である場合，記憶割付け処理がまったく不要となる．

6. 評　価

本研究によって設計された最適化技法に基づく WAM 命令を検証するため，WAM エミュレータを作成して計測実験を行った．

6.1 実験用プログラム

WAMエミュレータは，C言語によって記述されている．記述の方針は，移植性を考慮することと，フルセットの Prolog 処理系を想定した上で計測実験に必要な部分を実現したサブセットとすることの二点である．後者は，移植性を損なわないように留意してはいるが，同じ計算機上で動作する商用 Prolog 処理系と比較することを考慮したためである．

想定したフルセットの Prolog 処理系は，コンパイラを含むインタプリタであり，Prolog 処理系ではよく採用される普通の構成である．

インタプリタは項として表現された節を解釈実行する．呼び出される述語がコンパイル済みであれば，WAM エミュレータによってコンパイルドコードを解釈実行する．（WAM エミュレータもソフトウェアによって実装されたインタプリタであるが，ここでは「インタプリタ」は Prolog インタプリタを指し WAM エミュレータとは区別する．）インタプリテッド述語のコンパイルド述語の混在は自由にできる．

コンパイラはプログラムを WAM 命令セットによる表現に変換する．コンパイラはプログラムロード時やユーザの対話的な指示によって起動させ，時には動的な述語操作によって起動される．

本計測用プログラムは，WAM エミュレータ，いくつかの組込み述語，WAM コードのロードルーチンによって構成される．インタプリタ，コンパイラは実装されていない．計測対象プログラムは，人手でコンパイルした WAM のアセンブリ表現を内部表現に変換する専用のプログラムによって，WAM コード領域にロードされ実行される．計測対象プログラムとそのロード部分を除く WAM エミュレータは，約4000行である．

6.2 計 測 実 験

計測実験の環境は，計算機は SPARC station，OS は SunOS 4.1 (UNIX)，Cコンパイラは GNU C コンパイラ (version 1.37.1) である．

比較対象として，同じ環境での Quintus Prolog Release 3.1.1 (以下 Quintus と略す) を用いた．Quintus は，静的述語はコンパイルして実行し，動的述語はインタプリタによって実行している．

6.2.1 大域整数変数への還元

次の述語特性宣言を与えた述語 v を，大域整数変数へ還元する技法によるコンパイルコードを計測した．

```
property(v(io(i, integer)),
        [dynamic, max=1, unit_clause]).
v(0).
```

この述語のコンパイルコードを呼び出して引数の整数値を取り出すことを1万回繰り返した（以下の計測も同様）．比較のため静的述語としてコンパイルした場合と，Quintus において静的述語とした場合，動的述語とした場合を計測した（**表 2**）．

本処理系では，動的述語，静的述語の実行性能は同等である．Quintus では，静的述語に比し，動的述語の実行時間が4倍ほどかかる．これはインタプリタが動的述語を実行するためであろう．静的述語の呼び出

表 2 大域整数変数へ還元した述語の計算
Table 2 Time of computing a predicate as a global integer variable.

手法	実行時間
大域整数変数へ還元（本処理系）	0.08 msec
動的述語（Quintus Prolog）	0.04
静的述語（本処理系）	0.08
静的述語（Quintus Prolog）	0.01

表 3 大域整数変数へ還元した述語の操作
Table 3 Time of changing a predicate as a global integer variable.

手法	実行時間
大域整数変数へ還元（本処理系）	0.13 msec
動的述語（Quintus Prolog）	1.02

しについて比べると，本処理系の WAM エミュレータは Quintus に比べ約 8 倍時間がかかる．

次に，この述語の動的操作として消去と設定を計測した．比較のため Quintus でも計測した（**表 3**）．大域整数変数への還元技法を適用した本処理系での述語の操作は，Quintus に比べ約 8 倍高速である．

6.2.2 try_link ブロックによる実現

次の述語特性宣言を与えた述語 set を，try_link ブロックで表現する技法を用いてコンパイルしたコードを計測した．

```
property(
        set(io(i, integer, excl), out(o, term)),
        [dynamic, known, unit_clause]).
set(1, a).      set(2, a).
set(3, a).      preassert(set(4, a)).
```

この述語のコンパイルドコードを呼び出して，すべての節を決定的に順次呼び出すことを繰返した（述語は合計 3 万回呼び出される）．比較のため静的述語としてコンパイルした場合と，Quintus において静的述語とした場合，動的述語とした場合を計測した（**表 4**）．

本処理系では，動的述語，静的述語の呼出し性能も同等である．Quintus では実行時間が 3 倍ほど異なっている．これも動的述語がインタプリタによって実行されるためであろう．この場合の静的述語の呼出しによって比較すると，本処理系の WAM エミュレータは Quintus に比べ約 7 倍実行時間がかかる．

次にこの述語のコンパイルドコードを呼び出して，すべての節を非決定的に順次呼び出すことを繰返した（この述語も合計 3 万回呼び出される）．比較のため静的述語としてコンパイルした場合と，Quintus において静的述語・動的述語とした場合を計測した（**表 5**）．

表 4 try_link ブロックによる述語の決定的計算
Table 4 Time of deterministic computation of a predicate by a try_link.

