ユニフィケーションによる関数型プログラムの実行

繁田 良則　　赤間 清　　宮本 衛市

北海道大学　工学部　情報工学科

代表的な関数型言語であるMirandaを取り上げ、GLPの理論をベースにした論理型言語ULへの埋め込みおこなう。ここで、埋め込みとはMirandaプログラムの実行をULの実行の原理、すなわち、オブジェクトのユニフィケーションによって説明することを意味している。単に、Mirandaのインタプリタを ULで実現したのではないことに注意されたい。また、この埋め込みは、関数型言語と論理型言語の融合の新しいアプローチと考えることができる。

関数型言語には、高階関数、遅延評価、型機構などの特徴があるが、本論文では、高階関数と遅延評価ついて議論をおこなっている。

Execution of Functional Programs By Unification

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 embed Miranda programs which are typical functional programs into UL programs based on the theory of generalized logic programs(GLP theory). We explain execution of Miranda programs by unification mechanism in UL. This does not mean that we implement Miranda interpreter using UL, but we show a new approach to integrate functional programs and logic programs.

In functional programs, there are some important ideas(i.e. higher order function, lazy evaluation and type checking mechanism). In this paper, we discuss execution mechanism of higher order function and lazy evaluation in Miranda programs and in UL programs through some example programs.

1 はじめに

関数型言語 Miranda のプログラムから GLP の理論 [1] をベースにした論理型言語 UL[4] への埋め込みを試みる。

この研究は、既存の様々な言語から UL への埋め込みを試みる実験の一環である (図 1)。これまでに、Lisp および Smalltalk-80 から UL への埋め込みについての報告 [10][11] を行なっている。

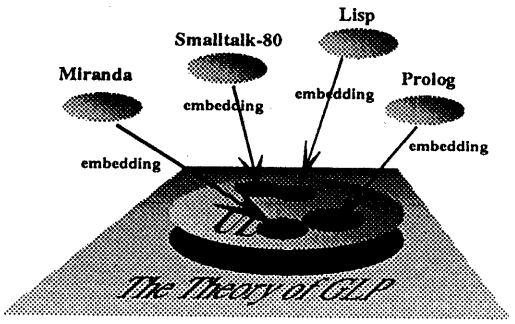


図 1: UL への埋め込み

種々の言語から UL への埋め込みによって、UL を軸として言語間の関係を明らかにできる可能性がある。つまり、各々の言語に備っている機能を UL という一つの言語に写像することで、UL を共通言語として議論を行なうことができるのである。このことは、同時に UL のベースである GLP の理論が、これまで別々に扱われていた様々な言語の理論に対して統一的視点を与える可能性を示唆するものでもある。そして、実際に UL への埋め込みができることを一つずつ確かめる一環として、本論文では、Miranda から UL への埋め込みを試みている。

Miranda には、高階関数、遅延評価、データ抽象化機構 (型機構) などの特徴があるが、本論文では高階関数、遅延評価に焦点を当て、これらを UL へ埋め込んでいる。データ抽象化機については、別の機会に報告する予定である。

2 関数型言語 Miranda

ここでは、Miranda の特徴について説明する。

2.1 特徴

Miranda の特徴は、以下の様なものであるとされている [5][6][7]。

- 等式による定義
- 高階関数
- 数学の表記法に近くなるように工夫された構文
- 遅延評価
- 型推論とデータ抽象化機能

2.1.1 等式による定義

Miranda では、定数の定義、変数の定義、関数の定義を区別せず、等式を用いて名前とその定義として統一的に扱っている。

```
a = 10              || 定数の定義に相当
q = a * a           || 変数の定義に相当
square x = x * x    || 関数の定義に相当
|| から行末まではコメント
```

定数と変数の定義を、引数なしの関数定義とみれば、すべてが関数定義であると考えることもできる。また、Miranda では破壊的代入や副作用を排除しており、参照透明性 (referentially transparency) が保証されている。等式による定義と参照等価性によって、Miranda ではプログラムに対する形式的な推論が可能となっている。

2.1.2 高階関数

高階関数は、関数を第 1 級の対象 (first-class object) として扱うことを意味している。つまり、関数を引数として受け渡したり、結果として返したりできるのである。例えば、

```
twice f x = f (f x)
```

