UL/αと型

川口雄一*・赤間　清**・宮本衛市**

UL/α and Types

Yuuichi KAWAGUCHI, Kiyoshi AKAMA, Eiiti MIYAMOTO

要　旨

我々は一般的論理プログラムの理論（GLPの理論）について研究を行っている。現在，我々はGLPに基づく宣言型言語UL/αを作り，その中に様々な計算パラダイムを宣言的に埋め込む実験を行っている。本稿では，UL/αに型の概念を埋め込む。そして型も含めた全ての計算対象を統一的な枠組みの上で扱うことを試みる。これは単なる既存のシステムの埋め込みではなく，新しいシステムの構築でもある。

Abstract

We study the theory of genelaized programming which is called GLP. We now make the language UL/α that is based on GLP, and try to embed some computational paradigms. In this paper, I embed the idea of Types.

This is not only embedding an old system into ours, but also constructing a new system.

1．はじめに

計算機言語における型の役割は大別すると以下になる。

- プログラムの読み易さを向上させる。
- コンパイル時にエラーを発見し易くする。
- 効率良いオブジェクトコードを作る。

以上のような観点より例えば手続き型言語のPascalやFORTRAN等ではデータに対する型の概念を言語中に取り入れている。更に，例えば関数型言語のMLでは従来の型検査に加えて，多相型や型推論の機能を提供している。また，宣言型言語のPrologのように元々は型の概念を持っていなかった言語にも，型を取り入れようとする研究がなされている[1]。

本研究は，我々が現在開発中の宣言型言語UL/α上に色々な計算機言語のパラダイムを埋め込む実験の一環として行われている[3],[4],[5]。言語UL/αは一般化論理プログラムの理論（GLPの理論）[2]によって意味付けがなされている。従ってこれら一連の埋め込み実験により，種々の計算機言語に対する宣言的意味を統一的な観点から与えることが可能になる。

本稿では，型に関する計算をUL/α上で実現することにより，宣言型言語における型計算がUL/αとして，すなわちGLPとして如何に表現されるかを示す。

2．GLPの理論

GLPの理論では扱う対象間の関係をspecialization structureと呼ばれる構造の上で定義する。このspecialization structureが定義された対象は，これを用いて，その宣言的意味，例えば最小不動点意味等を導き出すことが可能になる。その定義は以下である。

定義 specialization structureとは$\langle \mathcal{A}, \mathcal{S}, \mu \rangle$の三つ組であり，以下の条件を満たすものである。

(1)　$\mu : \mathcal{S} \rightarrow$ *partical-map* $(\mathcal{A})$

＊　助手　情報工学科
＊＊　北海道大学　工学部

(2) $\forall s_1,\ s_2 \in \mathcal{S},\ \exists s \in \mathcal{S}$ s.t.
$\mu(s) = \mu(s_2) \circ \mu(s_1)$

(3) $\exists s \in \mathcal{S}$,
$\forall a \in \mathcal{A}$ s.t. $\mu(s)(a) = a$

ただし$\mathcal{A}$は任意の集合で，その要素は論理 object と呼ばれる。この object 間の関係を$\mathcal{S}$によって定める。

上記のとおり，その構造定義が非常に抽象的になされていることにより，計算機科学における多種多様な対象に関し，specialization structure を定義可能である。現在，我々は specialization structure によって様々な計算対象を表現することを試みている。

3．宣言型言語 UL/α

宣言型言語 UL/α は，GLP の理論に基づいて作られている。従って，UL/α 上で動作するプログラムは GLP の specialization structure によって，その宣言的意味を導くことが可能である。

UL/α のプログラムは S 式によって記述される。記号＊で始まるシンボルか？は変数である。特に？は無名変数である。UL/α は Prolog のように，述語の宣言と問い合わせによってプログラミングを行う。

言語 UL/α の最大の特徴は，「情報付き変数」にある。すなわち，普通の論理変数に対してユーザー定義の情報を付加することが可能である。ここでいう「情報」とは任意の S 式によって表現されるものである。例えば (info) という情報の付いた変数？は，

```
?~ (info)
```

