ICOT Technical Memorandum: TM-0898

TM-0898

KL1講習会テキスト
ハッカーのための基礎データ編

松本　幸則

July, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

KL1講習会テキスト
ハッカーのための基礎データ編

マルチPSIにおけるKL1の各種基本処理の重さについて
第0.4版

新世代コンピュータ技術開発機構
第7研究室
松本幸則[1]

1989年12月18日 revised 1990年5月9日

[1] *yumatumo@icot21.icot.jp*

はじめに

本書では、KL1 プログラムを書こうとする時、どのような書き方をすれば処理性能がより良く(または悪く)なるか、また、具体的な性能値がどの程度になるかなどを判断する時の助けとなる基礎データを提供する。

本書で示した各種の値は、マルチ PSI 上の KL1 処理系(ファームウェア 1.12 版)における実測値に基づいている。

KL1 の各種基本処理の重さを表現するのに append プログラムの 1 リダクション処理時間を 1 とした相対値で表現しているが、この値は、いうまでもなく KL1 処理系の実現方法に依存する。従って上述の相対値は PIM 処理系では異なってくる。但し、多くの基本処理については、何が重く何が軽いかは PIM においても同様の傾向を生ずると期待される。また、マルチ PSI 上の現処理系への依存度が特に大きい数値についてはそのことを明記するように心掛けた。

本書は KL1 講習会中級編修了相当の知識を有する読者を対象としている。本書が、諸兄の KL1 プログラミングの一助となれば幸いである。

- 目次

0. 処理コストの基準について
1. unification について
2. 組み込み述語関係
3. ゴールのサスペンド
4. 種々のサイズの影響
5. priority の使用によるもの
6. ゴール、データの PE 間の受け渡し
7. MRB(Multiple Reference Bit) について
8. マージャについて

第 0 章

処理コストの基準について

- 処理コストの基準 ——append 里 (LI = Logical Inference)
- Multi-PSI の 1PE 当たりの性能は 130K append 里 PS(LIPS)
- ところで append 里とはどの程度の処理?

例

```
append([],L2,L3):-true|L3 = L2.
append([X|L1],L2,L3):-true|
        L3=[X|LL3],append(L1,L2,LL3).
```

- 2 番目の節の実行に必要な処理
  - 第 1 引数の示す先が list かどうかチェック ——list でないデータ型に instantiate されていれば fail、未定義ならば suspend
  - 第 1 引数が黒い場合 —— 新しい cons セルの領域を確保する。
  - 第 1 引数が白い場合 —— 新しい cons セルの領域として、第 1 引数の示す list のセルを使用する
  - 第 3 引数が示す先の内容が、先ほど確保した新しい cons セルを指すようにする。
  - X が示す内容を新しい cons セルの car 部に書く。
  - L1 の内容を呼び出すゴールの第 1 引数にセット
  - 新しい cons セルの cdr 部を指すポインタを第 3 引数にセット
  - append の再帰呼びだし (jump)

第 1 章

unification について

1.1 ボディ部の unification について

以下のプログラムにおける、ボディ部の unification の重さ

```
p(A,B):-true|A=B.
```

A,B とも変数同士　　軽い (約 0.3 append 里)
A,B の片方が変数　　軽い (約 0.2 append 里)
(但し hook 変数を instantiate した場合は、多少重い)

A,B どちらも instantiate 済み (普通は使わない)
atom　　軽い (約 0.1 append 里)
integer　　軽い (約 0.1 append 里)
string　　やや重い (サイズに依存)
SW(1W 32bit) で 0.6 append 里
16W で 0.8 append 里
以下は general unification となるため重い。
list 全体　　重い (構造に依存)
8 セルで 10append 里
16 セルで 20append 里
vector　　重い (構造に依存)
8 要素の平坦なもので 22 append 里
16 要素で 44 append 里

