一般化論理プログラミング言語UL/αのコンパイラ

出葉 義治　繁田 良則　赤間 清　宮本 衛市

北海道大学　工学部　情報工学科

一般化論理プログラムの理論に基礎づけられたプログラミング言語UL/αのコンパイラシステムについて説明する．UL/αは，多様なデータ構造を，柔軟にかつ論理的に取り扱うために情報付き変数を導入した新しい論理型言語である．この言語を用いることで，ユーザは，型情報，高階関数，継承などの対象を的確に表現し，処理することが，容易に行なえる．

本稿では，はじめにUL/αの概要を示し，次にそれに沿った仮想マシンの構造，コンパイル処理の方法，生成される仮想マシンコードなどを，実行例などを挙げて説明する．

A Compiler for the Generalized logic programming language UL/α

Yoshiharu Dewa　Yoshinori Shigeta　Kiyoshi Akama　Eiichi Miyamoto

Division of Information Engineering, Faculty of Engineering
Hokkaido University
Kita 13 Nishi 8, Kita-Ku, Sapporo 060, Japan

We present a compiler for the logic programming language UL/α which is based on the theory of generalized logic programs.

The most important feature of UL/α is its ability to treat various data structure logically using *variables with information.* To use this language,user can exactly describe various object, such as type information,higher order function and inheritance structure.

This paper shows the outline of the structure of our abstract machine and the methods of compiling. We also explain the compiling process using examples.

1 はじめに

最近の知識処理システムで扱う問題は取り扱うデータ構造が多様であり，その構造を的確に表現し，処理する能力が求められている．そのような，能力を持った論理型言語として我々は，論理型プログラミング言語UL/αを提唱している．UL/αは，多様なデータ構造を，柔軟にかつ論理的に取り扱うために情報付き変数を導入している．この言語を用いることで，ユーザは，与えられた問題に対するデータ構造を論理的に的確に表現し，処理することが，容易に行なえる．

本論文ではUL/αのコンパイラ及びその仮想マシンの実現方法について提案する．

2 プログラミング言語UL/α

2.1 UL/αのシンタックス

UL/αは，GLPの理論 [1] に基づいて設計された論理型プログラミング言語である．そのシンタックスは，次のようになっている．

- プログラムは，S式記法の Prolog と同様に記述される．
- 基本データ構造は，S式記法の Prolog ＋ 情報付き変数である．
- 情報付き変数の書式は，変数と情報のペアによって(変数) ~ (情報) という形で表わされる．
  情報には任意のS式を用いることができる．
- "*" ではじまるシンボルは，変数である．
- "?" は，無名変数である．

2.2 情報付き変数

情報付き変数 [2] の記述力の高さが，UL/αのプログラミングの最も重要な特徴である．UL/αから情報付き変数を除くと，通常の Prolog とほぼ等しくなる．したがって，UL/αを Prolog として使用することも出来る．しかし，情報付き変数は，Prolog に付加された特殊なデータ構造にすぎないと考えるべきでない．

UL/αのプログラムは，その大部分が情報付き変数によって記述され効果をあげる例が非常に多い．高階関数 [6] の表現などが，その良い例といえる．

この情報は何度でも付け替えることができる．図 1に，通常の論理変数（以後，純粋な変数と呼ぶ.）と情報付き変数の変化の違いを示す．情報付き変数では，一旦 (animal) から (dog) に変化しても，さらに (pochi) に変わることができる．論理変数では，一度，値が決まった変数はバックトラックしない限り二度と変わらない．このことが，純粋な変数と大きく違う点である．

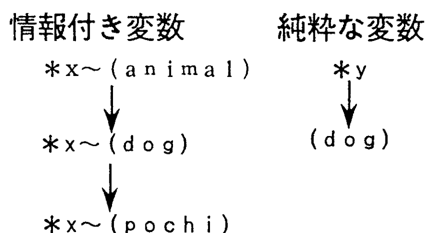

図 1: 情報付き変数の変化

ユーザーはそのユニフィケーション規則を自由に設定することができるので，多様な論理オブジェクトを作ることができる．

例えば，「1m という長さ」を， `*x~1m` と表し，「100cm という長さ」を `*y~100cm` と表すとする．「1m という長さ」と「100cm という長さ」をユニファイしたとき「1m という長さ」になるというユニフィケーションは，

```
(as (defUnify 1m 100cm 1m))
```

