TR-I-0180

並列自然言語処理における単一化手法の高速化

藤岡 孝子* 苫米地 英人 古瀬蔵 飯田 仁

Takako FUJIOKA, Hideto TOMABECHI, Osamu FURUSE, Hitoshi IIDA

1990.9.14

概要

本報告では単一化に基づく自然言語処理の中で、最も処理時間の割合が大きい単一化手続きを高速化するための、Tomabechi の時間差準破壊型単一化アルゴリズムについて考察し、これを並列に処理する手法を提案する。また、日本語文解析における単一化手続きにおいてこの並列アルゴリズムを用いて実験を行ない、並列化の効果と課題について述べる。

ATR 自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

*早稲田大学大学院理工学研究科電気工学分野修士課程

1 はじめに

本稿では、従来の unification algorithm を考察した上で、さらに高速化、効率化が期待でき、また並列化に適している Tomabechi の時間差準破壊型単一化アルゴリズムについて述べ、これを並列に処理するアルゴリズムを提案する。また、Earley の日本語文解析 parser においてこれを適用した時の実験をおこなった結果から得られた並列化における問題および効果について考察する。

自然言語処理の分野では最近盛んに Unification-based Grammar による処理が行なわれるようになってきている。現在主に利用されている Unification-based Grammar にはＬＦＧ[Bresnan, 1982]、ＨＰＳＧ[Pollard and Sag, 1987]などがある。

Unification-based Grammar による自然言語処理では、文の構造が素性の Subsumption（包含関係）で表され、Partial Order の monotonic な数学的モデルとして論理的に整然としており厳密な検証が可能であると言う利点がある。しかし、自然言語そのものは本質的に monotonic でない現象を含んでいると言う事実を忘れることはできない。さまざまな自然言語の現象を扱うためには unification のアルゴリズムの中に付加的な例外処理を挿入する手法が考えられるが、こうすると純粋な数学的モデルの世界を崩すことになってしまうと言う問題がある。

また一方計算機による自然言語処理として Unification-based Grammar を採用することには次のような利点がある。まず数学的に厳密であるので計算機上の演算に向いている。また言語学としての文法理論が現在 Unification-based Grammar を基盤にして発展してきている事情を考えると、parsing system を Unification-based にすることによって言語理論の研究成果を積極的に利用しやすいと言える。

しかしその一方では言語理論そのものが本来の unification では扱えない制約（constraint）を採り入れることにより拡張されてきているため純粋な Unification-based parser ではもはや通用しないというジレンマが起こってきている。 さらに、一般の Unification-based system の場合、文の素性構造（feature structure = fs）は graph として表され、system の処理のほとんどはこの graph の unificaiton の操作に費やされるが、grammar や lexicon を表す fs と入力文の fs の unification は非常に多くの回数実行される上、解析がトップダウンであれボトムアップであれ同じ fs が何回も使用されるので、元の文法の情報を変化させずに保持する必要性がある。つまり、一つの graph に対して original と unification の結果との２つの情報が常に必要とされる。したがって unification を行なう際には copying が不可欠となるが、そのコストが高いことから処理時間が膨大にかかってしまう。実際 parsing 全体の７割から８割以上の時間が一般に unification に割かれている。このような事情から、unification のアルゴリズムの高速化が大きな課題となっており、主に copying のコストを下げるさまざまなアルゴリズムが提案されてきた。

これとは別に、計算機による処理速度を上げる方法として、並列処理を行なうことが一般に行なわれていることから、unification を並列に処理することによる高速化にも目を向けるべきであろうと思われる。

そこで、unification algorithm を並列処理という点から考察し、並列化に適していると思われる Tomabechi の時間差準破壊型アルゴリズムを並列処理する手法を考案することによる高速化を試みた。

2 従来の unification algorithm について

最も基本的な unification alogorithm は、元の情報を残すために unify する２つの graph のそれぞれ copy を作り、この copy に対して破壊的（destructive）な操作を行なうというものである。従来の unification alogrithm ではいずれも copy のコスト削減に重点をおいた高速化が試みられてきた。しかし、graph の一つのノードから出発するそれぞれの arc の処理は本質的に order-in-

dependent であることから、これら一つ一つに対し並列に再帰的な unification 処理を行なうことが可能であり、これによる高速化が期待できると考えられる。よって並列化可能なアルゴリズムを考える必要があるといえる。