手法	実行時間
try_link ブロック（本処理系）	0.36 msec
動的述語（Quintus Prolog）	0.14
静的述語（本処理系）	0.36
静的述語（Quintus Prolog）	0.05

表 5 try_link ブロックによる述語の非決定的計算
Table 5 Time of non-deterministic computation of a predicate by a try_link.

手法	実行時間
try_link ブロック（本処理系）	0.17 msec
動的述語（Quintus Prolog）	0.09
静的述語（本処理系）	0.17
静的述語（Quintus Prolog）	0.02

表 6 try_link ブロックによる述語の動的操作
Table 6 Time of dynamic change of a predicate by a try_link.

手法	実行時間
try_link ブロック（本処理系）	0.54 msec
動的述語（Quintus Prolog）	2.82

非決定的な計算の場合には，Quintus では結果が 4 倍ほど異なっている．この場合の静的述語の非決定的呼出しによって比較すると，本処理系は Quintus に比べ約 8 倍実行時間がかかる．

次にこの述語の動的操作について計測した．abolish/2 による節の消去と assert/1 による四つの節の定義を繰り返して計測した．（assert/1 は計 4 万回呼び出される．）比較のため Quintus でも計測した（**表 6**）．

この技法を適用した本処理系での述語の操作は，Quintus に比べ約 5 倍の速度である．

6.3 計測結果の考察

本研究による動的述語の最適化技法を適用した述語の実行効率は，同じ述語の静的な場合と同等の性能であることが実証された．普通，コンパイラを備えた Prolog 処理系であっても動的述語はインタプリタによって実行されるため，コンパイルド述語と同等であることは性能向上を意味する．どの程度の性能向上であるかは，本処理系がインタプリタを持たないため実測していないが，本処理系と同じようにソフトウェアによって構成されたエミュレータによってコンパイル

ドコードを実行していると思われる Quintus において，動的述語の呼出しに対する静的述語の呼出しが3，4倍高速であるという結果を考えると，本処理系でインタプリタを実装したときの性能差も数倍と考えてよい．

次に，本研究の最適化技法を施した述語の動的操作の実行効率に関して考察する．本処理系では，節操作用組込み述語の内部にインタプリテッド述語用の処理が実装されていないため，Quintus の動的述語操作と比較すると，5倍から8倍の性能向上となっている．

コンパイルド述語の実行効率，すなわち，WAM エミュレータの実行効率が，本処理系は Quintus に比べ7，8倍遅い．Quintus が商用処理系としての高度な最適化を行っているのに対して，本処理系は，実験系であり特別な最適化を行っていない．そのことが本エミュレータの遅さの主な原因である．したがって，本処理系を商用処理系と同等の努力によって計算機環境に合わせて最適化したときには数倍の効率向上が見込めると推測できる．すなわち述語の動的操作の効率はさらに数倍向上する．

これらをまとめると，本研究によって導入された動的述語の最適化技法は次のような性能であるといえる．

（1）動的述語でありながら静的述語と同等の実行効率である．これはインタプリタで実行する従来の実現方法と比較して3，4倍の性能である．

（2）述語の動的な操作の効率は，従来の実現方法に比べ控えめに見積もって5倍程度向上する．

これらに加えて，計算結果には現れていないが，ごみ集め処理の起動に影響しないことも大きな特長である．

7. 結　　言

Prolog などの論理言語処理系において，プログラムの定義した述語の利用意図を「明示的に指示する」ことにより，効率化を図る，という方式を提起した．そしてこれに基づき，構文を拡張し，それを利用して処理系に伝えた情報をもとに最適化する方法を設計した．それは「宣言構文」を追加することにより行う．

その設計仕様は，標準的な言語仕様をもつ Prolog 処理系との互換性があるものである．処理系がこれを解釈すれば効率化が図れるし，それが不可能でもコメントとして見ることができ不都合が起こらない．これにより，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．また副次的には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，この構文を利用することで可能になる最適化技法を WAM 命令セットの拡張の形で設計した．これによって次のようなことが可能になる．

