記 号 処 理 57－2
（1990．11．22）

KL1 におけるメタプログラミング

越村 三幸
日本ビジネスオートメーション (株)
藤田 博
三菱電機 (株) 中央研究所
長谷川 隆三
(財) 新世代コンピュータ技術開発機構

並列記号処理言語 KL1 においてメタプログラミングをする際の問題点について述べる．KL1 では並列処理が自然に記述できるが，Prolog のようにオブジェクトレベルとメタレベルの変数を識別することができない．したがって，Prolog で用いられるようなメタプログラミング技法を KL1 では用いることはできない．本論では，オブジェクトレベルの変数とメタレベルの変数の扱い方に焦点を当てる．そして，効率的な実現手法を提案しその評価について考察する．また，応用として Prolog インタプリタの実現例を示す．

Meta-Programming in KL1

Miyuki Koshimura
Japan Business Automation Co., Ltd
2-1 Nissin-cho, Kawasaki-ku, Kawasaki, Kanagawa 210, JAPAN
e-mail: koshimura@icot.or.jp
Hiroshi Fujita
Central Research Laboratory, Mitsubishi Electric Corporation
1-1, Tsukaguchi-honmachi 8-chome, Amagasaki, Hyogo 661, JAPAN
e-mail: fujita@sys.crl.melco.co.jp
Ryuzo Hasegawa
Institute for New Generation Computer Technology (ICOT)
21F., Mita-Kokusai Bldg., 1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN
e-mail: hasegawa@icot.or.jp

Meta-programming is rather difficult in KL1 than in Prolog, because in KL1 meta level variables can not be distinguished with object level variables in consistent manner. In KL1, therefore, it is not possible to write meta-programs in the similar way to Prolog. This paper gives efficient meta-programming techniques in KL1 and evaluates their performance through running several sample programs. In addition, a Prolog interpreter with the techniques is shown.

1 はじめに

KL1[1] は並列論理型言語 Flat GHC[10] にさまざまな拡張を施した言語である．KL1 は論理型言語の特徴であるユニフィケーションを言語機能として取り入れてはいるが, 次の点で逐次論理型言語 Prolog と決定的に異なる．

1. バックトラック機構がない．
2. 論理変数の扱いに制限がある．

このため Prolog と比べ

1. 探索プログラム
2. メタプログラム

の記述が難しい．

反面 KL1 は並列言語であるため並列処理が自然に記述できる利点がある．従来の手続き型言語で並列処理を記述した時に起こったような同期問題に関することに頭を悩ますようなことはほとんどない．また実際の並列マシン [5] 上で走らせることにより, プログラムを変えることなくスピードの向上が期待できる．

本論文では KL1 でメタプログラミングを行う際の問題について考える．特にオブジェクトレベルの変数とメタレベルの変数の取り扱いに焦点を当てる．そして効率的なメタプログラミング技法を提案, 評価し本技法の有効性について考察する．またこの技法は OR 並列に適していることを述べ, AND 並列を引き出す際の問題点を考察する．

2 メタレベルとオブジェクトレベル

並列論理型言語 [7] と呼ばれる言語でのメタプログラミングには大きく分けて二通りの方法がある．その違いはオブジェクトレベルの変数をどのように表現しているかによる．一つはオブジェクトレベルの変数を (記述言語の) 論理変数で表す方法である [6]．これは Prolog のメタプログラミングで通常用いられている方法である．もう一つはオブジェクトレベルの変数を基底項表現する方法である [15, 8]．

論理変数表現 この方法の最大の利点はオブジェクトレベルの変数管理を記述言語のもつ変数管理機構に任せられる点である．これにより簡潔かつ高速なメタプログラムの作成が可能となる．

基底項表現 この方法では変数管理機構を記述言語でプログラミングする必要がある．これは Lisp のような記号処理言語でのメタプログラミングで通常用いられる方法である．メタプログラミングの正道とでも言うべき方法であるが, 変数管理機構をプログラミングしなければならないのでプログラムの複雑化と低速化をもたらす．

Shapiro は Flat Concurrent Prolog (FCP) による簡潔な Or-Parallel Prolog インタプリタを示している [6]．ここではオブジェクトレベルの変数には論理変数が用いられている．このインタプリタはガード部の full test-unification を利用して Prolog のユニフィケーションを実現している．Full test-unfication により並列論理型言語と Prolog の融合が可能になっているわけである．