この観点から従来の unification algorithm の主なものを考察するとこれらは copy のコスト削減の trade-off としてのコストが高いだけでなく、局所化されていない共有な global 領域を頻繁に使うという特徴があり、並列化に不適であることがわかる。以下その特徴を整理して述べる。

- Pereira によるアルゴリズム[Pereira, 1985]

  destructive unification であるが、original の graph が unification によって変化すると、その初期値と更新値の組を environment という global 変数にすべて記憶させる。これにより unification の結果の値を後で計算するというものである。このアルゴリズムでは copy をしない代わりに、ノードの数 n が大きくなると計算の overhead が（ｌｏｇ（ｎ）＋Ｃ）と大きくなるという欠点がある。

- Karttunen & Kay の structure sharing アルゴリズム[Karttunen and Kay, 1985]

  これは graph 間で、ある同じ部分構造を持っているもの同士は、データ構造も共有させることによって、copy を最小限にしようというアルゴリズムである。しかし、sharing のために解析木を 2 進木にし、その相対アドレスを利用するなどの繁雑な操作が必要であり、また、並列プロセスが同時に共有データ構造に頻繁にアクセスするために並列化の効果はあまり期待できないと思われる。

- Karttunen の reversible unification アルゴリズム[Karttunen, 1986]

  このアルゴリズムは unification が成功したものについてのみ copy をつくる destructive unification である。graph を破壊する時、破壊前と破壊後の値の組を 1 つの array に次々と退避させていき、unification が成功すれば、ここで copy をつくり、その後（失敗した時も）array から original の graph を再構成するようにする。この方法では成功、失敗に関わらず destructive に unicfication が行なわれ、その結果 array への値の copy がノードの数だけ起こる。この時 array 構造になっているためにその長さに比例する情報追加時間がかかる。このため graph が大きくなると処理速度が大幅に遅くなるといえる。

- Wroblewski の non-destructive unification アルゴリズム[Wroblewski, 1987]

  このアルゴリズムは unification が進むにつれ incremental に copy をすることで over-copying を避けている。まず、単純な destructive unification の手続き unify1 を定義した上で、手続き unify2 は graph に copy がない場合にのみ copy をつくってこれに対し unify1 を適用する。unify2 では将来失敗するかも知れないノードについても copy をつくるので、ある程度 unification が進んでから失敗する場合の無駄が大きくなる恐れがある。

- Godden の lazy unification アルゴリズム[Godden, 1990]

  unification 中に起こるすべての copy の操作を遅らせる lazy evaluation を用いて成功したもののみ copy を作る点では Karttunen と同様であるが実際の処理では lazy evaluation されるデータ構造は処理系の内部で copy されているため、このコストは graph 自身の copy を作るコストと本質的にあまり差がない。

3 Tomabechiの時間差準破壊型単一化アルゴリズム

Tomabechiの時間差準破壊型単一化アルゴリズム[Tomabechi, ms]では、失敗したもののcopyを避けることと、originalの情報を効率良く保持することをtime-stampを用いることにより効果的に解決している。

3.1 アルゴリズムとデータ構造

まず、graphをunifyする時、ノードの値にtime-stamp（整数）を属性として付加しておき、1つのunificationが終るとこれに1加えることによってoriginalとresultとの情報を区別する。また、すべての再帰的なunificationが成功した時点ではじめてcopyを作る。このcopyingはgraphに対し2度目の再帰的処理をすることを意味するが、unificationの途中でのみ有効なforward（unificationの結果、他のノードと全く同じ構造を持つノードは、相手のノードとその構造を共有する）の情報が現在のtime-stampつきで各ノードのスロットに入れてあり、成功後のcopyingはこのスロットのみを参照しながら迅速に行なわれる。一般に、自然言語処理においては1回の解析の間に行なわれるunificationの大半は失敗することを考えれば[Tomabechi, ms]そのコストはover-copyingに比べて全体では少なくなるといえる。また、失敗した時にもやはりtime-stampが更新されこれらのスロットの値は次のunificationでは単に無視され、originalの情報のみがすぐに復活する。

