日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Prologへの匿名述語の導入と実装

井出 陽子　向井 国昭

高階述語呼び出しなどに便利な匿名述語の構文および意味を標準の Prolog 意味論に基づいて定義し，SWI-Prolog の上に実装した．匿名述語の呼び出しに標準の単一化がそのまま使えるほか，パラメータを持つ匿名述語も扱える．匿名述語をコンパイル時に通常の述語として展開するため，通常の述語呼び出しと実行効率は同じである．ただし実行時に動的に生成された匿名述語に対しては専用のインタプリタで対応する．使用実験の結果，匿名述語の利便性や記述力の高さ，安定性が確認できたので，標準環境として使えるように匿名述語機能をパッケージ化した．

1 はじめに

本稿の目的は，Prolog に匿名述語を導入することである．導入した匿名述語の特徴を下に挙げる：

1. 単一化： 標準の単一化がそのまま使用できる．
2. 高階プログラミング： 匿名述語は高階述語呼び出しに便利である．匿名述語は論理プログラムの記述力を高める．
3. 効率性： Prolog らしい意味論を保ちつつ，簡潔明快に実装した．実行速度は名前付き述語を定義した場合と変わらない．

本稿では，第 2 節で，まず予備的考察をし，必要な Prolog の述語や概念を説明した後，匿名述語の構文と意味を定義する．第 3 節では，匿名述語の利用例の中から特に重要と思われる，高階述語から匿名述語の利用，DCG からの利用を紹介する．第 4 節では，匿名述語の実装方法を説明する．第 5 節で関連研究，とくに λ 項の Prolog への導入について述べた後，第 6 節でまとめを述べ，本稿を締めくくる．

Introducing Anonymous Predicates into Prolog and its Implementation.

Yoko Ide, Kuniaki Mukai, 慶應義塾大学大学院 政策・メディア研究科, Graduate School of Media and Governance, Keio University.

2 匿名述語の導入

2.1 自由変数と束縛変数

一般に λ 計算において [12]，自由変数が β 変換後束縛されてしまうことがないように β 変換前に束縛変数の名前更新をしなければならない．しかし必要なデータは β 変換で供給すればよいと考えると自由変数は実質不要となる．つまり次の式のように自由変数をいったん束縛して，あとで β 変換で自由変数にもどせばよい．$(\lambda x M)N \Longrightarrow (\lambda x((\lambda a M)a))N \Longrightarrow (\lambda a M[x := N])a \Longrightarrow M[x := N]$. すると，自由変数が現れないのであれば β 変換のための束縛変数の名前変更には Prolog の組み込みの `copy_term` 述語を使ばよいことが分かる．言い換えると，自由変数を明示するように λ 項の構文を導入すれば β 変換をシステムの `copy_term` で実質置き代えることができる．ちなみに自前の β 変換と Prolog 組み込みの `copy_term` では後者がおよそ 10 倍速い．

2.2 前提

この節では本システムに必要な Prolog の概念や組込み述語の説明をする．本システムに必要な Prolog の述語は次のとおり．

- call/N,
- copy_term/2,

- meta_predicate/1,
- term_expansion/2, goal_expansion/2

これらの述語が用意された Prolog システム系ならば，匿名述語を動かすことが可能である．

Prolog の基本的なデータ構造は項である．Prolog の項とは，定数，変数，複合項のいずれかである．リストは'.' を関数子名とする次数 2 の複合項である．以下，$a_1,\ldots,a_m,t_1,\ldots,t_n$ は Prolog の項を表す．$L_1,\ldots,L_n$ は Prolog のリストを表す．

まず，高階述語 `call/N`，`maplist/N` について説明する．`call(A)` は `A` をゴールとして扱い，それを呼び出す．`call`$(f(a_1,\ldots,a_m),t_1,\ldots,t_n)$ は $f(a_1,\ldots,a_m,t_1,\ldots,t_n)$ を呼び出すことと同じである．たとえば，`call(=,X,1)`，`call(=(X),1)`，`call(=(X,1))` の実行結果はすべて `X = 1` である．`N` は次数が可変であることを示す．