残念ながら KL1 のガード部においては full test-unification ではなく one-way unification しか許されないので Shapiro の方法をそのまま KL1 に適用することはできない．しかし Prolog 述語のモード情報が十分静的に分かっていれば, 実行時の full test-unification は必要でなくなり one-way unification で十分となるので, KL1 で直接実現することは可能である [17]．

オブジェクトレベルのプログラムのモード情報が十分に分かっているという仮定の下での他のアプローチにコンパイル方式がある．これは全解探索型 Prolog プログラムを KL1 プログラムに変換する方法である [9, 14]．これらの方法の欠点は, モード情報が未知な問題に対しては無力なことである．特に定理証明などはモード情報の静的解析が困難な分野の一つであろう．

以上より,KL1 では情報の流れが静的に分かっている対象に対しては,KL1 の論理変数をオブジェクトレベルの変数として利用することができることがわかる．これは KL1 のガード部での one-way unification により情報の流れが静的に決定されてしまうからである．

一方情報の流れが動的に決定するような対象, 例えばユニフィケーションが本質的な問題, に対しては論理変数をオブジェクトレベルの変数として利用することはできない．このような対象を KL1 で記述する場合はやはり, オブジェクトレベルの変数を基底項表現する必要がある．したがって, 変数管理プログラムも記述しなければならない．

次節以降で, オブジェクトレベルの変数表現を具体的に幾つか与え, それらについて考察する．

3 オブジェクトレベルの変数の表現

オブジェクトレベルの変数の表現法として次の 3 種類を考えた．

1. KL1 変数表現．KL1 の論理変数でオブジェクトレベルの変数を表す．
2. 基底項表現．KL1 の基底項でオブジェクトレベルの変数を表す．
3. 折衷表現．KL1 変数表現と基底項表現の折衷表現．

そして, それぞれについてユニフィケーションプログラムを KL1 で記述しその性能評価をした [12]．本節ではそれぞれの表現法の特徴を述べ, 次節で評価結果について述べる．

3.1 KL1 変数表現

この方式の利点は変数管理を KL1 言語処理系に任すことにより高速性が期待できることであるが, 前節で言及した通り一般的には健全なプログラムの記述は不可能である. というのは論理変数に対する unboundness の判定が困難だからである.

並列言語である KL1 では逐次型言語の Prolog などと異なり, 変数が未定義のままであることを認識することに本来意味が無い. 未定義であると認識した直後に (並行して走る別のプロセスにより) 既に未定義ではなくなっているかも知れないからである. 例えば, 次のような KL1 プログラムを考える.

```
p(X) :- unbound(X) | X = 1.
p(X) :- bound(X) | true.

q(X) :- unbound(X) | X = 2.
q(X) :- bound(X) | true.
```

そして, p/1 と q/1 を並列に走らせてみる.

```
?- p(X), q(X).
```

p/1 と q/1 の X に対する unboundness チェックが同時に行われた場合両方のチェックが成功するので, p/1 により 1が q/1 により 2 が X にユニファイされる. KL1 にはバックトラック機構がないので, この矛盾によりプログラム全体が異常終了してしまう.

以上のような不健全性を踏まえた上でユニフィケーションプログラムを書いた. 具体的には, KL1 に unbound という組み込み述語が用意されているのでこれを利用した. ただし, オブジェクトレベルの変数の unboundness はユーザが保証してプログラムしていることを前提とした[1].

当初この方式では変数管理をまったくする必要がないと期待していたが, 実は変数管理の大部分をしなければならない. ユニフィケーションに失敗した時は未定義変数は未定義変数のままであって欲しいからである. Prolog のようにバックトラックしてくれれば容易にこの要求にこたえるユニフィケーションプログラムを書くことができるが, 単一代入の KL1 ではそうはいかない.

このようなユニフィケーションプログラムを KL1 で書く場合は, オブジェクトレベルのユニフィケーションが失敗しないことを確認してからメタレベルのユニフィケーションをする必要がある. つまり, ユニファイ中にはオブジェクトレベルの変数の値をその変数自身にバインドせずユニファイ終了後にバインドしなければならない. したがってユニファイ中は, 変数と変数の値の組の管理つまり変数管理が必須になる.

ユニフィケーションプログラムに変数管理が必要となるので KL1 変数表現の利点はユニフィケーション終了時の変数の値の伝播を処理系に任せられる点だけになってしまう.