は第 1 引数に関数 f をとり、それを第 2 引数 x に 2 回適用する高階関数の定義である。また、Miranda では関数はすべてカリー化された関数として扱われる。従って、2 個以上の引数をもつ関数はすべて高階関数となる。

2.1.3 工夫された構文

リスト処理を簡潔に表現するためにドット式とZF 式と呼ばれる表記法が用意されている。Miranda では、これらはリストを返す関数の syntax sugar として扱われている。

ドット式は、等差数列を簡便に表現するもので、例えば、

```
oneToTen = [1..10] ||1 から 10 までのリスト
naturals = [1..]   || 自然数のリスト
odds     = [1, 3..] || 奇数の無限リスト
```

などである。ただし、Miranda では'||' から行末まではコメントである。また、ZF 式の一般形は、

```
[body |qualifier;... qualifier]
```

で、qualifier は " 変数<-リスト " という形のリスト要素生成式であるか、生成されたリスト要素の値域を制約するフィルタ式である。。例えば、0 以上の偶数は、

```
evens = [x |x<-[0..]; x mod 2 = 0]
```

で表現できる。

2.1.4 遅延評価

遅延評価の例を示す。

```
hd (map sqr [1..])
    ⇒ hd(sqr 1:(map sqr [2..])
    ⇒ sqr 1
    ⇒ 1
```

この例は、[1..] の先頭の 1 要素だけ簡約し、残りの簡約を遅延している例である。

また、遅延評価は正規順評価(normal order evaluation) によって説明されている。

2.1.5 データ抽象化機能

Miranda は、静的で強く型付けされた言語であり、プログラムはコンパイラによって実行前に型の無矛盾性の検査を行なう。また、Miranda には型推論機能が組込まれているので、ユーザによる型の指定は必要ない。また、型変数を用いる多相型や代数データ型そして抽象データ型など柔軟で豊富な型を備えている。

本論文では、データ抽象化機構の埋め込みについては触れていない。別の機会に報告する予定である。

3 プログラミング言語 UL

この節では、UL の特徴や GLP の理論との関係について述べる。そして、UL の記法と実行例について説明を行う。

3.1 UL の特徴

UL は、一般化論理プログラムの理論 (GLP の理論)[1] に基礎付けられた論理型プログラミング言語である。GLP の理論では、オブジェクト (原子論理式) を自由に定めることができる (ある公理を満たさなければならないが、その制限は非常に緩い) 。これに対応して、UL には、ユーザによってユニフィケーションを自由に定義できる情報付き変数が導入されている。

表 1: GLP の理論と UL との対応関係

GLP の理論	UL
オブジェクト	情報付き変数
オブジェクトの空間とその構造の定義	情報付き変数の unification の定義
GLP プログラム	UL プログラム

本論文で議論する UL への埋め込みの基本的戦略は、式、関数など Miranda が扱っている対象をそのまま情報付き変数で表現し、適切なユニフィケーションを定義することである。そして、本論文では、情報付き変数のユニフィケーションによって、Miranda プログラムの実行が説明できることを例を示しながら説明する。

また、これ以降、UL の情報付き変数を " オブジェクト " と呼ぶことにする。GLP の理論における原子論理式と混同しそうな場合には、" 情報付き変数 " と書くことにする。

3.2 UL の記法

- UL プログラムは S 式記法 (標準的な lisp の S

式記法+論理変数+情報付き変数)

- 変数は'*' ではじまるシンボル
- '?' は無名変数
- 'v~info' は情報付き変数 (オブジェクト)

情報付き変数は、変数 v と情報 info のペア v~info で表現される。このとき、info は任意の情報であり、プログラム中では任意の S 式に対応する。また、情報付き変数のユニフィケーションは述語 defUnify/3 によって定義する。例えば、2 つの情報付き変数 *x~infoX, *y~infoY のユニフィケーションは、述語呼出し

(defUnify infoX infoY infoNew)

が成功するとき成功し、2 つの変数はユニファイされともに*x~infoNew となる。

3.3 実行例

以下のようにユニフィケーションが定義されている場合、

(as (defUnify animal dog dog))

以下のユニフィケーション

(unify ?~animal ?~dog) ;; ? は無名変数

は成功し、

(unify *~dog *~dog)

となるが、

(unify ?~animal ?~car)