1.2 ガード部の unification について

以下のプログラムにおける、ガード部の unification の重さ

```
p(A,A):-true|...
```

変数同士　　suspend
片方が変数　　suspend

どちらも instantiate 済み

atom	軽い (約 0.1 append 里)
integer	軽い (約 0.1 append 里)
list 全体	重い (構造に依存)
	8 セルで 5append 里
	16 セルで 10append 里
vector	重い (構造に依存)
	但しボディ部のものよりは軽い
	8 要素の平坦なもので 4.0 append 里
	16 要素で 7.9 append 里
string	やや重い (サイズに依存)
	8W で 0.6 append 里
	16W で 0.9 append 里

なお、以下のように、ガード部に陽にコードを記述した場合には、上記の約半分の処理の重さとなる。例えば、節 3 で約 0.3 append 里

```
p(a):- true|....          % 1
p([a,b,c]):-true|....     % 2
p([a|B]):- true|....      % 3
p({a,b,c}):-true|....     % 4
```

第 2 章

組み込み述語関係

2.1 タイプチェック

wait,atom,integer,list	軽い (0.1 append 里以下)
vector,string	それぞれ 0.1append 里、0.3append 里

2.2 比較、算術演算

(ボディ部に現れる場合はサスペンドのない場合)

`<,>,=<,>=,=:=,=\=,\=,+,-,<<,>>, /\,\/,xor,\`

約 0.1 ～ 0.15 append 里

`*,/,mod`

他の比較、算術演算よりは多少重い
積で 0.5 append 里、商・剰余で 1 append 里

2.3 ベクタ関係

new_vector	約 1.1 + 0.07 × size append 里
vector_element(B,G)	set_vector_element より少し軽い ガード部で約半分、ボディ部ではほぼ同じ
set_vector_element	
対象ベクタが白	約 0.7 ～ 0.8 append 里
対象ベクタが黒	
サイズが 6 以下 *Multi-PSI*	約 1.0 + 0.3 × size append 里
サイズが 7 以上 *Multi-PSI*	約 1.4 append 里

2.4 ストリング関係

new_string	約 1.0 + 0.1 × ワード size(32bit) append 里
string_element(B,G)	set_string_element より少し軽い
	ガード部で約 0.7 append 里 (白黒ほぼ同じ)
	ボディ部で約 0.9 append 里 (白黒ほぼ同じ)
set_string_element	
対象ストリングが白	軽い (約 1.0 appcnd 里)
対象ストリングが黒	1.6 + 0.1 × size append 里

第 3 章

ゴールのサスペンド

3.1 サスペンドコスト

単一サスペンド	約 1.8 append 里 /1 回
組込述語の単一サスペンド	約 3.3 append 里 /1 回
多重 (2 重) サスペンド	約 3.7 append 里 /1 回
多重 (4 重) サスペンド	約 6.8 append 里 /1 回
多重 (8 重) サスペンド	約 11.7 append 里 /1 回

サスペンドは高くつくので、なるべく起こさない方が良い。一度サスペンドすると、実行再開時には、ガードのチェックを最初からやり直す。 — どうせサスペンドするのなら、早いうちに。(尚、ガード部の実行は基本的には左から右へ行なう。節の順序は、otherwise,alternatively がない限り意味を持たないことに注意。コンパイラが自動的に並び変えることがある。)

教訓

- 組み込み述語をガード部に書くかボディ部に書くかにより、変数待ちに関してその節の意味が異なることに注意

  例

```
  p(A,...):- A1 := A + 1| q(A1,...).          % 1
  p(A,...):-true| A1 := A + 1, q(A1,...).     % 2
% q が A1 の具体化を待って動くものであれば %1 が良い
% q が A1 に関係無く動けるものであれば %2 が良い
```

- 多重待ちの変数については、その中で具体化され易い変数を、引数順序の若い側に置く方が処理が軽い。

  例 1

```
a(...):- true|.... ,p(X,Y,Z),...

 p(a,_,_):-true|...
 p(_,b,_):-true|...
 p(_,_,c):-true|...
```