[1] この前提は単一プロセッサ内では可能

3.2 基底項表現

この方式では変数管理が必要となるが, 変数管理の方法を二通り考えた. 一つは変数管理表 (環境) をプロセス表現するもので, もう一つはベクタ表現するものである. どちらもオブジェクトの表現と環境の対によってオブジェクトを表現するもので Lisp の eval で用いられている方法に近い[2].

3.2.1 プロセス方式

これは GHC によるユニフィケーションプログラム [2] を KL1 用に書き換えたものである. この方法が最も KL1 らしいプログラムといえようか. Lisp の a-list をプロセスで表現したものと思えば良い. 各変数について一つずつその変数管理プロセスを生成し, それらを直列にストリームでつなげるものである. ストリーム並列性がありユニフィケーションのもつ並列性が最も良く反映されるプログラムである.

3.2.2 ベクタ方式

オブジェクトレベルの変数の値を環境 (ベクタメモリ) に保持する方法である. オブジェクトレベルの変数は値の保持されている場所へのポインタで表現する[3]. この表現法の欠点は, 環境を持ち回って変数管理をするために制御が逐次的になることである.

KL1 ではベクタというデータ型があるが, これは要素に直接アクセスできるという点がリストに対して著しく有利である. この利点を活かし, ベクタ上に変数セルをとり, 変数はそのベクタインデックスとして表現する. これにより, 変数の unboundness は変数セルに書かれるタグで判定できるし, 2 つの変数の同一性はベクタインデックスの同一性で簡単に判定できる.

変数セルは図 1のようにベクタ上にとられる. 変数セルには次の 3 種類が入る.

0 未定義

point(Num) 変数番号 Num へのポインタ

data(Data) 具体値へのポインタ

[2] 例えば, 以下の表現と環境の対は同じオブジェクト f(a) を表している.

- 表現 $= f(a)$, 環境 $= \{\}$
- 表現 $= f(X)$, 環境 $= \{X \rightarrow a\}$
- 表現 $= Y$, 環境 $= \{Y \rightarrow f(X), X \rightarrow a\}$

[3] ユニフィケーションに失敗した場合は単に環境を捨てるだけで良い.

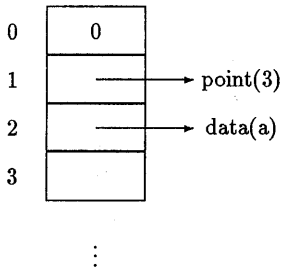

図 1: 変数環境

この表現法でプログラムの制御を代えて 2 通りのプログラムを書いた.

1. 逐次制御. 構造体の同士のユニフィケーションをする場合, その要素毎の unifier をフォークするが, 環境をその unifier 間でバケツリレーするので結局制御は逐次になってしまうプログラム.

2. 徹底逐次制御. 上記逐次制御を徹底したプログラム. 現 KL1 処理系のスケジューラ [5] を考慮し, 実行が中断しない限りスケジュールキューにはゴールが入らないようにした.

3.3 折衷表現

KL1 変数表現の利点は変数値の伝播を KL1 言語処理系に任せられる点であったが, この利点を基底項表現に取り入れようと試みたのがこの折衷表現である. 基底項表現では変数のデリファレンス[4]する必要があるが, 折衷表現ではこの操作を処理系に任せることができる.

この方式では, 環境は 1 ビットストリングを用い変数のタグを参照するために使う. ビットの意味は,

0 (値部は) 未定義

1 (値部は) 変数または具体値へのポインタ

である.

そして, オブジェクトレベルの変数の表現を 2 通り考えた.

1. 普通ライン型. 変数をストリングインデックス (変数番号) と値部の対で表現する. 値部には KL1 変数が割り当てられ, その変数が具体化されているかはビットストリングを参照することで判定する. 例えば変数表現を {N,Val}(N は変数番号,Val は変数値) とすると {1,Val1} と {2,Val2} をユニファイする場合, 環境の 1 要素目 (変数番号が 1 だから) のビットを 1 に書き換え Val1 と {2,Val2} を (メタレベルの) ユニファイする.