は、ユニフィケーションが定義されていないので失敗する。

4 UL への埋め込み

3節で説明したように、UL プログラムでは、まずオブジェクト (情報付き変数) の形とそのユニフィケーションを定義しなければならない。そこで、まず最初に埋め込みに必要なオブジェクトを定義し、その後、埋め込みの方法について説明を行なうことにする。

4.1 オブジェクトの形

Miranda を UL に埋め込むために用いるオブジェクトは以下の 10 種類である。

表 2: オブジェクトの形

Miranda	オブジェクトの形	略記
式	*x~(Form *f . *a)	[x:Frm *f . *a]
関数	*x~(Function *f)	[x:Fun *f], *f
整数型	*x~(Number *num)	[x:Num *num]
無限大	*x~(Infinite)	[x:Inf]
真値	*x~(True)	[x:Tru]
偽値	*x~(False)	[x:Fls]
nil	*x~(Nil)	[x:Nil]
cons	*x~(Cons *a *d)	[x:Cns *a *d]
ドット式	*x~(Dot *s *e *i)	[x:Dot *s *e *i]
ZF 式	*x~(ZF *l *f)	[x:ZF *l *f]

Miranda では、ドット式と ZF 式はリストを返す関数の syntax sugar であるとされているが、UL では他のオブジェクトと対等なもの (first-class object) として扱っている。ユーザの望むオブジェクトを自由に定義できる UL にとっては、この方が自然な扱いなのである。これは、Miranda が対象をできるだけ "関数" という言葉で説明しようとするのに対し、UL では、対象をそのまま "オブジェクト" によって表現しようとする違いである。

以降この 10 種類のオブジェクトをまとめて "Miranda オブジェクト" と呼ぶことにする。

また、*x~(.....) の表記は繁雑であるので、本論文では略記法を使用する。略記法は、

[変数名:オブジェクト名　スロット...]

となっている。また、?~(.....) のように無名変数の場合には、変数名を省略することにする。

略記法の注意点をまとめておく。

- Function オブジェクトは "foldr" などのように直接シンボルで略記する
- *x~(Form ?~(Form ?~(Function plus) ?~(Number10)) ?~(Number 20)) のようにカリー化された Form オブジェクトを [x:Frm plus [Num 10] [Num 20]] のように略記する
- [Cns [Num 10] [Cns [Num 20] [Cns [Num 30] [Nil]]]] のようなものを [Lst: [Num 10] [Num 20] [Num 30]] と略記する

- 以降に示すプログラム中で、Form, Number, ... と書くべきところを Frm, Num, ... のように略記する

4.2 ユニフィケーションの定義

4.1節で導入した10種類のオブジェクトのユニフィケーションを定義する。図2は、ユニファイ可能なオブジェクトのペアを表している。線で結ばれた上段と下段のオブジェクトがユニファイすると下段のオブジェクトになる。また、同じオブジェクト同士のユニフィケーションも可能である。例えば、Form オブジェクトを Number オブジェクトのユニフィケーションが成功すると、両オブジェクトは Number オブジェクトになる。

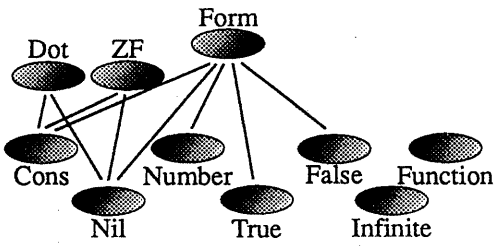


図 2: ユニファイ可能なオブジェクトのペア

例として Number オブジェクトのユニフィケーションの定義を示す。

```
C1:(as (defUnify
          (Num *n) (Num *n) (Num *n)))
C2:(as (defUnify
          (Num *n) (Frm *f . *a) (Num *n))
       (ε [Frm *f . *a] *reducedform)
       (unify *reducedform [Num *n]))
```

ここで、述語 unify は、2つのオブジェクトをユニファイする述語である。また、述語εは、Miranda プログラムに対応する述語である。述語ε については4.3.1節で詳しく述べる。

Number オブジェクト同士のユニフィケーションは、値がユニファイ可能な場合に成功する。また、Number オブジェクトと Form オブジェクトのユニフィケーションは、述語ε/2 によって Form オブジェクトを簡約し、その結果と Number オブジェクトがユニファイ可能であるとき成功し、Number オブジェクトになる。