例2

```
a(...):- true|.... ,p(Z,Y,X),...

 p(c,_,_):-true|...
 p(_,b,_):-true|...
 p(_,_,a):-true|...
```

% 具体化され易さがX,Y,Zの順であれば、例1の方が処理が軽い

- 単独待ちの変数については、その中でsuspendし易い変数を、引数順序の若い側に置く方が処理が軽い。

例3

```
a(...):- true|.... ,p(X,Y,Z),...

 p(a,b,c):-true|...
```

例4

```
a(...):- true|.... ,p(Z,Y,X),...

 p(c,b,a):-true|...
```

% suspendし易さがX,Y,Zであれば、例3の方が処理が軽い

第 4 章

種々のサイズの影響

4.1 述語の引き数の数に依存する点

引き数の数が 32 個[1] を超える *Multi-PSI*	compile error
引き数の数が 12 個以上 *Multi-PSI*	reduction が多少遅くなる (11 番目以降の引数をベクタ化)

教訓

- 引き数の中で、渡されたゴールがすぐ使うことなく長時間引き回すようなものを纏めてベクタにしておくと処理が多少軽くなる

例

```
p(A1,A2,.....,A11):-true| ...,p q(.....)... .
                          ↓
p(A1,A2,..,A5,Vect):-true| ..., q(....,Vect)... .
                  (ただし Vect={A6,A7,..,A11})
```

4.2 ベクタのサイズとアクセス速度

MRB が黒且つサイズが 7 以上 *Multi-PSI*	特定要素へのアクセスが遅くなる場合有り (黒ベクタは mutable 構造[2] をとるため)
MRB が白またはサイズが 6 以下 *Multi-PSI*	コンスタントオーダでアクセス 組み込み述語参照

[1]実際には、意味のあるプログラムではワーキング用のレジスタが幾つか必要になるため、32 個以下でも compile エラーが起こることがある。PIM ではこの問題は解消される予定。

[2]mutable 構造: ベクタの一部を変更して新しいベクタを作る時、通常は無変更要素を全てコピーするが、この方法では、変更のない部分を共有し、差分だけを別に作って新たなベクタを表現する。

4.3 ストリングのサイズとアクセス速度

サイズに関係無く一定　　組み込み述語参照

第 5 章

priority の使用によるもの

- priority 指定時の、論理 priority と物理 priority の変換処理によるオーバヘッドの存在 —— 約 0.9 append 里 *Multi-PSI* [1]
- 使用されている priority キュー間のリンク変更によるオーバヘッドの存在 —— 約 3 append 里前後

[1] 乗算器がつけば速くなる

第 6 章

ゴール, データの PE 間の受け渡し [1][2]

(注：これらのコストは何割か改善の余地ありといわれている)

6.1 ゴール関係

ゴール投げ出し　　約 20 + K_t × arity 数 append 里
ゴール受け取り　　約 10 + K_r × arity 数 append 里
　引き数がアトミックデータ
　　$K_t + K_r$ = 約 2
　引き数が白の変数　　$K_t + K_r$ = 約 2.6
　引き数が黒の変数　　$K_t + K_r$ = 約 4
　(引き数が構造体の場合も変数と同じと考えて良い。但しゴールの
　投げ出し時にはポインタのみ移動)

6.2 データ関係

以下の値は、別プロセッサに read 要求を出し、その値を受け取れるまでの処理時間を示す。

- リストの PE 間移動

白い cons セル
　car 部がアトミック　　約 24 append 里 / 個
　car 部がポインタ　　約 30 append 里 / 個
黒い cons セル
　car 部がアトミック　　約 28 append 里 / 個
　car 部がポインタ　　約 35 append 里 / 個