によって定義する．ここで，defUnify は論理オブジェクトのユニフィケーションを定義するための予約述語である．この定義が行なわれた後， `*x~1m` と `*x~100cm` のユニファイを行なうと成功し，結果は `*x~1m` つまり，「1m という長さ」になる．

このように，UL/αのプログラムは，完全に宣言的な論理オブジェクトを構成できるが，手続き的には C のポインタと同様な実行効率を持っている．

情報付き変数は，単に型のついたオブジェクトをつくるだけでなく，さまざまな複雑なデータ構造を表現することができ [5]，それによって，さまざまな言語をUL/αに埋め込む [3][4][6] ことが可能となる．

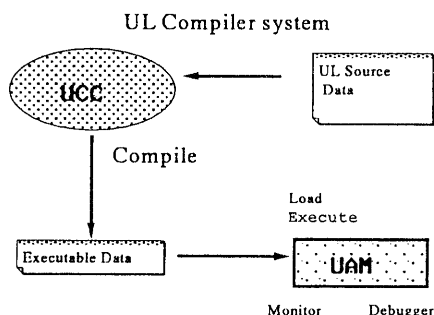

図 2: コンパイル処理の流れ

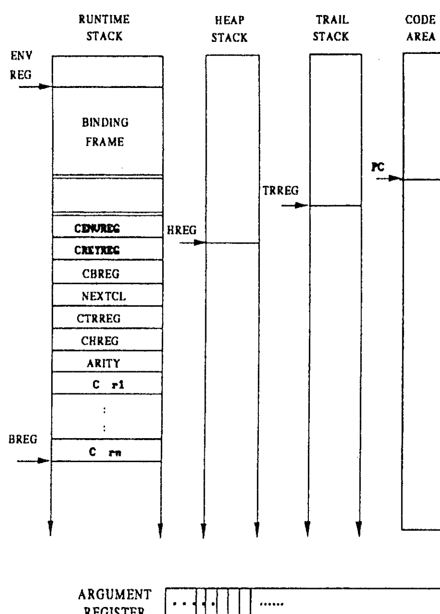

図 3: UAMの構造

3 仮想マシン

3.1 仮想マシンの構造

UL/αの仮想マシン（UAM）の仕様は，大筋として Warren の仮想マシン（WAM）[7][8][9] の仕様に沿ったものとなっている．構成は，４つの領域（コード領域，ヒープスタック，ランタイムスタック，トレールスタック）からなるメインメモリと大域レジスタ群，引数レジスタからなる（図3）．

- コード領域 (CODE AREA)
  仮想マシン命令を格納する領域．
- ヒープスタック (HEAP STACK)
  コンスセルの保存と大域変数の待避に使われるスタック．
- ランタイムスタック (RUNTIME STACK)
  局所変数などのためのフレーム (environment frame) とバックトラックのためのフレーム（choice point frame）を格納する．
- トレールスタック (TRAIL STACK)
  変数の書き換えが起きた時に，書き換える前の状態を保持するためのスタック.（バックトラック時に使用される）
- 大域レジスタ群
  (ENVREG,BREG,HREG,TRREG,PC)
  仮想マシンの動作環境を保持するためのレジスタ群.
- 引数レジスタ (ARGUMENT REGISTER)
  次の述語に渡っていく引数の内容を保存するためのレジスタ.