4.3 UL への埋め込み

この節では、

- 等式による定義
- 高階関数とカリー化
- ドット式と ZF 式
- 遅延評価

について、埋め込みの方法と、実際の埋め込み例を示す。

4.3.1 等式による定義

UL への埋め込みの基本的戦略は、Miranda の扱っている対象をそのまま UL のオブジェクトに対応させ、Miranda における等式による定義を、述語εの定義に置き換えることである。例えば、以下の Miranda プログラム

関数 foo の定義 (Miranda)

```
foo x y = bar (plus x y)
```

この関数定義は、UL では、

- 左辺が Form オブジェクト [Frm foo *x *y]
- 右辺が Form オブジェクト [Frm bar [Frm plus *x *y]]

に対応するので、

関数 foo の定義に対応する述語εの定義 (UL)

```
(as (ε [Frm foo *x *y]
       [Frm bar [Frm plus *x *y]]))
```

という述語εへと埋め込まれる。

また、パターン照合を用いて定義される関数'hd' は、

```
Miranda: hd (a:d) = a
UL     : (as (ε [Frm hd [Cns *a *d]] *a))
```

のように埋め込まれる。

4.3.2 高階関数とカリー化

Miranda を埋め込むために導入した Form オブジェクトはすべてカリー化している。例えば、

```
Miranda:(Foo x y) カリー化すると ((Foo x) y)
UL      :?~(Form ?~(Form Foo *x) *y)
```

そして、4.1節で説明した略記法を用いると、

```
略記:[Frm [Frm Foo *x] *y]
     あるいは [Frm Foo *x *y]
```

となる。[Frm Foo *x *y] もカリー化されていることに注意されたい。また、特にカリー化を強調したい場合には、[Frm [Frm Foo *x] *y] の記法を用いることにする。

また、高階関数を扱うために、UL では以下に示す述語εの定義がある。

高階関数を扱うための述語εの定義

```
C3:(as (ε [Frm *f . *a] [Frm *reduced . *a])
       (ε *f *reduced))
```

この定義は、Form オブジェクトを部分的に変化 (*f から*reduced へ) させることを意味している。また、Form オブジェクトはカリー化されているので、例えば、Form オブジェクトが [Frm [Frm Foo *x] *y] のときには、[Frm Foo *x] が変化することになる。

以下のように動作する高階関数 foldr の埋め込み例を示す。

```
foldr (+) 0 [2,4,6]      ⇒ 12
foldr (*) 1 [2,4,6]      ⇒ 48
foldr (:) [3,4] [1,2]    ⇒ [1,2,3,4]
```

foldr の定義 (Miranda)

```
foldr op z = aux op z
```

aux の定義 (Miranda)

```
aux op z [] = z
aux op z (a:d) = op a (aux op z x)
```

foldr の定義 (UL)

```
C4:(as (ε [Frm [Frm foldr *op] *z]
          [Frm [Frm aux*op] *z])
```

aux の定義 (UL)

```
C5:(as (ε [Frm [Frm [Frm aux *op] *z] [Nil]]
          *z))
C6:(as (ε [Frm [Frm [Frm aux *op] *z]
               [Cns *a *x]]
          [Frm [Frm *op *a]
               [Frm [Frm [Frm aux *op] *z]
                    *x]]))
```

である。

'foldr plus 0 [10,20]' の実行は、Number オブジェクトと Form オブジェクトのユニフィケーションによって行なわれる。

```
(unify [Num *result]
       [Frm foldr plus [Num 0]
            [Lst [Num 10] [Num 20]]])
```

そして、

```
[Frm  foldr plus [Num 0]
      [Lst [Num 10] [Num 20]]]
      ⇓ C4
[Frm aux plus [Num 0]
      [Lst [Num 10] [Num 20]]]
      ⇓ C6
[Frm plus [Num 10]
          [Frm aux plus [Num 0]
                        [Lst [Num 20]]]]
      ⇓ C6
[Frm plus
      [Num 10]
      [Frm plus [Num 20]
                [Frm aux plus [Num 0] [Nil]]]]
      ⇓ C5
[Frm plus [Num 10]
          [Frm plus [Num 20] [Num 0]]]
      ⇓
[Frm plus [Num 10] [Num 20]]
      ⇓
[Num 30]
```