まず，動的述語に対していままで困難であった最適化が適用できるようになる．したがって，動的な述語操作の負荷軽減が可能になる．そのなかで，ごみ集め処理の起動を減らすことができるということもある．また大域変数的に用いている述語などの最適化も同じ枠組みのなかで設計した．

動的述語の最適化技法は次のような性能であることが実測値から導かれた．動的述語も静的述語と同等の実行効率をもつ．これは動的述語をインタプリタで実行する従来の実現方法と比較して3，4倍の速度向上となる．そして述語の動的な操作の効率は，従来の実現方法に比べ5倍程度向上できる．

参 考 文 献

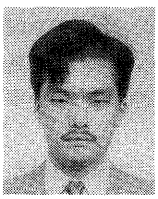
1) Warren, D. H. D.: An Abstract Prolog Instruction Set, *SRI International Technical Note*, No. 309 (1983).
2) Warren, D. H. D.: IMPLEMENTING PROLOG—Compiling Predicate Logic Programs, *D.A.I Research Report*, No. 39, 40 (1975).
3) 矢野稔裕，瀧口伸雄，小谷善行：Prolog のデータベース操作述語の最適化について，情報処理学会記号処理研究会報告，SYM-65 (1992).
4) Quintus Prolog Reference Manual, Quintus Computers Inc., Mountain View (1987).
5) K-Prolog version 4 解説書，アイザック (1987).
6) Dobry, T. P.: *A High Performance Architecture for Prolog*, Kluwer Academic Publishers (1990).
7) Weiner, J. L. and Ramakrishnan, S.: A Piggyback Compiler for Prolog, *Proc. of SIGPLAN '88 Conference on Programming Language Design and Implementation Atlanta*, Georgia (1988).
8) Appleby, K., Carlsson, M., Haridi, S. and Sahlin, D.: Garbage Collection for Prolog Based on WAM, *Communications of ACM*, Vol. 31, No. 6, pp. 719-741 (1988).
9) 小長谷明彦：高速 Prolog インタプリタの構築法とその評価について，情報処理学会記号処理研究会報告，SYM-46 (1988).
10) 中村克彦：Prolog 処理系，情報処理，Vol. 25, No. 12, pp. 1329-1335 (1984).
11) 松本一夫，砥崎賢一，上原邦昭，豊田順一：

PROLOG コンパイラにおける非決定性処理の最適化方式，情報処理学会記号処理研究会報告，SYM-45 (1988).
12) Tick, E.: A Prolog Emulator, *Technical Note*, No. CSL-TN-87-324, Stanford University (1987).
13) 桐山 薫，阿部重夫，黒沢憲一：内蔵型 Prolog プロセッサ IPP の最適化コンパイル方式の提案と性能評価，情報処理学会論文誌，Vol. 29, No. 6, pp. 589-595 (1988).
14) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: *Data Structures and Algorithms*, Addison-Wesley (1983).

付録 述語特性宣言の表現に関する BNF

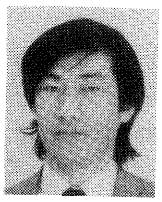
```
〈述語特性宣言〉::=
  property(〈pred〉)|
  property(〈pred〉, 〈prop〉)|
  property(〈pred〉, [〈proplist〉])
〈proplist〉::=〈prop〉|〈prop〉, 〈proplist〉
〈pred〉::=〈onepred〉|[〈predlist〉]
〈predlist〉::=〈onepred〉|〈onepred〉, 〈predlist〉
〈onepred〉::=〈name〉|〈name〉/〈int〉|
  〈name〉(〈arglist〉)
〈arglist〉::=〈arg〉|〈arg〉,〈arglist〉
〈arg〉::=〈io〉|〈io〉(〈name〉)|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

(平成4年12月24日受付)
(平成5年9月8日採録)



矢野 稔裕（正会員）
1965年生．1990年東京農工大学工学部数理情報工学科卒業．1992年同大学院電子情報工学科博士前期課程修了．現在，日本電気(株)に勤務．

瀧口 伸雄（正会員）
1963年生．1985年東京農工大学工学部数理情報工学科卒業．1987年同大学院工学研究科修士課程修了．三菱電機(株)中央研究所を経て，1990年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助手．知識処理方式，自然言語処理・学習などに興味を持つ．人工知能学会，ACM，IEEE 会員．



小谷 善行（正会員）
昭和24年生．昭和46年東京大学工学部計数工学科卒業．昭和52年同大学院博士課程工学系研究科修了．同年東京農工大学工学部数理情報工学科講師．現在同大学電子情報工学科（コンピュータサイエンス）教授．記号処理言語を含むソフトウェア工学および知識処理に興味を持つ．人工知能学会，日本ソフトウェア科学会，電子情報通信学会，認知科学会各会員．