これらの構造は，WAM のそれと同じである．

3.2 仮想マシンにおける情報付き変数の実現

UL/αの根幹を成すものは，2.2節で述べた情報付き変数である．これを仮想マシン上で実現させるためには，情報の表現が必要になる．

WAM では，データ構造は<タグ，アドレス>というセルで表されている．例えば，変数のタグは REF であり，アドレスが他のセルを指すこと

で，他の変数へのつながりを表し，自分自身を指すことで，未定義な状態を表す．図4の中で，ランタイムスタックにある REF タグがついた変数*X，*Y，*Z は，上から「未定義状態」,「コンス(c)」,「変数*Yと等しい」という状態を表している．

そこで，情報付き変数を表すために，INFO タグを用意することにする．

情報付き変数のセルは，その変数が持つ情報をあらわすセルを指すアドレスを持つ．図4の変数*は，コンス(c) を指しているので,「コンス(c) という情報を持っている」という状態を表している．すなわち*~(c) を表している．

REF タグのついたセルを，純粋な変数と呼び，情報付き変数と区別をすることにする．

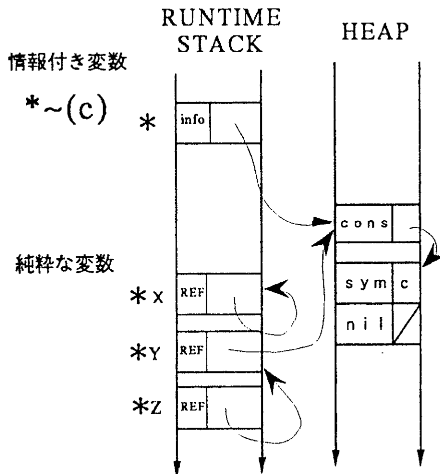

図 4: UAMにおける変数の表現

3.3 トレールスタックの変更

今までの WAM では，変数は内容が一旦定まるとそれは，それ以上変化しないので，トレールスタックには書き換えられた変数の番地だけを保存しておけば，バックトラック時にもとの変数の状態を回復できる．しかし，情報付き変数の場合には，情報の内容も保存しておかねばならない．何故ならば，情報付き変数の情報は，何度でも付け替えが起こり，付け替えられた後にバックトラックが起こった場合，元の情報に戻さねばならないからである．そこで，トレールの内容は，

（番地　書き換える直前の内容）

という組で表わされている．

3.4 情報付き変数のための新しい命令

情報付き変数を操作するため，以下の仮想マシン命令を追加した．

- putIVal R,X
  変数Xを新しくランタイムスタックに取り，レジスタRの内容を情報として持つ情報付き変数を生成する．(図5)

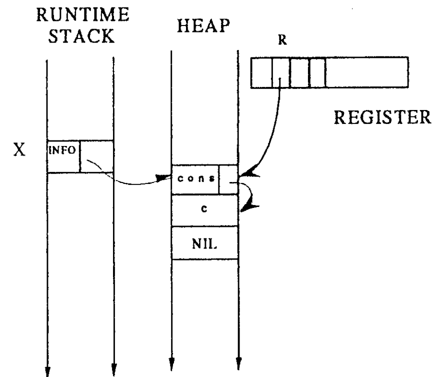

図 5: putIVal の動作

- putITVal R1,R2
  レジスタＲ２が指している変数（純粋な変数であることが，コンパイラで保証されている）にレジスタＲ１を情報として付加する．(図6)