となって、*result=30 で成功する。

4.3.3 ドット式と ZF 式

以下の Miranda のドット式は、

```
[1..10] 最小 1 、最大 10 、公差 1 のリスト
[1,3...] 最小 1 、最大 ∞ 、公差 2 のリスト
```

UL では、Dot オブジェクト

```
[Dot [Num 0] [Num 10] [Num 1]]
[Dot [Num 1] [Inf] [Num 2]]
```

になる。

また、1 以上の整数で 2 の倍数でないリスト表す ZF 式は、

```
[x |x <-[1..]; oddp x]
```

であるが、UL では、

```
[ZF [Dot [Num 0] [Inf] [Num 1]] [Frm oddp]]
```

という ZF オブジェクトになる。ここで oddp は引数を 2 で割った余りが 0 より大きいとき真になる関数 (UL では対応する述語ε) とする。

本論文では、ZF オブジェクトにリスト生成式とフィルタがそれぞれ 1 つであるように限定しているが、2 つ以上の場合でも基本的には同様に扱うことができる。

Dot オブジェクトと Cons オブジェクトとのユニフィケーションの例を示す。

```
C7:(unify [Cns *car *cdr]
          [Dot [Num 1] [Num 10] [Num 3]])
          ⇓
    [Cns [Num 1]
         [Dot [Num 4] [Num 10] [Num 3]]]
```

ZF オブジェクトと Cons オブジェクトのユニフィケーションも基本的には、Dot オブジェクトの場合と同じだが、フィルタの存在により多少複雑になる。つまり、生成式によって生成される要素のうちフィルタを通過するものが来るまで生成式を読み進めるのである。

ZF オブジェクトと Cons オブジェクトとのユニフィケーションの例を示す。mod2Gt0、mod3Gt0 をそれぞれ偶数のとき、2 で割った余りが 0 以上のとき、3 で割った余りが 0 以上のとき真になる関数 (に対応する述語ε) とすると、

```
(unify [Cns *car *cdr]
       [ZF
         [ZF [Dot [Num 2] [Inf] [Num 1]]
         [Frm mod2Gt0]]
         [Frm mod3Gt0]])
          ⇓
[Cns [Num 5]
     [ZF
       [ZF [Dot [Num 6] [Inf] [Num 1]]
           [Frm mod2Gt0]]
       [Frm mod3Gt0]])
```

この例では、二重のフィルタ mod2Gt0、mod3Gt0 を満足する最初の要素 [Num 5] まで生成式が読み進められ、ユニフィケーションが成功する。

4.3.4 遅延評価

遅延評価の例として、エラトステネスのふるい法を用いて素数列を求める Miranda プログラムを UL に埋め込む。そして、素数列を求める過程を追いながら、UL における遅延評価について考察を行なうことにする。

```
primes の定義 (Miranda)
primes = sieve [2..]
sieve の定義 (Miranda)
sieve(a:d) = a:sieve [x <-d; x mod a > 0]

primes の定義 (UL)
C8:(as (ε [Frm primes]
          [Frm sieve
               [Dot [Num 2] [Inf] [Num 1]]]))
sieve の定義 (UL)
C9:(as (ε [Frm sieve [Cns *a *d]]
          [Cns *a
               [Frm
                 sieve
                  [ZF *d [Frm modNGt0 *a]]]]))
```

ここで、modNGt0 は 2 引数 n、x をとり、x を n で割った余りが 0 以上のとき真になる関数 (に対応する述語ε) である。

素数列の計算は Cons オブジェクトと Form オブジェクト [Frm primes] とのユニフィケーション

```
C10:(unify [Cns *1st *rest] [Frm primes])
```

によって行なわれる。そして、

```
(unify [Cns *1st *rest] [Frm primes])
```

```
        ⇓ prime の定義 (C8)
(unify [Cns *1st *rest]
        [Frm sieve
              [Dot [Num 2] [Inf] [Num 1]]])
        ⇓ sieve の定義 (C9)
(unify [Cns *1st *rest]
        [Cns
          [Num 2]
          [Frm
            sieve
            [ZF
              [Dot [Num 3] [Inf] [Num 1]]
              [Frm modNGt0 [Num 2]]]]])
        ⇓
*1st=[Num 2]
*rest=[Frm sieve
              [ZF
                [Dot [Num 3] [Inf] [Num 1]]
                [Frm modNGt0 [Num 2]]]]
```