2. 急行ライン型. 普通ライン型において変数の値を参照する場合, 変数参照チェイン数だけストリングを参照しながらチェインを手繰って行かなければならない. これを処理系に任せたのが急行ライン型である. この方法では変数はストリングインデックス (変数番号) と値部と最終値部の三組で表現する. 最終値部はその変数とユニファイされている変数群の一つが変数以外のデータとユニファイされた時にそのデータとユニファイされる.

   例えば変数表現を {N,Val,FVal}(N は変数番号,Val は変数値,FVal は最終変数値) とすると {1,Val1,FVal1} と {2,Val2,FVal2} をユニファイする場合, 環境の 1 要素目のビットを 1 に書き換え Val1 を {2,Val2,FVal2} に具体化し FVal1 と FVal2 を (メタレベルの) ユニファイする. この後で {2,Val2,FVal2} とアトム a をユニファイした場合, 環境の 2 要素目 (変数番号が 2 だから) のビットを 1 に書き換え, Val2 を a に具体化し FVal2(FVal1) を a に具体化する.

4 各方式の評価

前節で述べた各方式について PSI-II 上の擬似マルチ PSI (マルチ PSI のシュミレータ) で測定を行った. その評価結果を示し (表 1,2) それについての考察を述べる. なお測定に用いた例題は次の通りである (A と B のユニフィケーション).

1 A = i(i(X1,X2),X3),
  B = i(i(i(Y1,Y2),Y3),i(i(Y3,Y1),i(Y4,Y1)))

2 32 個の異なった変数を持つ binary tree

3 A = p(h(X1,X1),h(X2,X2),Y2,Y3,X3),
  B = p(X2,X3,h(Y1,Y1),h(Y2,Y2),Y3)

4 A = i(i(X,X),i(i(Y,Y),i(V,i(W,Z)))),
  B = i(Y,i(Z,i(i(U,U),i(i(V,V),W))))

5 A = (X0,X1,X2,X3,X4,X5,X6,f(X0),f(X6)),
  B = (X1,X2,X3,X4,X5,X6,a,f(X6),f(X0))

6 A = X, B = f(a)

7 A = [X,x,Y,y,Z,z,U,u,V,v,W,w],
  B = [x,X,y,Y,z,Z,u,U,v,V,w,W]

8 A = f(X,x,Y,y,Z,z,U,u,V,v,W,w),
  B = f(x,X,y,Y,z,Z,u,U,v,V,w,W)

9 A = [X0,X1,X2,X3,X4,X5,X6,X7,X8,X9],
  B = [0,1,2,3,4,5,6,7,8,9]

10 A = f(X0,X1,X2,X3,X4,X5,X6,X7,X8,X9),
  B = f(0,1,2,3,4,5,6,7,8,9)

4.1 変数表現 - KL1 変数表現と基底項表現

オブジェクトレベルの変数を KL1 の変数で表現する方法の最大の利点は変数管理を KL1 言語処理系にまかることが期待できることである. しかし実は値の伝播を除いた部分の変数管理をする必要がある. ユニフィケー

[4]変数間の参照ポインタを最後までたどること

表 1: 1 ユニフィーケーション時間 (単位: マイクロ秒)

問題	1	2	3	4	5	6	7	8	9	10
KL1 組込み	154	2300	446	589	686	2.43	160	332	124	288
KL1 変数表現	771	21500	4210	4480	5160	104	3090	3150	2640	2670
基底項表現 - プロセス方式 -	3020	70600	25400	25500	12400	379	6500	6790	8110	8250
基底項表現 - 逐次制御 -	306	4180	878	1080	1510	34.2	620	844	487	697
基底項表現 - 徹底逐次制御 -	261	3510	831	915	1170	37.4	603	873	516	737
普通ライン型	345	4500	992	1260	1830	50.4	682	929	568	876
急行ライン型	398	4970	1160	1380	1600	66.2	689	915	671	867

表 2: 1 ユニフィーケーション時間比 (KL1 組込み = 1)

問題	1	2	3	4	5	6	7	8	9	10
KL1 組込み	1	1	1	1	1	1	1	1	1	1
KL1 変数表現	5.01	9.35	9.44	7.60	7.52	42.8	19.3	9.49	21.3	9.27
基底項表現 - プロセス方式 -	19.6	30.7	57.0	43.3	18.1	156	40.6	20.5	65.4	28.6
基底項表現 - 逐次制御 -	1.98	1.82	1.97	1.83	2.20	14.1	3.88	2.54	3.93	2.42
基底項表現 - 徹底逐次制御 -	1.69	1.52	1.86	1.55	1.71	15.4	3.77	2.63	4.16	2.56
普通ライン型	2.24	1.96	2.22	2.14	2.68	20.7	4.26	2.80	4.58	3.04
急行ライン型	2.58	2.16	1.17	2.34	2.33	27.2	4.31	2.76	5.41	3.01