- ベクタの PE 間移動

白いベクタ
　要素がアトミック　　約 18 + 1.9 × size append 里 / 個
　要素がポインタ　　約 18 + 2.7 × size append 里 / 個
黒いベクタ
　要素がアトミック　　約 21 + 1.9 × size append 里 / 個
　要素がポインタ　　約 21 + 4.9 × size append 里 / 個

- ストリングの PE 間移動

白いストリング	約 17 + 0.9 ×ワード size(32bit)append 里 / 個
黒いストリング	約 20 + 0.9 ×ワード size append 里 / 個

6.3 ゴール、データの PE 間の受け渡しに際しての処理の概略

6.3.1 変数セルについて

- 生成時期 — ボディ部に初出の変数が現れた時。
- 生成場所 — 変数が初出したゴールが実行される PE。

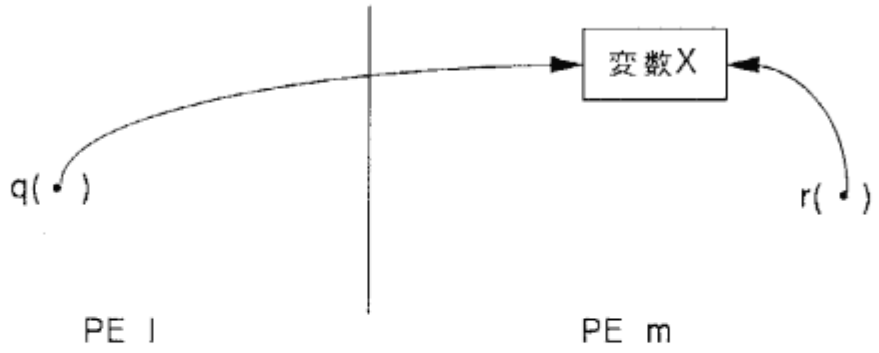
例 p :- true | q(X), r(X).

変数 X に対応するセルが、ゴール p が実行されたプロセッサの中に生成される。

6.3.2 外部参照ポインタについて (いつ作成されるか)

1. ゴール送出の時

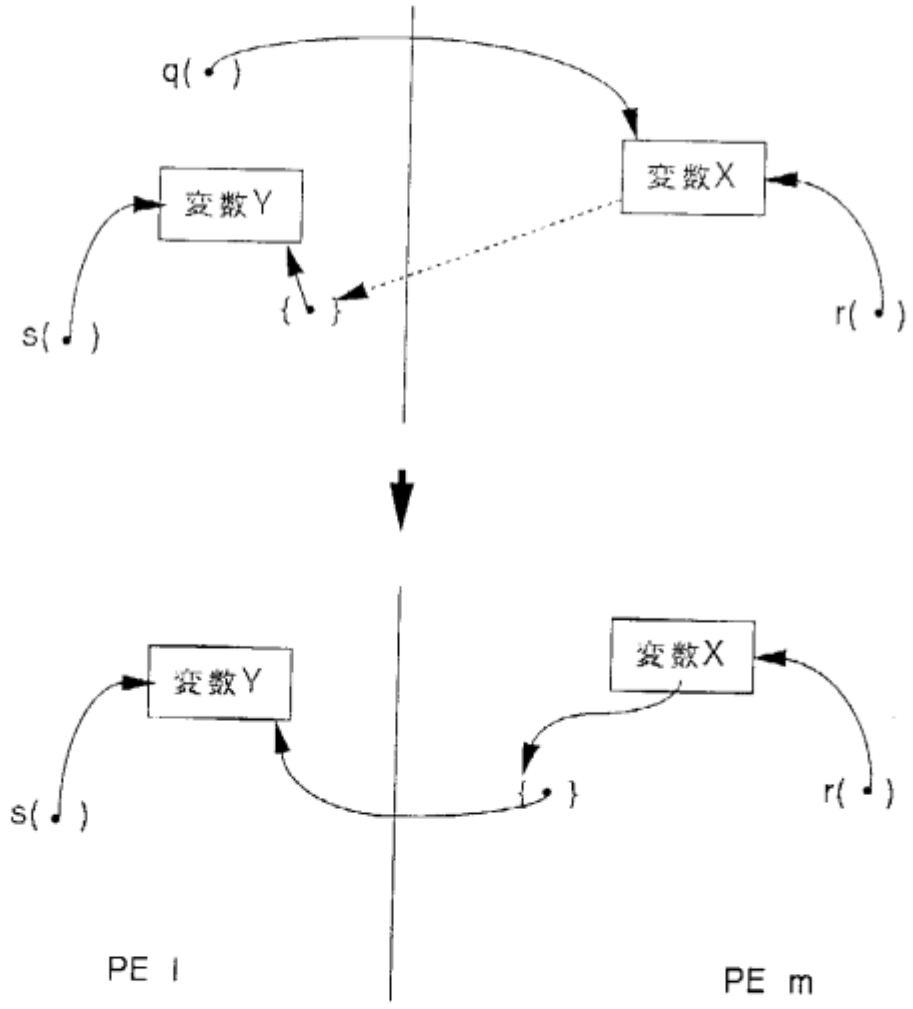
```
例  p :- true| q(X)@processor(1),r(X).
                    % P in processor m
```