となる。

また、素数列の先頭3個を求めるためには、

```
C11:(unify [Cns*1st
                [Cns *2nd
                     [Cns *3rd *rest]]]
           [Frm primes])
```

なるユニフィケーションを行なえば良い。そして、このユニフィケーションは、

```
*1st = [Num 2]
*2nd = [Num 3]
*3rd = [Num 5]
*rest = [Frm sieve
              [ZF
                [ZF
                  [ZF
                    [Dot [Num 6] [Inf] [Num 1]]
                    [Frm ModNGt0 [Num 2]]]
                  [Frm ModNGt0 [Num 3]]]
                [Frm ModNGt0 [Num 5]]]]]
```

となって成功する。

さらに、素数列の順次求めるためには、Cons オブジェクトとのユニフィケーションを繰返せば良い。以下に素数列を表示する述語'printPrimes' の定義を示す。

素数列を表示する述語

```
C12:(as (printPrimes [Cns *prime *rest])
        (print *prime)
        (printPrimes *rest))
```

そして、述語呼出し (printPrimes [Frm primes]) によって、

```
[Num 2]
[Num 3]
[Num 5]
[Num 7]
[Num 11]
...
```

のように素数が表示される。

さて、上で説明した Cons オブジェクトと Form オブジェクト [Frm Primes] のユニフィケーション C_{10} と C_{11} について考えてみる。ともに、*rest が [Frm sieve ...] のままで、それ以上計算 (述語εの呼出し) が進んでいないことがわかる。このことは、UL では、

- オブジェクト同士のユニフィケーションによって計算が進み、
- そして、ユニフィケーションに必要な計算 (述語εの呼出し) だけが行なわれること

を意味している。つまり、オブジェクトのユニフィケーションによって、Miranda における遅延評価のメカニズムを説明できるのである。

5 考察

いま、4.3.4節で説明した素数列を表示する述語 'printPrimes' について考えてみる。この述語は、関数型言語と論理型言語の融合の例となっている。この述語は、述語呼出し

```
(printPrimes [Frm primes])
```

によって素数列を順に表示する。その定義は、

```
(as (printPrimes [Cns *prime *rest])
    (print *prime)
    (printPrimes *rest))
```

であり、通常の UL プログラムである。しかし、Cons オブジェクトと Form オブジェクトのユニフィケーションを定義しておくことで、Form オブジェクト [Frm Prime] が素数列のリストを返す関数のように振舞っている。このように、UL では、Miranda オブジェクトを導入することで、論理型言語の枠組み内で、自然に関数を扱うことができるのである。

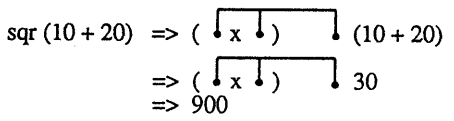
関数型言語と論理型言語を融合/統合を試みる多くの研究がある [14][15][17]。その多くは、等式論理 (equationallogic)[16] にその理論的基礎を置いている。この等式論理は、従来の論理プログラムに理論に、等式の概念を導入することで、関数型言語と論理型言語を融合/統合しようとしている。

これに対して、UL/GLP のアプローチは、オブジェクト (原子論理式) を自由に定義できる枠組から出発し、論理プログラムの枠内で関数型言語と論理型言語の融合を行なっている。

また、UL/GLP のオブジェクトは、単に関数型言語と論理型言語の融合のためだけにあるのではなく、論理プログラムに柔軟なデータ表現を提供するものでもある。例えば、関数型言語を効率良く動作させるための方式にグラフ簡約 (graph reduction) 法 [9] がある。これは、式'srq (10 + 20)' を最外簡約法で簡約する場合に、

$$\begin{aligned} \text{sqr}\ (10+20) &\Rightarrow (10+20) \times (10+20) \\ &\Rightarrow 30 \times (10+20) \\ &\Rightarrow 30 \times 30 \\ &\Rightarrow 900 \end{aligned}$$

となって、'(10 + 20)' の計算が 2 回行われ、効率が悪いので、



のようにグラフを用いて'(10 + 20)' を共有し、簡約を行う方法である。グラフ簡約方法は、一般には処理系の問題と考えられているが、Miranda オブジェクトを導入した UL では、プログラム上でこの共有を表現でき、特別な処理系を用意することなく、効率よく計算を行うことができる。