3.5 情報付き変数同士のユニファイ

ここでは，情報付き変数同士のユニファイが，どのようなスタックの変化をもたらすかを説明する．

例を図7に示す．これは，

```
(as (defUnify (a) (b) (c)))
```

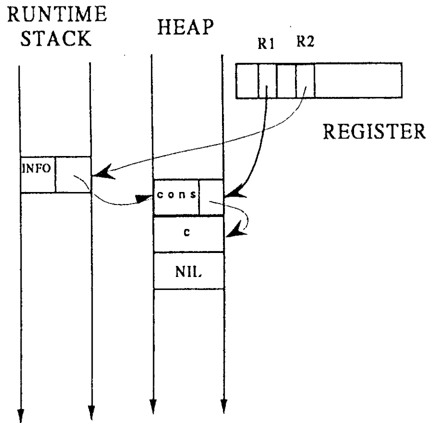

図 6: putITVal の動作

が定義された状態で，*x~(a) と *y~(b) のユニファイを実行したものである．

まず，一番左側のスタックは，ユニファイされる前の状態を表わしている．変数*X，*Yはそれぞれ (a),(b) という情報がついている．

次にユニファイが行なわれると真中のスタックのように変化する．束縛定理 [9] に従い，スタック下方の変数が上方の変数を指すように書き換えが行なわれる．また，このとき変数*Yのタグは REF になる．そして，変数*Xに新しい情報 (c) が付け替えられる．

情報の付け替えは後述の内部組み込み関数によって行なう．

4 コンパイラ

UL/α のコンパイラ (UCC = UL/α Code Compiler) はUL/α 自身によって記述されている．UL/α のソースを入力して，仮想マシン命令をファイルに出力する (図 2)．このファイルはリロケータブルで連結可能であり，ある程度の可読性を持つ．

4.1 情報付き変数のコンパイル法

情報付き変数同士のユニファイは，次の 2 通りの場合に起こる．

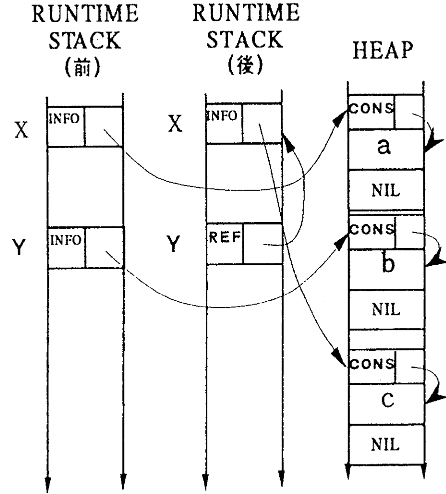

図 7: 情報付き変数同士のユニファイ

1. プログラム中に陽に現れる場合
   情報付き変数がプログラム中に記述され，呼び出し側の情報付き変数とユニファイが起きる場合．

2. プログラム中では，情報付き変数の存在がわからない場合
   プログラム中では，pure variable で記述されているが，実行時にユニフィケーションによって情報付き変数と，呼び出し側の情報付き変数とのユニファイが起きる場合．

この両方に対応するため，UL/α のコンパイラでは，従来のユニファイ命令は使用せず，ユニファイルーチン unify/2(付録 A参照) をコールする形でユニフィケーションが行われるようにコードを生成する．

また，unify/2 は，以下の仮想マシン組み込み命令を呼ぶ．

- getInfo X,Info
  情報付き変数 X から情報 Info の取りだしを行なう．

- bindPurevar X,Y
  X,Y のどちらかが純粋な変数である（プログ

ラムによって保証されている）ので，その変数に相手をバインドする．

- bindInfovar X,Y,Info
  変数 X と Y に情報 Info を付ける．

- pureVar X
  変数 X が純粋な変数かどうか調べる．違えば失敗する．

- infoVar X
  変数 X が情報付き変数かどうか調べる．違えば失敗する．

4.2 前処理

ここでは，コンパイラの前処理について説明する．

4.2.1 内部ユニファイルーチンのための処理

4.1節で，述べたように情報付き変数の出現は，2通りある．よって

- ヘッド中に同一の変数が現れた場合
- 情報付き変数が現れた場合

に内部ユニファイルーチンへの呼びだしを付加してやれば良い．例えば，

```
(as (p *x *y^(info) *x))
```

というヘッドは，

```
(as (p *a *b *c)
    (unify *a *c)
    (unify *b *d^(info)))
```

という形に変換されることになる．

4.2.2 変数の処理

例えば，

```
(as (p *x c *y)
    (q *x *z *i^(dog))
    (r *z *i *y))
```

は前処理によって，まず次のように変換される．

```
(p 3 0
 ((0 0 3 1)
  ((var 1 (t f 0))
   (sym c)
   (var 2 (p f 0)))
  (call (q 3)
   ((var 1 (t s 0))
    (var 3 (p f 1))
    (var 4 (p f 2)          ------(A)
           (info
            (cons 5 (sym dog)
                    (nil))))))
  (call (r 3)
   ((var 3 (u s 1))
    (var 4 (u s 2))
    (var 2 (p s 0))))))
```