2. 構造体(ネストしたもの、または変数を含むもの)がプロセッサを渡る時

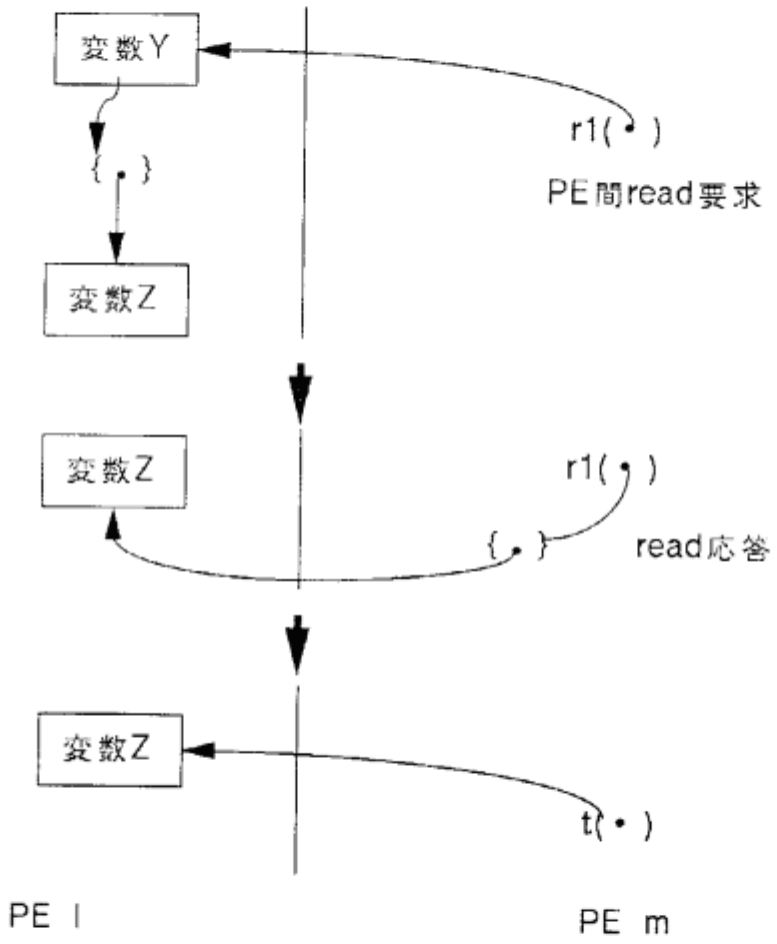
- アクティブユニファイがプロセッサ渡りになるとき

例 q(X):- true| X={Y},S(Y). % 前例の続き



- パッシブユニファイがプロセッサ間readとなる時

```
例  s(Y) :- true | Y={Z},r1(Y)@processor(m)
                        ,something(Z).
    r1({Z}) :- true | t(Z).
```