次に，匿名述語の実装に使用した `copy_term/2` について説明する．`copy_term/2` は，入力項のすべての変数名を変更するようにして作ったコピー，すなわちフレッシュなコピーを出力項に返す．たとえば，クエリ `?- copy_term(f(a,X), Y)` の実行結果は `Y=f(a, _G328)` である．

`term_expansion/2` はコンパイル時に入力項を出力項にマクロ展開する．`goal_expansion/2` はボディ部に対して同様にマクロ展開を行う．

2.3 匿名述語の構文

匿名述語は Prolog の項によって表現される：

$$Head:-Body$$

$Head$，$Body$ はそれぞれ無い場合がある．$Body$ は Prolog のゴールである．$Head$ は $[A_1,\ldots,A_n]$ か $Free^\wedge[A_1,\ldots,A_n]$ である．$A_1,\ldots,A_n$ と $Free$ は Prolog の項である．$[A_1,\ldots,A_n]$ を引数リストと呼ぶ．$Free$ をパラメータと呼ぶ．$Free$ は匿名述語の中の自由変数である．匿名述語のスコープの外側にある変数を指定できる．$Free$ にないすべての変数はローカル変数と考える．匿名述語は呼び出し時に変数名を新しく置き換えるため，ローカル変数が匿名述語の外側に現れても問題はない．しかし，それらが外側に現れることはプログラムを読む側に混乱を招くため望ましくない．匿名述語であることを明記する場合，複合項 `pac`[†1]`/1` によって表現する：

`pac`(匿名述語)

匿名述語の標準形を複合項 `pred/3` で定義する：

`pred`$([A_1,\ldots,A_n],Free,Body)$

標準形は記号'.'，^，:-を使用せずにすべての匿名述語を表すことができる．

以上のように，様々な匿名述語の表現がある．いずれも従来の Prolog の構文に基づいて定義した．

2.4 匿名述語の意味

匿名述語を代表して，標準形 `pred/3` の意味を説明する．Arg を引数のリスト，$Free$ を自由変数のリスト，$Body$ を Prolog のゴールとすると，

`pred`$(Arg,Free,Body)\Longleftrightarrow p(Arg,Free)$

ここで，$p(Arg',Free')$:-$Body'$ は $Arg,Free,Body$ のフレッシュコピー $Arg',Free',Body'$ を生成した後，新しいアトム p を関数子としてアサートされた述語である．このとき $Arg,Free,Body$ は変数の関係を保つよう同時にフレッシュコピーされる．匿名述語を呼び出すということは，それと等価な名前付き述語をアサートし，その述語を呼び出すことと等しい．

3 匿名述語の利用例

3.1 匿名述語を含むプログラム

匿名述語を渡す場合，通常のリストと間違われることのないように宣言しなければならない．組み込み述語 `meta_predicate` $Head$,... は述語の引数の種類を指定する．$Head$ の各引数はメタ変数指定子である．1 から 9 の整数 N を指定された引数は，次数が N 以上の項と解釈される．指定子？は，モードが指定されていないことを表す．たとえば，高階述語 `call/N` は，`call(1,?)`，`call(2,?,?)` などと定義される．`maplist/N` も同様である．高階述語の第 1 引数がこのように宣言されているため，第 1 引数において'.'`/N` は匿名述語として実行される．引数に匿名述語を渡す述語を定義する場合，このように `meta_predicate` を用いて宣言する．

†1 pac は parametric anonymous clause の略

3.2 高階述語から匿名述語の利用

匿名述語は高階述語からの呼び出しに使うと便利である．次のクエリは，Prolog 組み込みの高階述語 call/N，maplist/N からの呼び出し例を示す．

```
1.?-call([X,X], 1, V).
  V = 1.
2.?-call([A,B,C]:-append(A,B,C),X,Y
    ,[1,2]).
  X = [],
  Y = [1, 2] ;
  X = [1],
  Y = [2] ;
  X = [1, 2],
  Y = [] ;
  false.
3.?-maplist([X,Y]:-Y is X*X*X
    ,[1,2,3,4,5],B).
  B = [1, 8, 27, 64, 125].
4.?-maplist(A^[X,[X,A]],[1,2,3],L),A=ok.
  L = [[1,ok], [2, ok], [3, ok]].
5.?-maplist([X,[X,A]],[1,2,3],L), A=ok.
  L = [[1,_G453], [2,_G480], [3,_G521]].
```