ション中にオブジェクトレベルの変数に値を代入していくような Prolog 風のプログラムは実用的ではない. というのはユニフィケーションに失敗した場合, オブジェクトの変数の値を元に戻さないといけないからである. したがって, ユニフィケーションプログラムは成功することが分かってから実際にオブジェクトレベルの変数と値をユニファイしなければならない. つまり, 変数管理が必要になるわけである.

このような理由により KL1 変数表現の利点は全くないと言ってよい. 事実基底項表現に比べ 3 ～ 4 倍[5]低速になっている. これは構造体の皮むきをする度に unboundness の検査をしなければならないのと, ユニフィケーションに成功してから本当の変数に値を束縛する必要があるためだと考えられる.

4.2 変数値の伝播

オブジェクトレベルの変数値の伝播を KL1 言語処理系に任せようと試みたプログラムが, 普通ライン型と急行ライン型である. これらのプログラムは変数の KL1 変数表現と基底項表現の折衷表現というべきものである.

基底項表現に比べて,

1. 変数環境がビットストリングであるため消費メモリの節約が期待できる.
2. デレフ用変数 Val, FVal により変数チェインのたぐりの高速化が期待できる.

の 2 点において有利であると考えられる.

1 変数当たりの消費メモリは

普通ライン型 2 ワード + 1 ビット.

急行ライン型 3 ワード + 1 ビット.

基底項表現 - 逐次制御 - 未定義の場合, 2 ワード. それ以外の場合, 4 ワード

である. 折衷方式の方が優れているようだが同じ変数の出現が複数ある場合, 2 回目以降の 1 出現について普通ライン型では 2 ワード, 急行ライン型 では 3 ワード必要なのに対して, 基底項表現は 1 ワードですむので, 変数の出現回数が多ければ基底項表現が有利になる.

普通ライン型と急行ライン型 のスピードは基底項表現に較べ 1.1 ～ 1.5 倍低速である. 折衷方式では変数に対するタグが 0 と 1 と 2 種類なのに対し, 基底項方式では 0 と point と data の 3 種類である. 基底項表現の方がタグが多いため処理の最適化が少しできるためであると考えられる. したがって, 折衷表現でもタグの種類

[5]KL1 変数表現と基底項表現 - 逐次制御 - の比較

を 3 種類用意すると, メモリ使用量は増える[6]が, スピードの改善は期待できる.

4.3 変数管理 - プロセス方式 / ベクタ方式

プロセス方式による変数管理は並列処理言語らしいプログラミングができユニフィケーション問題の持つ並列性を自然に引き出すことができる. しかしプロセス方式はテーブル方式に比べ 1 桁以上[7]低速である. 標準的なユニフィケーションの並列度が 2 ~ 3 である [2] ことを考えるとプロセッサの台数効果を考慮に入れてもベクタ方式の方が勝っているといえるだろう.

なぜプロセス方式はベクタ方式に比べこんなにも低速なのであろうか? プロセス方式では変数の個数分の変数管理プロセスができる. 変数管理が必要になるとこの変数管理プロセスが起動される. 一方ベクタ方式では変数表をユニファイヤ自身が保持するためこのようなプロセスはできない. 変数管理プロセス自体の処理は軽いがこのプロセスは起動されるとすぐに休眠状態となる. このように, プロセス方式では軽いプロセスの切替えが頻繁に起こる.

現在の KL1 言語処理方式は頻繁にプロセス切替えが起こるようなプログラムに対しては有利な方式とはいえない. しかしプロセス切替えが頻繁に起こることを仮定した処理系 [11] ならばプロセス方式も一考の余地はある.

4.4 徹底逐次制御

今回作成したプログラムで最高速なものは基底項表現 - 徹底逐次制御 - である. これは完全逐次プログラムである. つまり, サスペンドしない限りどのゴールもスケジューリングキューに enqueue されない. リストの場合の処理について - 逐次制御 - と比べることによりその差異を考えてみる.