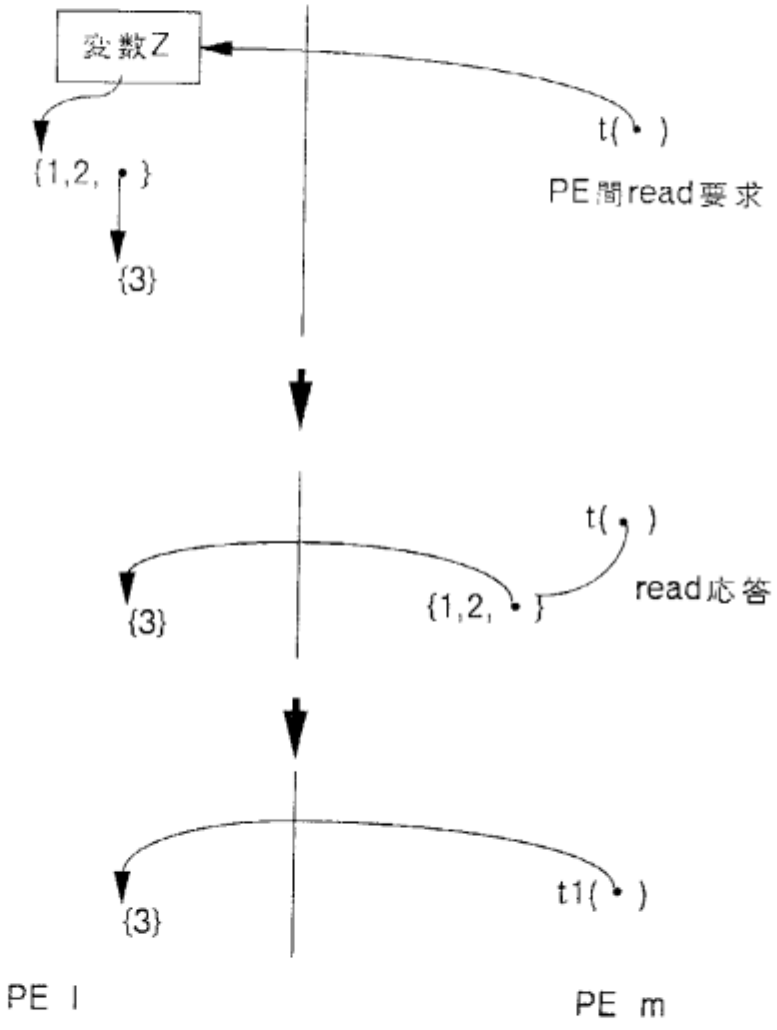
変数Yが未定義ならば、PE間readの応答はYが定義されるまで待たされる。

変数Yが例のように構造体に定義されていれば、その構造の殻部分と、その要素である変数へのポインタが返される。

6.3.3 構造体のPE間転送について(一層転送の原則)