maplist$(f(a_1,\ldots,a_m),L_1,\ldots,L_n)$ はリストのすべての要素へ $f(a_1,\ldots,a_m)$ を適用する．クエリ 1 は単一化の例を示す．X = 1, V = 1 と単一化する．[X,X] は恒等関係述語 $(=)$ を表す．クエリ 2 は匿名述語が単一化による双方向性を実現していることを示す．クエリ 3 は関数表現として便利であることを示す．匿名述語 [X,Y]:-Y is X*X*X はラムダ λxy.(y is x*x) と同じ働きをする．クエリ 4 はパラメータ渡しの例である．クエリ 4 と 5 は自由変数を指定する場合としない場合の違いを示す．

3.3 DCG における匿名述語の利用

次の例は DCG のマクロ展開を示す．

a--> b, c. $\Longleftrightarrow$ a(X, Y):- b(X, Z), c(Z, Y).

a--> [b, c], d. $\Longleftrightarrow$ a([b, c|X], Y):- d(X, Y).

a--> pac([b]). $\Longleftrightarrow$ a(X, Y):- pac([b], X, Y).

2 番目の例で分かるように，DCG におけるリストは通常のリストと意味が違う．そのためリストで表された匿名述語は pac/1 を用いた項で表す．

4 匿名述語の実装

4.1 匿名述語の評価器 apply_pac

匿名述語は関数（入力と出力を持つ箱）ではない．call/N のような実行エンジンの第 1 引数となるべき Prolog の項である．この節では，本システムが用意した，匿名述語を実行する述語 apply_pac について説明する．apply_pac は匿名述語の頭部，ボディ部をそれぞれ評価した後，匿名述語の引数と適用する項とを単一化する．そして，ボディ部を実行する．評価器 apply_pac は次のように定義される．

```
apply_pac(X,A):-var(X),!,
  cast_to_list(X,X0), copy_term(X0,A).
apply_pac(H:-G,A):-!, apply_pac(H,G,A).
apply_pac(:-G,_):-!, apply_pac([],G,[]).
apply_pac(H,A):-apply_pac(H,true,A).

apply_pac(P,G,[]):-var(P), !,
  copy_term((P,G),(P,G0)), call(G0).
apply_pac(F^Q,G,A):-!,
  copy_term((F,Q,G),(F,A,G0)), call(G0).
apply_pac(Q,G,A):-
  copy_term((Q,G),(A,G0)), call(G0).
```

実行例を?-apply_pac([X]:-X>3,[4]) によって示す．apply_pac の第 1 引数は匿名述語に対応する複合項で，第 2 引数はその匿名述語に適用する引数を表す．まず，匿名述語の引数と適用される引数は copy_term による単一化が行われる．この例では，X = 4 と単一化される．次に，匿名述語のボディ部が call/1 によって呼び出され結果を出力する．この例では，4 > 3 が呼び出された結果 true が出力される．

4.2 匿名述語の実装

’.’，:-，^は次のように定義される．

```
'.'(A,B,C):-apply_pac([A|B],[C]).
'.'(A,B,C,D):-apply_pac([A|B],[C,D]).

^(A,B,C):-apply_pac(A^B,[C]).
^(A,B,C,D):-apply_pac(A^B,[C,D]).

:- meta_predicate:-(?,1,?).
:-(A,B):-apply_pac((A:-B),[]).
:- meta_predicate:-(?,2,?,?).
:-(A,B,C):-apply_pac((A:-B),[C]).
```