逐次制御 .

```
unify([XH|XT],[YH|YT], T,NT) :-
    unify(XH,YH, T,T1), unify(XT,YT, T1,NT).
```

徹底逐次制御 .

```
unify([XH|XT],[YH|YT], Cont, Env,NEnv) :-
    unify(XH,YH, [{XT,YT}|Cont], Env,NEnv).
```

現在の KL1 処理方式ではボディ部にユーザゴールが複数ある場合は最初の一つを除きスケジューリングキューに入れられる. そして最初の一つは引き続き実行される[8].

- 逐次制御 - の場合リストの Tail の処理はスケジューリングキューに enqueue される. 一方 - 徹底逐次制御 - では Tail の処理は自前のスタック (Cont) に積み, Head 部分の処理を行うどのゴールも enqueue されない. その代わりユニファイされる構造体の葉に到達した時に Cont からユニファイする対象を取り出してユニフィケーションを継続するプログラムを起動しなければならない. ユニファイする構造体の葉の数を L とするとこのプログラム は L 回起動される. 一方 - 逐次制御 - では L-1 回ゴールが enqueue される.

- 徹底逐次制御 - は - 逐次制御 - に比べ大体 10 ~ 20 % 高速である. これは完全逐次にした結果, データが常にレジスタ上にあるからである[9]. リダクション数は - 逐次制御 - に比べ L 回増えるのにも拘わらず高速なのはそれにも増して enqueue が重いということである[10].

このプログラム自体には台数効果は全く期待できない. しかし, 台数効果が最も期待できる変数プロセス管理方式はこれより 1 桁以上低速であることをふまえると徹底逐次制御プログラムの方に軍配を挙げざるをえない.

つまり, ユニフィケーションは現 KL1 処理系では粒度が小さ過ぎる問題ということになる. ユニフィケーションはメタプログラミングでは肝となる処理ではあるが負荷分散という観点からは, もう少し粒度の大きい仕事を分散させるのが有効であることが分かる.

5 プログラム例 - Prolog インタプリタ -

オブジェクトレベルの変数を基底項表現する場合の変数管理プログラムの記述は大変煩わしい. そこで我々はこれらのプログラムをライブラリ化した [13]. これは基底項表現 (徹底逐次制御) に基づいたプログラム群である. このライブラリが提供する機能は, ユニフィケーション (変数出現チェックあり / なし) やパターンマッチ, 環境操作やオブジェクトの入出力関連の機能などである.

このライブラリを使うことにより, 変数管理に煩わせられることなくメタプログラミングすることができる. 例えば, このライブラリを利用した OR- 並列 Prolog インタプリタ (図 2) では, メタプログラムは変数環境を持ち回るだけで, 変数管理の細かいところまで気を使う必要はない.

このインタプリタはライブラリの機能の内, ユニフィケーション機能とオブジェクト検索機能を使っている. ユニフィケーション機能は `meta#unify(X,Y, Env,NEnv)` を呼ぶことにより利用できる. ここで, X と Y はオブジェクトの表現で Env がユニフィケーション前の環境,NEnv がユニフィケーション後の環境である. またオブジェクト (Prolog 節) 検索機能は `'$PrologDataBase':get/3` を呼ぶことにより利用できる. これにより述語名とアリ

[6] 2 ビット必要

[7] 基底項表現 - プロセス方式 - と - 逐次制御 - の比較

[8] goto で jump するものと思ってよい

[9] PIM ではこうはならない [16]. したがって,PIM では - 徹底逐次制御 - は遅くなるであろう.