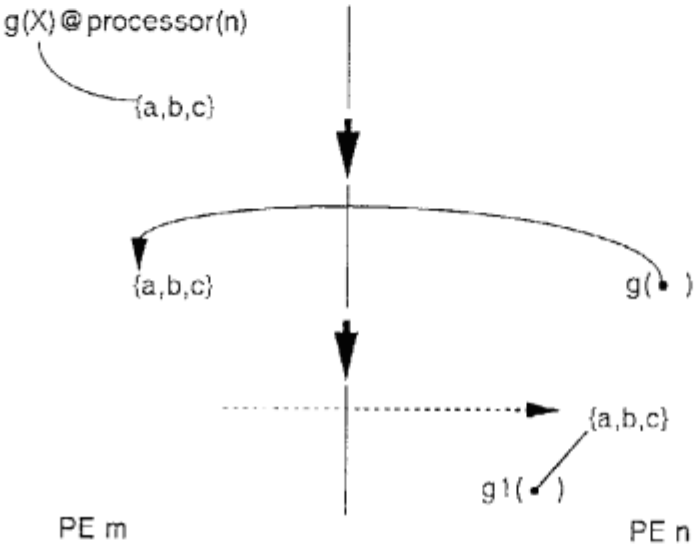
1. ネストした構造体は一度のPE間read要求に対して1階層分のみ送られる。

例 `t({1,2,A}):-true|t1(A).`



2. ゴールの引き数が構造体の場合 — そのポインタのみが送られる (ゴールも一種の構造体)。なお、その構造体本体は、送られたゴールの子孫が値を必要とした時に初めて PE 間 read 要求により送られる。

例
```
...,g(X)@processor(n).
g(X) :- true | g1(X).
g1({a,b,Z}) :- true | g2(Z).
```



(注) 一層転送は、「データは必要な時にのみ送る」と言う基本思想による。ネストした構造体全体が必要な時は転送レスポンスが悪いことに注意。(特にリスト)

6.3.4 プログラムコードについて

- プログラムコードは module 単位で管理。
- module は一種のベクタ (コード部は平坦な一層ベクタをなし、プログラム中に陽に書かれた定数ベクタ、ストリングはネストした構造として現れる)
- プログラムコードの転送時期 — 投げられたゴールが、その参照 module のコードの存在していない PE に到着後、最初に実行される時 (一層分のみ)
- プログラム中に陽に書かれた定数ベクタ、ストリングの本体はその値が必要になった時に初めて転送 (一層転送の原則)
- 同一プログラムコードの重複転送は発生しないようなインプリメント。
- PE 内 GC により、どのゴールからも参照されていないプログラムコードは消滅。

教訓

- 以下の2つの例では、後者の方がレスポンスが良い

```
p([{A,B}|T],..):-true|  cal(A,B,...),      % 1
                          ...
                        p(T,...).

p({A,B,T},..):- true|   cal(A,B,...),      % 2
                          ...
                        p(T,..).
```

6.4 輸出入表関係

- 輸出入表のサイズの限界は mpsi.param にて設定

教訓

- 輸出入表のサイズの限界を越えて変数を輸出入することはできない。変数輸出入の絶対数に注意
- 輸出入表の回収(プロセッサ渡りの参照ポインタの回収)が難しくなるような coding には注意 —— 輸入側で複数参照を作って且つ誰も使わずに捨てるような場合 —— その PE のローカル GC まで回収不可能

第 7 章

MRB(Multiple Reference Bit) について

7.1 大まかな MRB の白黒の判定基準

- 一つの節のガード部に現れた (複数でも可能) 変数が、ボディー部に 2 回以上現れた場合 —— MRB 黒
- 一つの節のガード部に現れた (複数でも可能) 変数が、ボディー部に 1 回のみ現れた場合 —— MRB 白
- 一つの節のボディー部のみに同じ変数が 3 回以上現れた場合 —— MRB 黒
- 一つの節のボディー部のみに同じ変数が 2 回 (以下) 現れた場合 —— MRB 白

以上を纏めると、一つの節内の同じ変数の出現数を数え、3 回以上の場合黒となる。但し、ガード部に同じ変数が何度現れても、それは一回と数える。

例

```
p(A,B):- guard_builtin(B,C)|
         q(B,D),
         r(C,E),
         s(C,F),
         u(F,F,G),
         p(E,G).
```

% A: 白ならば領域回収,B:2 回 = 白,C:3 回 = 黒,D:1 回 = 白,E:2 回 = 白,F:3 回 = 黒,G:2 回 = 白

- 注意の必要な場合
  - ガード部に passive unification がある場合で、かつ、unify 対象の変数がボディ部で使用される場合。—— 変数同士の unification の場合には、同じ変数としてカウントし、白黒判定