と記述される。

この情報付き変数同士の単一化は，ユーザーが自由に定義可能で，これは述語defUnifyの宣言により行われる。例えば二つの情報付き変数 ?~ (A)，?~ (B) は，

```
(defUnify (A) (B) (C))
```

が成功するときに単一化され，その結果 ?~ (C)になる。

この単一化の定義が specialization structure の$\mathcal{S}$と対応している。

4．型付き宣言型言語の例

型付き宣言型言語の例として Typed Prolog [1] をあげる。Typed Prolog で計算を行う場合，(1)データの型，(2)述語の型，(3)プログラム，の 3 つを宣言する必要がある。このとき(2)は与えないか，部分的にしか与えなくてもよく，その場合は，型推論により型情報を補う。(3)は普通の Prolog プログラムであり，これに対して型検査／型推論が行われる。

4．1 Append

Typed Prolog で Append を記述した例を示す。

```
% TYPE DECLARATIONS
type list (T) → '[]' ; [T | list (T)].

% PREDICATE DEFINITIONS
pred append (list (B), list (B), list (B)).

% PROGRAM
append ([], L, L).
append ([X | L1], L2, [X | L3]) : -
                    append (L1, L2, L3).
```

この記述を入力として受取り，Typed Prolog の型検査／型推論のアルゴリズムは次の出力を返す。

```
% Type Constructors
type list (1).

% Types for Functions
type [] : [] → list (B).
type .: [B, list (B)] → list (B).

% Types for Predicates
pred append (list (B), list (B), list (B)).

% Program
append ([], L, L).
append ([X | L1], L2, [X | L3]) : -
                    append (L1, L2, L3).
```

この例では(3)が完全に与えられているので型検査を行い，成功する。

この例で示されるとおり，Typed Prolog の扱う計算対象としてはlist (B)等のデータ型と []

(nil) や[T|L] (cons) 等の関数と呼ばれるものの2種類がある。そして他の多くの型付き言語と同じく，それぞれの対象に対し別個の計算を施し，プログラムを実行する。実行時に型の計算は行われない。

5．UL/αへの埋め込み

本節では，Typed Prologに関する考察を基に，型に関する計算をUL/α上に実現する。

基本的な方針としては，計算対象である型，データ等を区別して扱うことをせず，統一的な枠組みの上にこれらを組み込む。その上で，データの計算と同じ原理で，自然に型の計算も行われるようにする。

5．1 計算対象

Typed Prologでの扱いとは異なり，我々は計算対象として型を普通のデータから区別して扱わない。すなわち，型もデータもアトムも(1)全て単一の形をした計算対象として(2)同じ計算の枠組みの中で扱う。UL/αではこの対象を，一般的に以下に示す情報付き変数 *aとして表現する。

```
対象：：＝ *a~((型名　型引数…)) |
          *a~((型名　型引数…)
            *r~(set (スロット名　対象)
                    (スロット名　対象)…))
```

情報“(型名　型引数)”が，一般的にいう「型」の部分である。情報 *r の部分はその型の詳しい特徴を表現しており，スロットの集合 (set) である。スロットの中には再び対象が入る。型引数やスロットを持たない対象も存在できる。

例えば，「整数型のデータ」は?~ ((NUM))で表現され，「整数を要素とするリスト型のデータ」は型引数を伴い，?~ ((LIST NUM))で表現される。同様に「整数値5」は実は5という型を持つデータであり，?~ ((5))で表現される。

スロットのある対象の例としてはCONSデータがある。CONSにはCar部とCdr部が存在する。従って，CHAR型の文字データを要素とする（a b. nil）という形のCONSは以下で表現される。

```
?~ ((CONS CHAR) ?~ (set
    (Car ?~ ((a)))
    (Cdr ?~ ((CONS CHAR) ?~ (set
                (Car ?~ ((b)))
                (Cdr ?~ ((NIL CHAR))))))))
```