変数は
(var id 番号 (flag1 flag2 flag3){(info 情報)})
の形で表されている．

flag1 は，変数のタイプを示している．それぞれ

- Temporary
  クローズ内で寿命がつきる変数．レジスタ操作だけで処理が行なえ，メモリに内容を保存しなくても良い．

- Permanent
  他のクローズに受け渡す必要のある変数．メモリに保存する必要がある．ただし，受け渡し終ったら破棄することができる．

- Unsafe
  早期フレーム廃棄により内容が保証されない変数．ボディの中で初めて現れ，最後の述語呼びだしで使われている変数は，ラストコールオプティマイゼーション [10] によって，その呼び出しが行なわれた直後にフレームの廃棄が起こり，破壊されるので，ヒープ領域に保存する必要がある．

flag2 は，変数の出現順序を表している．f は最初に現れたことを表し，s は 2 番目以降に現れたことを表す．

flag3 は，フレームに積まなくてはならない変数（局所変数と呼ぶ.）の数を表している．

もし情報付き変数だった場合には，情報の項が付加される．例えば，上の例の `*i~(dog)` は，(A) のように変換される．

この出力結果に従って，クローズごとに仮想マシン命令を出力する．その様子は，5節に示す．

この前処理により，変数の出現（新規かそうでないか）が，明示され，頭から順に仮想マシン命令を生成できる．

5 実行例

5.1 情報付き変数を含まない例

まず，情報付き変数を含まないプログラム

```
(as (test *z)
    (test2 *z *x)
    (test3 *x *y))
```

をコンパイルしてみる．生成される命令列は，次のようになる．

```
 1:(label [test/1])
 2: (alloc)                alloc frame
 3: (putVar 0 2)           make *x
                             load A2
 4: (call [test2/2] 2 1) gosub test2/2
 5: (putUVal 0 1)          reload
                                 *x to A1
 6: (putTVar 2)            make *y
                                 on Heap
 7: (dealloc)              dealloc frame
 8: (exec [test3/2] 2)     goto test3/2
```

このコードは，Warren の仮想マシンにある命令だけである．その動作を順に説明すると

- ランタイムスタック上に局所変数のためのフレームを用意する (2 行目)．
- 局所変数 0 をフレームに生成し，そのアドレス値をレジスタ 2 に格納する (3 行目)．
  これは，プログラム中の*x を生成したことに相当する．
- test2/2 を呼び出す (4 行目)．
  引数レジスタ 1 は，変化していないのでそのまま test2/2 の第 1 引数として受け渡される．
- test3/2 のために引数レジスタ 1 を局所変数 0(*x) から読み込む．また，その内容をヒープ上に生成する (5 行目)．
  (引数レジスタ 1 の内容は，test2/2 を呼び出したため保証されない．)
- 新しい変数*y をヒープ上に生成し，レジスタ 2 に，そのアドレス値を格納する (6 行目)．
  これは，*y が UNSAFE であるための措置である．
- フレームを廃棄する (7 行目)．
- test3/2 に飛ぶ (8 行目)．

5.2 情報付き変数を含む例

次に，情報付き変数を含むプログラム

```
(as (test *x~dog)
    (test2 *x *y~animal))
```

をコンパイルしてみる．このプログラムは，前処理によって

```
(as (test *z)
    (unify *z *x~dog)
    (test2 *x *y~animal))
```