    ```
    p(A):- B=A,C=A,D=A|q(B,C),r(D).
    ```

  - ガード部に構造体の要素の取り出しがある場合で、かつ、構造体本体も取り出した要素もボディ部で使用される場合。—— 取り出した要素が黒くなる。本体は通常のカウント方法。要素の参照数を数える時は本体の参照数も加える。

```
p(A):-A=[X|Y]|q(A),r(X,Y).
p(V,N):-vector_element(V,N,Elm)|q(V,Elm).
```

例

```
copy(A,B,C):-atom(A)| B=A,C=A.
copy(A,B,C):-integer(A)| B=A,C=A.

copy([X|Y],B,C):-true|  B=[X1|Y1],C=[X2|Y2],
                        copy(X,X1,X2),copy(Y,Y1,Y2).
copy(A,B,C):-string(A,S,ES)| ...(省略).

copy(A,B,C):-vector(A,N)|
                  new_vector(B1,N),
                  vcpy(A,N,B1,B,C).

vcpy(S,0,D,ND1,ND2):-true| ND1=S,ND2=D.
vcpy(S,N,D,ND1,ND2):- N>0| N1:=N-1,
                  set_vector_element(S,N1,E,Es,Ds),
                  copy(E,Es,E1),
                  set_vector_element(D,N1,_,E1,D1),
                  vcpy(Ds,N1,D1,ND1,ND2).
```

教訓

- 一般に大きなベクタ、ストリングなどについては、メモリ効率の観点から多重参照は避けた方が良い
  - 頻繁に使い捨てるような構造体は (回収できるように) 白になる方が良い。—— 例 ストリーム
  - 大きな構造で書き換えがあるような構造体は (コピーの手間を無くすために) 白になる方が良い。—— 例 ベクタでデータベースを表現

第 8 章

マージャ

8.1 マージャについて [1]

入力ストリームにデータが 1 個ながれてきた時のコスト

ユーザ定義マージャ	
黒データマージ	約 7.5 append 里
白データマージ	約 7 append 里
組み込みマージャ	
黒データマージ	約 2 appcnd 里
白データマージ	約 1 append 里

組み込みマージャプロセス —2 引数マージ述語を呼んだ時点で特別のプロセスが作られる。

マージャプロセスは、常に次の入力を要求している。離れた PE 上にマージャプロセスがあり、マージャの入力ストリーム変数がこちらの PE にある場合、PE 間 read 要求が常に届いている —eager な PE 間データ転送に使える。

教訓

- マージャはネストさせずに、上位で一度マージプロセスを生成後は、ベクタを unify した方が高速

例

```
p(Data,Out,C):- C =\= 0| merge(S1,S2,Out),  % 1
                         cal(Data,D1,S1),
                         C1 := C-1,
                         p(D1,S2,C1).
p(_,Out,0):- true| Out=[].


p(Data,Out,C):-true| merge(S,Out),  % 2
                     p1(Data,S,C).
p1(Data,Out,C):- C =\= 0| Out={S1,S2},
                          cal(Data,D1,S1),
                          C1 := C-1,
                          p1(D1,S2,C1).
p1(_,Out,0):- true| Out=[].
```

第 9 章

その他

9.1 ゴールレコード生成+エンキュー+デキューのコスト [1]

約 0.7 append 里

参考文献

1 : Y.Inamura,N.Ichiyoshi,K.Rokusawa,K.Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In Proceedings of the North American Conference of Logic Programming '89

2 : 市吉伸行, 石塚裕一, 佐藤令子. マルチ PSI/V2 第 1 次評価初期測定結果報告 (ICOT 検討会資料 PSM-I-P-EVL-012)

3 : 稲村 雄. KL1 での構造体仕様 (ICOT 検討会資料 PSM-I-P-TMM-140)

4 : KL1 講習会テキスト　入門、初級、中級編 (ICOT 編)