更には，（NUM. nil）のような，最初の要素が整数型であることのみを表現するリストも，以下で表現可能である。

```
?~ ((CONS NUM) ?~ (set
    (Car ?~ ((NUM)))
    (Cdr ?~ ((NIL NUM)))))
```

この形をなす情報付き変数の集合をもってobjectの集合$\mathcal{A}$を定義する。以上のように計算対象の構造を定めたことにより，全ての計算対象の構造を定めたことにより，全ての計算対象を単一の構造上で捉えることが可能になっている。

5．2 UL/α上の型付きプログラム

以上のように定義した計算対象を用いてUL/α上の型付きの計算システムを実現する。プログラムは，(1)対象個々が持つスロット，(2)対象間の親子関係の木，(3)型付きのプログラム，の3つの宣言によって規定する。(1)，(2)の宣言を用いて対象間の単一化規則を生成する。

全ての計算対象を単一の枠組みで扱うので，Typed Prologと異なり，対象が型であるかデータであるか等によって区別した宣言を行わない。

5．2．1 対象間の単一化

型がどんなスロットを持つことが可能かを定義するためには述語patternを用いる。例えばCONSにはCar部とCdr部というスロットが存在するので，次のように宣言する。型引数の部分を変数にすることで，多相型を表現している。

```
(pattern (CONS *t) (Car (*t))
                   (Cdr (LIST *t)))
```

この宣言により，内部的にRestrという述語が以下のように宣言される。

```
(as (Restr *x (CONS *t))
    (= *x
       ?~ (set (Car ?~ ((*t)))
               (Cdr ?~ ((LIST *t))). ?)))
```

述語Restrは，呼び出されると (CONS *t) に

対するスロットの集合と *x との単一化を行う。

スロットを持たない対象は，そのことを明示するため，次のように宣言する。

```
(pattern (NUM))
```

もう一つ，対象間の親子関係の木は述語treeを用いて宣言する。例えば，LISTは子としてCONSとNILを持つが，これは次のように宣言する。

```
(tree LIST (NIL CONS))
```

例のとおり，親子関係は型名のみをもって宣言する。この宣言によって，内部的には述語isaが次の形で宣言される。isaの引数は左が子で右が親である。

```
(isa NIL LIST)
(isa CONS LIST)
```

宣言されたpatternとtreeの情報を用い，対象同士の単一化は例えば，以下のように行われる。スロットを持たない対象のために，実際はdefUnifyを複数宣言している。

```
(as
(defUnify (*t1 *x) (*t2 *y) (*t3 *x))
   (lower *t1 *t2 *t3)
   (Restr *x *t3)
   (= *x *y))
```

ここで変数 *t1, *t2, *t3 は例えば(LIST NUM) に対応し， *x, *y はそれに属するスロットの集合に対応する。

述語lowerは引数 *t1と *t2の間に pathが存在し，子の方に位置するのが *t3であることを示す。従って図1のような木が定義されている場合，(lower (DATA) (NUM) (NUM))は成功し，(lower (CONS) (NIL) ?)は失敗する。lowerの内部では，述語isa呼び出される。

例として，(LIST NUM)型の objectと(CONS ALL)型の objectとの単一化を示す。型引数まで含めて変化している。

```
(= ?~ ((LIST NUM))
   ?~ ((CONS ALL) ?~ (set (Car ...)...)))

(= ?~ ((CONS NUM) ?~ (set (Car ···)...))
   ?~ ((CONS NUM) ?~ (set (Car ...)...)))
```

5．2．2　プログラムの記述

上のように構成された単一化に基づき，プログラムが如何に動作するかを示す。例としてAPPENDを示す。対象間の親子関係はあらかじめ図1と定義しておく。

スロットを持つのはCONSとAPPENDである。CONSは上述のとおりとし，APPENDのスロット定義は以下とする。

```
(pattern (APPEND *t) (Fst (LIST *t))
                     (Snd (LIST *t))
                     (Trd (LIST *t)))
```