この方法では、情報の退避と復活はすべてtime-stampのincrementで行なわれるため、graphの大きさに関わらずそのコストは常に一定で非常に小さい。さらに、再帰的に呼ばれたunificationがいずれか1つでも失敗すると次の再帰処理はもはや行なわず、すべてのunificationの実行を終らせる。従って各arcはそのsub-graphのunificationの結果を待つ必要がない。よってこの部分を自然に並列化することができこれによりさらに高速化することが期待できる。以下図1～3にデータ構造とアルゴリズムを示す（[Tomabechi, ms]より）

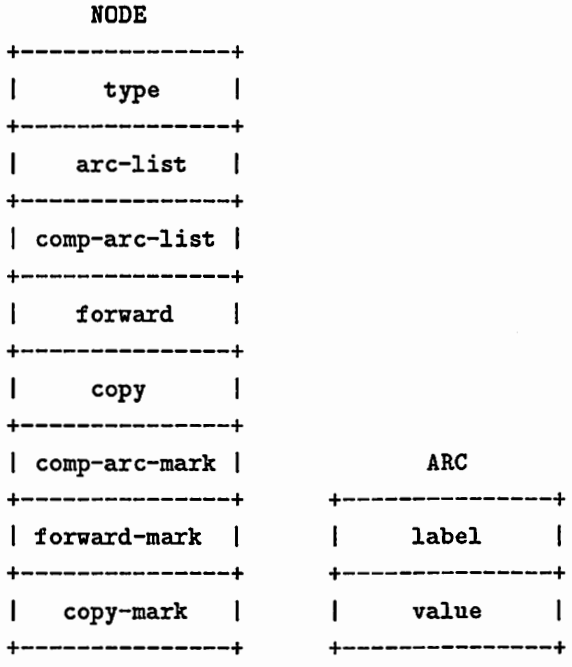


図1: ノードおよびアークの構造

```
  ;;;UNIFY-DAG is the top level entry. It calls UNIFY0 which calls UNIFY1.
  ;;;If UNIFY1 succeeds, a copy is returned to UNIFY-DAG. If UNIFY1 fails at
  ;;;any time of recursion, it immediately returns 'UNIFY-FAIL to UNIFY-DAG.
function UNIFY-DAG (dag1,dag2);
 RESULT := catch with tag 'UNIFY-FAIL calling UNIFY0(dag1,dag2)
 increment *unify-global-counter*  ;;; initially, this value starts from 10
 return RESULT;
end;

function UNIFY0 (dag1,dag2);
  if '*T* ==  UNIFY1(dag1,dag2);
      then COPY := COPY-DAG-WITH-COMP-ARCS(dag1);
           return COPY;
end;

function UNIFY1 (dag1-underef,dag2-underef);
 DAG1 := DEREFERENCE-DAG(dag1-underef);
 DAG2 := DEREFERENCE-DAG(dag2-underef);

 if (DAG1 == DAG2)          ;;; i.e., 'eq' relation, this may happen
    then return '*T*;       ;;; because of forwarding and loops

 else if (DAG1.type == :bottom) ;; i.e., variable
         then FORWARD-DAG(DAG1,DAG2,:temporary);
              return '*T*;

 else if (DAG2.type == :bottom)
         then FORWARD-DAG(DAG2,DAG1,:temporary);
              return '*T*;

 else if (DAG1.type == :atomic and DAG2.type == :atomic)
         then
             if (DAG1.arc-list == DAG2.arc-list)  ;;;contains atomic values
                then FORWARD-DAG(DAG2,DAG1,:temporary);
             return '*T*;
             else throw with keyword 'UNIFY-FAIL;
                 ;;; i.e., return directly to unify-dag (throw/catch construct)

 else if (DAG1.type == :atomic or DAG2.type == :atomic)
         then
             throw with keyword 'UNIFY-FAIL;

 else  NEW := COMPLEMENTARCS(DAG2,DAG1);
       SHARED := INTERSECTARCS(DAG1,DAG2);
       for each ARC in SHARED do
          RESULT := UNIFY1(destination of the shared arc for dag1,
                           destination of the shared arc for dag2);
          if (RESULT == '*T*)
             then
                return '*T*;
             else throw with keyword 'UNIFY-FAIL;

        If (the recursive calls to UNIFY1 successfully returned
            for all shared arcs) ;;; this check is actually unnecesary because if there is
           then                  ;;; a failure, unify1 has returned already (by throw).
              FORWARD-DAG(DAG2,DAG1,:temporary);
              DAG1.comp-arc-mark :=  *unify-global-counter*;
              DAG1.comp-arc-list := NEW
              return '*T*;
 end;
```