[10] enqueue_goal + proceed は execute より重いということ.

```
solveAnd([],[], Env, Sol) :- true | Sol = [Env].
solveAnd([],[Gs|Gss], Env, Sol) :- true | solveAnd(Gs,Gss, Env, Sol).
solveAnd([G|Gs],Gss, Env, Sol) :- true | clauses(G,Env, Clauses), solveOr(Clauses,G,Gs,Gss, Sol).

solveOr([{C,Env}|Cs], G,Gs,Gss, Sol) :- C = (H :- B) | Sol = {Sol1,Sol2},
    meta#unify(H,G, Env,Env1), expand(Env1, B,Gs,Gss, Sol1), solveOr(Cs, G,Gs,Gss, Sol2).
solveOr([], _,_,_, Sol) :- true | Sol = [].

expand(fail, _,_,_, Sol) :- true | Sol = [].  otherwise.
expand(Env, B,Gs,Gss, Sol) :- true | solveAnd(B,[Gs|Gss], Env, Sol).

clauses(G,Env, CLs) :- vector(G, Size), vector_element(G,0,F) |
    Arity := Size-1, clauses1({F,Arity},Env, CLs).
clauses(G,Env, CLs) :- atom(G) | clauses1({G,0},Env, CLs).

clauses1(FA,Env, CLs) :- true | clauses1(FA,0,Env, CLs).

clauses1(FA,M,Env, CLs) :- true |
    '$PrologDataBase':get({FA,M},Env, ExpEnv), clauses1Decide(ExpEnv, FA,M,Env, CLs).

clauses1Decide({}, _,_,_,CLs) :- true | CLs = [].
clauses1Decide(ExpEnv, FA,M,Env, CLs) :- ExpEnv = {_,_} | CLs = [ExpEnv|CLs1],
    M1 := M+1, clauses1(FA,M1,Env, CLs1).
```

図 2: OR- 並列 Prolog インタプリタ

ティよりその述語定義している節を検索することができる.

インタプリタの核部分は solveAnd/4 と solveOr/5 の二つである. Prolog 節の検索は clauses/3 が行う.

solveAnd/4 ゴールの列 Goals とゴール列のスタック GoalsStack とその環境 Environment を受け取り解 Solution を計算する. ゴール列を先頭から順に解いていく (第二, 三節). ゴール列が空になったら解が見つかったことになるのでその時の環境を返す (第一節).

solveOr/5 ゴール (G) と節のヘッド (H) をユニファイ (meta#unify(H,G, Env,Env1)) し, 成功すればその時の環境でボディ部を解く (expand/3 第二節).

clauses/3 ゴール G の候補節のリストを CLs に返す.

6 考察

KL1 でメタプログラミングする際のオブジェクトレベルの変数の表現法に焦点を当て, 基底項表現した場合のユニフィケーションプログラムを幾つか KL1 で記述しその性能比較を行った. それらの内最も高速だったプログラムは徹底逐次制御プログラムであり, これは Lisp でユニフィケーションプログラムを記述した場合のプログラムに非常に近い.

オブジェクトをその表現と環境の対で表現するこの手法は OR- 並列制御にはむいているが AND- 並列制御にはむかない. それはメタプログラム自身が環境を持ち回らなければならないことからきている. この手法を利用したメタプログラムの典型は, 例えば, 述語 meta1 と meta2 が操作するオブジェクトが AND 関係にある場合,

```
meta1(.., Env,Env1), meta2(.., Env1,Env2), ...
```

という感じになる. この場合, meta1 が何らかの処理をすることにより, 環境が Env から Env1 になり, その環境下で meta2 の操作で環境が Env2 となるように処理が進む. meta2 は Env1 が決定するまで, つまり meta1 の処理がほぼ終了するまで処理を開始することができない. 実質的には meta1 と meta2 の処理は逐次になってしまう.

プログラマの気持ちとしては

```
meta1(.., Env,Env1), meta2(.., Env,Env2), ...
```

と書きたいところだが, このように書くとあるオブジェクトに対し meta1 の処理をすると環境が Env から Env1 になり一方, meta2 の処理をすると環境が Env から Env2 になる, という意味になる. つまり meta1 と meta2 の操作するオブジェクトの関係は OR 関係になってしまう. この性質を利用したのが OR- 並列 Prolog インタプリタ (図 2) である.

AND- 並列を引き出すには, 環境管理プロセスを生成し環境の参照・更新はそのプロセスへのメッセージによることにすればよい. しかし, メッセージ通信による計

算は並列言語らしいプログラムは書けるが, 現 KL1 処理系では効率を損ねる. またこの方法で OR- 並列性を引き出す場合には環境管理プロセスのコピーが必要となるので, 効率的な実現は困難である.

このようなことを考えると, オブジェクトの表現 - 環境方式で AND- 並列を効率良く引き出すには KL1 の機能拡張が必要であると思われる. その拡張機能は環境 (KL1 のベクタ型) に対する test-unification のようなものである.

KL1 は並列言語と論理型言語双方の性質を備えてはいる. しかし論理型言語の特徴である論理変数は, 並列動作するプロセス間の同期制御の実現には大きな役割を果たしているが, Prolog のように論理変数をオブジェクトレベルの変数にも用いることは一般的にはできない. これはユニフィケーションの成功 / 失敗を条件に条件分岐するプログラムが書けないからである.