実際に動作するAPPENDプログラムを宣言するためには述語typeULを用いる。宣言は以下となる。

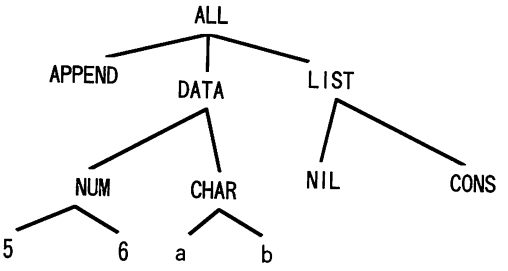

図1　計算対象の親子関係の木

```
(as (typeUL
    ?~ ((APPEND *t) ?~ (set
       (Fst ?~ ((NIL *t)))
       (Snd *~ ((LIST *t)))
       (Trd *~ ((LIST *t)))))))
(as (typeUL
    ?~ ((APPEND *t) ?~ (set
       (Fst ?~ ((CONS *t) ?~ (set
                (Car *a~ ((*t)))
                (Cdr *x~ ((LIST *t))))))
       (Snd *y~ ((LIST *t)))
       (Trd ?~ ((CONS *t) ?~ (set
                (Car *a~ ((*t)))
                (Cdr *z~ ((LIST *t)))))))))
```

```
(typeUL
?~ ((APPEND *t) ?~ (set
   (Fst *x) (Snd *y) (Trd *z)))))
```

6．実行例

上述のように宣言されたプログラムに対し，幾つかの問い合わせを行い，その動作結果を示す。

6．1 例1

普通の Prolog では(APPEND (5) *x (5 6))に当たる問い合わせを例として示す。2番目の引数が変数であるのは(1)の部分に示されている。

```
(typeUL
?~ ((APPEND NUM) ?~ (set
    (Fst ?~ ((CONS NUM) ?~ (set
             (Car ?~ ((5)))
             (Cdr ?~ ((NIL NUM))))))
    (Snd *list)                    ;.....(1)
    (Trd ?~ ((CONS NUM) ?~ (set
             (Car ?~ ((5)))
             (Cdr ?~ ((CONS NUM) ?~ (set
                      (Car ?~((6)))
                      (Cdr ?~ ((NIL NUM)
))))))))))
```

その結果は，普通のAPPENDと同様，以下になる。

```
(typeUL
?~ ((APPEND NUM) ?~ (set
    (Fst ?~ ((CONS NUM) ?~ (set
             (Car ?~ ((5)))
             (Cdr ?~ ((NIL NUM))))))
    (Snd ?~ ((CONS NUM) ?~ (set      ;(1')
             (Car ?~ ((6)))          ;(1')
             (Cdr ?~ ((NIL NUM)))    ;(1')
)))
    (Trd ?~ ((CONS NUM) ?~ (set
             (Car ?~ ((5)))
             (Cdr ?~ ((CONS NUM) ?~ (set
                      (Car ?~ ((6)))
                      (Cdr ?~ ((NIL NUM)
))))))))))
```

変数 *list が(1') に変化する過程を説明する。

1．述語typeULとマッチする時点で，変数 *listは?~ ((LIST *t))と単一化される。

2．問い合わせ側が(APPEND NUM)であることから，型引数 *t は NUM と単一化される。従って ?~ ((LIST *t)) は?~ ((LIST NUM))に変化する。

3．再帰的にもう一度 typeUL が呼び出され，このとき，Fstで示される引数は?~ ((NIL NUM))である。従って，一番目のAPPENDプログラムが適用され，SndとTrd で示される引数を単一化した結果，?~ ((LIST NUM))は

```
?~ ((CONS NUM) ?~ (set
    (Car ?~ ((6)))
    (Cdr ?~ ((NIL NUM))))
```