なお，実際はより多くの次数に対して同様に定義するが，ここでは省略した．定義から分かるように，す

$$
\begin{aligned}
\texttt{call}([\texttt{X},\texttt{X}],1,\texttt{V}) &\Longrightarrow \texttt{'.'}(\texttt{X},\texttt{X},1,\texttt{V})\\
&\Longrightarrow \texttt{apply\_pac}([\texttt{X},\texttt{X}],[1,\texttt{V}])\\
&\Longrightarrow \texttt{apply\_pac}([\texttt{X},\texttt{X}],\texttt{true},[1,\texttt{V}])\\
&\Longrightarrow \texttt{copy\_term}((([\texttt{X},\texttt{X}],\texttt{true}),([1,\texttt{V}],\texttt{G0})),\texttt{call}(\texttt{G0})\\
&\Longrightarrow \texttt{X}=1,\texttt{V}=1.
\end{aligned}
$$

図 1　匿名述語の実行例

べての匿名述語は apply_pac/2 によって評価され，copy_term/2 によってフレッシュな変数名に置き換えられた後，call/1 によって実行される．図 1 に，匿名述語呼び出しを実行した例をあげる．

'.'/2 はコンサルトを行う述語として組み込みで定義されている．再定義した'.' は 3 引数以上なので衝突は起こらない．ここで，3 引数以上で十分な理由は，実行エンジンの第 1 引数に置かれた匿名述語には必ず適用する引数が必要だということから自明である．ラムダそれ自体が実行できないように，匿名述語それ自体は実行できない．

4.3　名前付き述語への置き換え

maplist/N のように匿名述語を何度も使用する場合には，評価器 apply_pac/2 はつど評価するため効率が悪い．そのため，大量のデータに対して何度も使用する場合，名前付き述語を定義した方が効率的になる．第 5 章において紹介する，Prolog へラムダを導入するいくつかの研究も同様の理由から効率が悪いという問題がある．この問題を解決するために，匿名述語をコンパイル時に名前付き述語へと置き換えることを提案する．述語 expand_pac/1 はダイナミックに生成されたクエリを評価する．匿名述語があれば，それと等価な名前付き述語をアサートして置き換える．たとえば，?-expand_pac(call([X,X],1,V)) は，適当な名前付き述語 temp(X,X):-true をアサートした後，call(temp,1,V) を実行して結果を返す．expand_pac/1 は大量のデータに対する匿名述語の実行効率を大幅に向上する．次節で示すように，expand_pac による匿名述語の実行速度は名前付き述語を定義して呼び出す場合と等しい．

4.4　コンパイル時の置き換え

プログラムにおける匿名述語，DCG における匿名述語，そしてクエリにある匿名述語をコンパイル時に名前付き述語へ置き換える場合，それぞれ次のように定義する．

```
term_expansion --> expand_pac.
term_expansion --> dcg_translate_rule,
                   expand_pac.
goal_expansion --> expand_pac.
```

4.5　実行効率の比較

名前付き述語，匿名述語，また，関連研究として Ulrich [7] [8] のラムダを比較対象とし実行効率を調べる．本システムが定義した述語 time(N,Goal,T) を用いて実行速度を調べる．time/3 は Goal を N 回実行した速度 (単位：秒)T を出力する．

名前付き述語 named(X,Y):-Y is X*X の関数子 named，匿名述語 [X,Y]:-Y is X*X，そして Ulrich のラムダ項\X^Y^(Y is X*X) を F とし，$\texttt{time}(N,\texttt{maplist}(F,L,\_),T).$ を実行する．L には Long_List=[1,2,...,30] と Short_List=[1,2,3] を入力し，大小のデータに対する実行速度を調べる．匿名述語はコンパイル時に名前付き述語へ置き換える実行（expand_pac/1）と，逐次実行（apply_pac/1）の 2 通りの実行を行う．実行回数を $N = 1000000$ として実行速度 T を比較した結果を (表 1) に示す．

結果から次のことが分かる．

1. 匿名述語をコンパイル時に置き換えた場合，名前付き述語と実行速度は変わらない．
2. 匿名述語を逐次実行した場合，名前付き述語の実行速度より約 3 倍遅い．
3. 匿名述語をコンパイル時に置き換えた場合，Ul-