述語'SQR' を

```
(as (SQR *xx *sqr)
    (times *xx *xx *sqr))
(as (times [x:Num *x] [y:Num *y] [z:Num *z])
    (:= *z (× *x *y))) ;;*z = *x × *y を計算
```

と定義し、述語呼出し

```
(SQR [Frm plus [Num 10] [Num 20]] *result)
```

を行うと、

```
(SQR [Frm plus [Num 10] [Num 20]] *result)
        ⇓ (1)'SQR' の定義
    *xx=[Frm plus [Num 10] [Num 20]]
    *restult=*sqr
        ⇓ (2)'times' の定義
    *xx=*x=[Num 30]
    *a=30
        ⇓ (3)'times' の定義
    *y=*xx=*x=[Num 30]
    *b=30
        ⇓ (4)*c=*a×*b を計算
    *c=900
    *z=*sqr=*restult=[Num 900]
```

(2) で*xx と*x のユニフィケーションによって [Frm plus [Num 10] [Num 20]] が計算され、*xx=*x=[Num 30] となる。そして、(3) で*xx と*y のユニフィケーションがおこるが、(2) で *xx=[Num 30] となっているので、[Frm plus [Num 10] [Num 20]] の計算は行われずに、*y=*xx=*x=[Num 30] となる。

本論文では、Miranda から UL への埋め込みについて考察を行った。そして、

- Miranda オブジェクトの導入 (4.1節参照)
- そのユニフィケーションの定義
- Miranda プログラムに対応する述語εの定義

によって、Miradna プログラムが UL に埋め込めることを示した。そして、

- オブジェクトのユニフィケーションによって Miranda の実行 (高階関数や遅延評価) が説明できること

- 埋め込みが関数型言語と論理型言語の融合になっていること
- オブジェクトによってデータの共有が表現できること

を確認した。

6 おわりに

MirandaプログラムのULへ埋め込みをおこなった。今後、報告しなかったデータ抽象化機構および型推論についての報告を行なうことを考えている。また、埋め込んだMirandaプログラムを現在作成中のULコンパイラでコンパイルし、その実行効率などについて考察を行なう予定である。

今回行なったMirandaからの埋め込みと[10][11]で報告したLispおよびSmalltalk-80からの埋め込みは、既存の様々なプログラミング言語からULへの埋め込みの可能性を示唆するものであると考えている。また、同時に、理論的には、これまで別々に扱わざるを得なかった各プログラミング言語の理論に、ULの基盤であるGLPの理論が、統一的視点をもたらす可能性を示すものであると考えられる。

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] Kowalski,R.A.: Predicate Logic as Programming Language, Prod. IFIP-74 Congress, North-Holland, 1974, pp.569-574.

[3] Lloyd, J.W.: Foundations of Logic Programming, Springer-Verlag, 1984.

[4] 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.

[5] David Turner: An Overview of Miranda, SIGPLAN Notices, Vol.12, No.12, Dec, 1986.

[6] David Turner: Miranda: A non-strict functional language with polymorphic types, Springer-Verlag, LNCS 201, pp.1-16.

[7] 加藤和彦: 新しいプログラミングパラダイム第7章Miranda, 共立出版, pp.139-163.

[8] R.Milner: A theory of Type Polymorphism in Programming, J.Comput. Syst. Sci.,17, pp.348-375, 1978.

[9] R.Bird, P.Wadler, (武市正人 訳), 関数プログラミング, 近代科学社, 1991.

[10] 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会大会論文集, 1991.

[11] 渡辺慎哉, 赤間清, 宮本衛市: オブジェクト指向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会大会論文集, 1991.

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

[13] Hassan Ait-Kaci: Warren's Abstract Machine, MIT press, 1991.

[14] DeGroot, D. and Lindstrom, G. (eds.): Logic Programming : Functions, Relations and Equations, Prentice-Hall, 1986.

[15] Juan Jose Moreno-Navarro, Mario Rodriguez-Artalejo: Logic Programming with functions and predicates : The language BABEL, J.Logic Programming 12, 1992.

[16] S. Hölldobler: Foundations of equational logic programming, LNAI, 353, 1989.

[17] S. Okui, T. Nishioka, K. Kuroishi, and T. Ida: Semantics of a lazy equational programming language, 日本ソフトウェア科学会第8回大会論文集, pp.316-364, 1991.