このように考えると

$KL1 = Prolog -$ test-unification
$+$ 同期機構 (one-way unification)

とみなすことができ, またそうみなした方が効率の良いプログラムが書ける. これは FCP の

$FCP = Prolog +$ 同期機構 (read-only annotation)

とは対照的である. したがって, FCP は KL1 と比べるとメタプログラミングしやすいのは当然といえるだろ.

しかしその代償として, FCP の (分散環境化での) 実現は KL1 と比べると格段に難しい. FCP のガード部での full test-unification は論理変数に対する長時間のロック機構[11]が必要となるからである.

このことは FCP は KL1 に比べてその言語処理系自体が重くなることを意味する. 上の 2 式より大雑把にいって

$FCP = KL1 +$ test-unification

がいえるが変数管理プログラムを KL1 で記述することは, FCP の full test-unification の部分を KL1 で記述することに相当する. そして, 我々の

メタプログラム $= KL1 +$ 変数管理プログラム

によるアプローチと FCP によるメタプログラミングのアプローチでどちらが最終的に効率の良いプログラムが書けるかは, 今後の経験より明らかとなるだろう.

勿論, 静的に情報の流れが十分に分かっているような対象やそのような対象に問題が還元できる場合, KL1 は記述言語として非常に適した並列言語といえるだろう [3, 4]. そのような問題領域として, データベース, 自然言語処理, 自動プログラミング等がある. このような KL1 の言語機能を有効に生かせる問題領域を開拓していくのも今後の研究課題である.

[11] 構造体に同士がユニファイ可能かどうか確かめてから実際に変数に値を代入するまでのロック

参考文献

[1] Chikayama, T., Sato, H., Miyazaki, T.: "Overview of the Parallel Inference Machine Operating System (PIMOS)", In *Proc of FGCS'88*, 1988.

[2] Fujita, H.: "Parallel Unification and Meta-Interpreters in GHC", ICOT TR-468, 1989.

[3] Fujita, H., Hasegawa, R.: "A Model Generation Theorem Prover Using A Ramified-Stack Algorithm" ICOT TR, 1990.

[4] Hasegawa, R., Fujita, H., Fujita, M.: "A Parallel Theorem Prover in KL1 and Its Application to Program Synthesis", ICOT TR-588, 1990.

[5] Nakajima, K., Inamura, Y., Ichiyoshi, N., Rokusawa, K., Chikayama, T.: "Distributed Implementation of KL1 on the Multi-PSI/V2", In *Proc of 6th ICLP*, 1989.

[6] Shapiro, E.Y.: "Or-Parallel Prolog in Flat Concurrent Prolog", In *Concurrent Prolog Collected Papers* Vol.2, 1987.

[7] Shapiro, E.Y.: "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys* Vol.21,No.3, 1989.

[8] Tanaka, J.: "Meta-interpreters and Reflective Operations in GHC", In *Proc of FGCS'88*, 1988.

[9] Ueda, K.: "Making Exhaustive Search Programs Deterministic", In *Proc. of 3rd ICLP*, LNCS 255, Springer-Verlag, 1986.

[10] Ueda, K.: "Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard" ICOT TR-208, 1986.

[11] Ueda, K., Morita, M.: "A New Implementation Technique for Flat GHC", In *Proc of 7th ICLP*, 1990.

[12] 越村 三幸, 藤田 博, 長谷川 隆三: "KL1 上のユニフィケーションプログラムとその評価", ICOT TM, 1990.

[13] 越村 三幸, 藤田 博, 長谷川 隆三: "メタライブラリ・マニュアル", ICOT TM, 1990.

[14] 竹内 彰一, 高橋 和子, 清水 広之: "並列問題解決用言語 ANDOR-II", ICOT TR-235, 1987.

[15] 枚田 正宏: "並列記号処理言語 Oc とその自己記述", コンピュータソフトウェア, Vol.4, No.3, 1987.

[16] 平野喜芳, 後藤厚宏: "並列論理型言語 KL1 のコンパイル方式の改良", JSPP'90, 1990.

[17] 渕 一博: "KL1 プログラミング雑感 - prover の並列化の体験より -", KL1 Programming Workshop'90, ICOT TR-569, 1990.