表 1 実行速度の比較

	Short_List	Long_List
named	T = 2.98	T = 16.38
expand_pac	T = 2.98	T = 16.38
apply_pac	T = 8.69	T = 54.17
Ulrich's lambda	T = 10.12	T = 93.22.

rich のラムダより大小のデータで各々3.40 倍，5.69 倍速い．

これは匿名述語の十分な実用性と，関連研究より大幅な実行効率の向上を実現したことを示す．

4.6 copy_term の最適化

匿名述語の定義に関連して，コード最適化のため copy_term/2 を部分評価した．unfolding と冗長コードの除去を行う．その際，資料 [14] において実装した述語 reduce，subtree，copy_args を使用した．いずれも，項書き換えに利用した述語である．copy_term の最適化により本システムの高速化を実現した．

5 関連研究

5.1 λ 項の導入

本稿の匿名述語は λ 項を Prolog へ導入する試み [14] [16] の中から，とくにコンパイル可能かつ有用な部分として得られたものである．Prolog への λ 項の導入の発端はモンタギュの論理文法 (MG) [4] の実装実験のためであり，実装目標は MG の考えを英語の断片に適用したものとして良く知られている PTQ [3] であった．

PTQ は a man や some man などの名詞句を一般化された限量子としてとらえることなど，重要な意味論である [15]．PTQ によれば，たとえば英文 John is Bill および John is a man の意味はそれぞれ等式 John = Bill および man(John) で表わされる．しかしながら途中の式変形は λ 計算， 等号規則 $t = u \wedge P(t) \Longrightarrow P(u)$ に基づく式の簡単化処理など，多くの計算を必要とする [15].

そこで，Prolog 組み込みの限定節文法 (DCG) [11] [10] により PTQ の構文規則と λ 計算による意味の合成規則を記述した．ただし PTQ の本来の統語論 (構文論) はカテゴリ文法 [1] で記述されている．こうして PTQ の文を内包論理式へ翻訳する Prolog プログラムが開発され，JavaScript/Ajax と通信させることによりすでに PTQ のウェブ学習教材として，公開されている [13].

PTQ の出力は高階内包論理式である．高階内包論理式には λ 束縛のほか個体限量子 ∀, ∃ による束縛，時制限量子による束縛，内包演算子 ^による束縛が現われる．λ 計算 [12]，とくに β 変換はこれら束縛演算子のスコープに依存するため複雑になる．新しい束縛子が拡張のために必要となることも考慮しなくてはならない．PTQ の λ 計算は当初は，こうした要請を満すべく関数型表現の評価器パッケージとして実装されたが，その後も λ 項を Prolog の組み込みの言語要素とすべく引き続き検討改良を加えてきた．そもそも高階内包論理は一階述語論理 (FOL) と λ 計算を含んでおり，関数型と論理型を統合した言語であるとみることができる．Prolog は FOL の部分言語であるから，λ 項の Prolog への導入は自然である．実際 [14] [16] で導入した λ 項の意味を，抽象構文を用いて以下のように自然に記述できる．変数に項を割り当てる関数 θ を代入と同一視する．項 t に代入 θ を適用した結果の項を $t\theta$ で表す．$M = p(a_1, \ldots, a_k)$ とおいて，λ 項 $\lambda x_1 \lambda x_2 \cdots \lambda x_n\, M$ の意味を規定するには

$$\texttt{call}(\lambda x_1 \lambda x_2 \cdots \lambda x_n M, t_1, t_2, \ldots, t_{m-1}, t_m)$$