という形に変化している．先の例と比較して説明を行なう．

```
 1:(label [test/1])
 2: (alloc)                  alloc frame
 3: (putVar 0 2)             make *x
                                  load A2
 4: (putSymCon 3 dog)        A3 <- dog
 5: (putIVal 3 0)            make *x~dog
 6: (call [unify/2] 2 1)     gosub unify/2
 7: (putUVal 0 1)            reload
                                  *x to A1
 8: (putTVar 2)              make *y
                                  on Heap
 9: (putSymCon 3 animal)     A3 <- animal
10: (putITVal 3 2)           make *y~animal
11: (dealloc)                dealloc frame
12: (exec [test2/2] 2)       goto test2/2
```

前の例との違いは，情報付き変数の生成(4-5,10-11行目)の部分だけである．

- 情報の生成．ここでは，dog というシンボルを引数レジスタ3(ここでは汎用レジスタとして利用している．)に格納する．(4行目)．
- 3行目で生成した局所変数0にレジスタ3が指すもの(ここではdog)を情報として格納する(5行目)．
- 内部ユニファイルーチンを呼び出す．(6行目)

10-11行目は，4-5行目と同じである．

このようにして，情報付き変数を取り扱うことが出来る．

6 おわりに

本研究では，ＵＬ/αのコンパイラおよびその仮想マシンの実現方法を示した．

今後，コンパイルコードの最適化や，仮想マシンの高速化をさらに行なっていく必要がある．最適化においては，クローズインデキシングやプログラム変換による命令列の改善などを検討している．また，開発環境の整備も併せて行っていく予定である．

なお，仮想マシンは，Ｃで記述されており，実行はSONY NWS-3860及びSUN SparcStation2上で行っている．

A 内部ユニファイルーチンソースコード

```
(as (unify *x *y)
    (eql *x *y)
    (cut))
(as (unify *x *y)
    (or (pureVar *x) (pureVar *y))
    (bindPurevar *x *y)
    (cut))
(as (unify *x *y)
    (infoVar *x)
    (infoVar *y)
    (getInfo *x *xinfo)
    (getInfo *y *yinfo)
    (cut)
    (defUnify *xinfo *yinfo *newinfo)
    (bindInfovar *x *y *newinfo))
(as (unify (*carx . *cdrx)
           (*cary . *cdry))
    (cut)
    (unify *carx *cary)
    (unify *cdrx *cdry))
```

実際の仮想マシンでは，このソースをコンパイルした後，高速化のための変換を施して，組み込んでいる．

参考文献

[1] Kiyoshi Akama: Sufficient Conditions of Two Inference Rules for Generalized Logic Programs, Proc of LPC'91, pp.161-170,1991.

[2] 坪山徳保, 赤間清, 宮本衛市: 制約付き変数とその応用, 電気関係学会北海道支部連合大会, 1989.

[3] 渡辺慎哉, 赤間清, 宮本衛市: 関数型言語から論理型言語への変換について, 日本ソフトウェア科学会第8回大会, 1991.

[4] 川口雄一, 赤間清, 宮本衛市: 型付き宣言型言語のＵＬ/αへの埋め込み, 日本ソフトウェア科学会第9回大会, 1992.

[5] 繁田良則, 赤間清, 宮本衛市: ＵＬ/αにおける無限データ構造, 日本ソフトウェア科学会第9回大会, 1992.

[6] 繁田良則, 赤間清, 宮本衛市: ユニフィケーションによる関数型プログラムの実行, 情報処理学会研究報告 92-PRG-6,pp.41-50 1992.

[7] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309.SRI International,Menlo Park,CA,October 1983.

[8] David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings,Menlo Park,CA,1988.

[9] H.Aït-Kaci. *Warren's Abstruct Machine: A Tutorial Reconstruction,* MIT Press,Logic Programming Series.Cambridge,MA,1991.

[10] David H. D. Warren. *an improved Prolog implementation which optimises tail recursion* Research Paper 156,Dept. of Artificial Intelligence, University of Edinburgh,Scotland,1980.

[11] David S. Warren,Suzanne Dietrich,SUNY at Stony Brook *The SB-Prolog System, Version2.5* SRI International,September 1988.