に変化する。

この例でわかるように，型や型に関する計算を特別扱いせずに，プログラムは動作している。

6．2 例2

これは，型検査の結果失敗する例である。

```
(typeUL
?~ ((APPEND NUM) ?~ (set
    (Fst ?~ ((CONS ALL) ?~ (set
             (Car ?~ ((a)))
             (Cdr ?~ (((NIL ALL))))))
    (Snd ?)
    (Trd ?~ ((CONS ALL) ?~ (set
             (Car ?~ ((a)))
             (Cdr ?~ ((CONS ALL) ?~ (set
                      (Car ?~ ((6)))
                      (Cdr ?~ ((NIL ALL)
))))))))))
```

すなわち，(APPEND NUM)からの影響でAPPENDの引数は全て(LIST NUM)型であることになるが，aはCHAR型であり，この点で型検査は失敗する。

6．3 例3

次の例では，向きをもってスロットが伝播することを示している。問い合わせとして，以下を与える。

```
(typeUL
?~ ((APPEND NUM) ?~ (set
```

```
(Fst ?~ ((CONS DATA) ?~ (set
        (Car ?~ ((5)))
        (Cdr ?~ ((NIL DATA))))))
                                ;.....(3a)
(Snd ?~ ((CONS ALL) ?~ (set
        (Car ?~ ((6)))
        (Cdr ?~ ((NIL ALL))))))
                                ;.....(3b)
(Trd *list))))                  ;.....(3)
```

結果は以下に示すとおり，(3)の変数 *list が(3')では(CONS NUM)型に変化している。この変化は例1と同じ理由による。

```
(typeUL
?~ ((APPEND NUM) ?~ (set
   (Fst ?~ ((CONS NUM) ?~ (set   ;...(3c)
           (Car ?~ ((5)))
           (Cdr ?~ ((NIL NUM))))))
                                ;...(3a')
   (Snd ?~ ((CONS NUM) ?~ (set
           (Car ?~ ((6)))
           (Cdr ?~ ((NIL NUM))))))
                                ;...(3b')
   (Trd ?~ ((CONS NUM) ?~ (set   ;...(3')
           (Car ?~ ((5)))
           (Cdr ?~ ((CONS NUM) ?~ (set
                   (Car ?~ ((6)))
                   (Cdr ?~ ((NIL NUM))))))
```

(3a) でスロット Cdr が (NIL DATA) から (NIL NUM) に変化したのは，そのスロットを持つ型が (CONS DATA) であり，その型引数 DATAと (APPEND NUM) の型引数 NUM が lowerによって，NUM に変化したからである。(3b) から (3b') への変化も同様の理由による。

スロットの伝播には「外側の型から内側の型へ」という方向性がある。そのため，(3c) の (CONS DATA) 部分が，Car 部の内容(5)との影響によって (CONS 5) や (CONS NUM) になることはない。

7．まとめ

従来においては，別個の扱いを受けていた型，データ，アトム等の計算対象を本稿では単一の形の object として扱うことにより，型付き宣言型言語を UL/α 上に統一的に実現する方法を示した。

なお，型推論については未だ実現していないが，この統一的枠組みの上における新しい観点からの定式化が可能であると考えている。

また，従来の型理論は関数型言語のパラダイムとして論じられることが多く，研究も進んでいる。これと我々のシステムとの比較を行うことは，重要な課題である。

参考文献

(1) T. K Lakshman, Uday S. Reddy : *Typed Prolog : A Semantic Reconstruction of the Mycroft-O'Keefe Type System,* anonymous ftp from a. cs. uiuc. edu, 1991.

(2) Kiyoshi Akama : *Sufficient Conditions of Two Inference Rules for Generalized Logic Programs,* Proc. of LPC'91 (1991), pp. 161–170.

(3) 繁田良則，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回大会論文集，D 3－3，1991.

(4) 渡辺慎哉，赤間清，宮本衛市：オブジェクト指向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回大会論文集，D 3－4，1991.

(5) 渡辺慎哉，赤間清，宮本衛市：GLP から見た並行計算系，日本ソフトウェア科学会第9回大会論文集，A 5－1，1992.

（平成4年11月30日受理）