のふるまいを記述すればよい．まず $n < m$ の場合，それは次と等価である．

$$p(a_1\theta, \ldots, a_n\theta, t_{n+1}, t_{n+2}, \ldots, t_m),$$

ここで $\theta = [x_1 := t_1,\ x_2 := t_2,\ \ldots,\ x_n := t_n]$ とする．たとえば，$\texttt{call}(\lambda x \lambda y\ \texttt{append}(\texttt{x}, \texttt{y}), [1], [2], v)$ の結果は $v = [1, 2]$ である．一方，$n \geq m$ の場合は

$$t_m = \lambda x_m \lambda x_{m+1} \cdots \lambda x_n (M\theta)$$

と等価である．ここで $\theta = [x_1 := t_1, \ldots, x_{m-1} := t_{m-1}]$ である．たとえば $\texttt{call}(\lambda x \lambda y\ \texttt{append}(\texttt{x}, \texttt{y}), [1], v)$ の結果は $v = \lambda y\ \texttt{append}([1], \texttt{y})$ である．

M がカンマで区切られている場合，つまり Prolog のゴールの形の場合の意味は λ 項の評価のためスタックの操作と関連づけて定義されるが，詳細は省く．なお匿名述語の場合は，そのまま Prolog のゴールとして解釈される．

5.2 λProlog

Prolog へのラムダは数多く提案されている．λPrlog [2] [5] は Prolog を拡張した高階プログラミング言語であり，高階単一化とラムダ項を導入した．ただし，特別な構文やコンパイラの拡張を必要とする [6]．そのため従来の Prolog ユーザには受け入れづらいという問題がある．

5.3 Ulrich Prolog

Ulrich は Prolog の双方向性を活かすラムダを実装した [7] [8]．いわば関係型のラムダである．Ulrich のラムダはコンパイラの拡張の必要がない．また，`copy_term` を使い属性付き変数の属性もコピーする．本システムの匿名述語は Ulrich のラムダの実装を参考にした．

6 まとめ

本稿では，Prolog への匿名述語の導入について，その意味論，実装方法，コンパイル時の置き換えを中心に述べた．匿名述語の構文と意味は明快であり，誰でも簡単に使いこなすことができる．また効率面でも，匿名述語はコンパイル時の置き換えと `copy_term` の最適化によって，逐次実行より約 3 倍早く，十分に実用できる．そして，関連研究と比べて 3 倍以上の速い実行速度を実現している．匿名述語の応用例として，高階プログラミング，DCG についても説明を行った．パッケージはオープンソースであり，以下の URL から取得可能である．

http://web.sfc.keio.ac.jp/~ykide/pac.pl

謝辞

匿名述語について議論していただいた慶応義塾大学の清木康教授，倉林修一博士に感謝致します．

参 考 文 献

[1] J. van Benthem. Categorial grammar meets unification, 1988.

[2] Timothy S. Gegg-harrison. Representing logic program schemata in lambdaprolog.

[3] R. Montague. The proper treatment of quantification in English. In J. Hintikka, J. Moravcsik, and P. Suppes, editors, *Approaches to Natural Languages: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*. Dordrecht: Reidel., 1970. Also appeared in *Formal Philosophy*, ed. R. Thomason. New York: Yale U.P. (1974).

[4] Richard Montague. Universal grammar. *Theoria*, Vol. 36, pp. 373–398, 1970.

[5] Gopalan Nadathur and Dale Miller. An overview of λ prolog. In *In Fifth International Logic Programming Conference*, pp. 810–827. MIT Press, 1988.

[6] Lee Naish. Higher-order logic programming in prolog, 1995.

[7] Ulrich Neumerkel. Lambdas und schleifen in monotonen logikprogrammen. *Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Oct 2009.

[8] Urlich Neumerkel. lambda.pl, 2009. http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/.

[9] Richard A. O'Keefe. *The craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.

[10] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis*. CSLI, 1987.

[11] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Aritificial Intelligence*, Vol. 13, , 1980.

[12] 横内寛文. プログラム意味論. 情報数学講座 7. 共立出版, 1994.

[13] 向井国昭. CGI in Prolog. http://web.sfc.keio.ac.jp/~mukai/mathcgi/.

[14] 向井国昭, 井出陽子. 関数型評価器および項書き換え器の prolog 上の実装, 2008.

[15] 白井 賢一郎. 形式意味論入門—言語・論理・認知の世界. 産業図書, 1985.

[16] 井出陽子, 向井国昭. Prolog へのラムダ項の導入とマクロとしての展開, 2010.