図 2: 時間差準破壊型単一化アルゴリズム（その 1）

```
function COPY-DAG-WITH-COMP-ARCS (dag-underef);
 DAG :== DEREFERENCE-DAG(dag-underef);
 if (DAG.copy is non-empty
       and
     DAG.copy-mark == *unify-global-counter*)
    then return the content of DAG.copy;   ;;; i.e. existing copy

 else if (DAG.type == :atomic)
         COPY :== CREATE-NODE();      ;;; creates an empty node.
         COPY.type :== :atomic;
         COPY.arc-list :== DAG.arc-list;    ;;; this is an atomic value
         DAG.copy :== COPY;
         DAG.copy-mark :== *unify-global-counter*;
         return COPY;

 else if (DAG.type == :bottom)
         COPY :== CREATE-NODE();
         COPY.type :== :bottom;
         DAG.copy :== COPY;
         DAG.copy-mark :== *unify-global-counter*;
         return COPY;

 else
        COPY :== CREATENODE();
        COPY.type :== :complex;
        for all ARC in DAG.arc-list do
              NEWARC :== COPY-ARC-AND-COMP-ARC(ARC);
              push NEWARC into COPY.arc-list;
        if (DAG.comp-arc-list is non-empty
                    and
            DAG.comp-arc-mark == *unify-global-counter*)
           then
              for all COMP-ARC in DAG.comp-arc-list do
                  NEWARC :== COPY-ARC-AND-COMP-ARC(COMP-ARC);
                  push NEWARC into COPY.arc-list;

        DAG.copy :== COPY
        DAG.copy-mark :== *unify-global-counter*;
        return COPY;
  end;

function COPY-ARC-AND-COMP-ARC (input-arc)
 LABEL :== label of input-arc;
 VALUE :== COPY-DAG-WITH-COMP-ARCS(value of input-arc);
 return a new arc with LABEL and VALUE;
 end;
```

図 3: 時間差準破壊型単一化アルゴリズム（その２）

sent#	Unifs	USrate	Elapsed time(sec)		Num of Copies		Num of Conses	
			T	W	T	W	T	W
1	6	0.5	1.066	1.113	85	107	1231	1451
2	101	0.35	1.897	2.899	1418	2285	15166	23836
3	24	0.33	1.206	1.290	129	220	1734	2644
4	71	0.41	3.349	4.102	1635	2151	17133	22943
5	305	0.39	12.151	17.309	5529	9092	57405	93035
6	59	0.38	1.254	1.601	608	997	6873	10763
7	6	0.38	1.016	1.030	85	107	1175	1395
8	81	0.39	3.499	4.452	1780	2406	18718	24978
9	480	0.38	18.402	34.653	9466	15756	96985	167211
10	555	0.39	26.933	47.224	11789	18822	119629	189997
11	109	0.40	4.592	5.433	2047	2913	21871	30531
12	428	0.38	13.728	24.350	7933	13363	81536	135808
13	559	0.38	15.480	42.357	9976	17741	102489	180169
14	52	0.38	1.977	2.410	745	941	8272	10292
15	77	0.39	3.574	4.688	1590	2137	16946	22416
16	77	0.39	3.658	4.431	1590	2137	16943	22413

図 4: 比較実験の結果

3.2 Tomabechi と Wroblewski のアルゴリズムとの比較実験

図 4 は、Earley の parser で日本語の文を解析するためにＨＰＳＧを用いたシステムにおいて、上の Tomabechi のアルゴリズムと Wroblewski のアルゴリズムを使ったものを比較したものを示している。[Tomabechi, ms]

Unifs 配置文の parsing 中に行なわれるトップレベル unification の回数（再帰的に呼ばれる unification の回数は含まれない。） US rate は unification の回数に対する、成功した unification の率である。 Elapsed time は Symbolics 3620 上での 3 回ずつの parsing 中それぞれ最短であったもので、T は Tomabechi のアルゴリズム、W は Wroblewski のアルゴリズムに loop と variable を扱う trivial な追加をしたもの[Kogure, 1989]を用いた時の値を表している。 Num of Copies 配置文中に生み出される copy node の総数であり、arc の数は含まれない。 Num of Conses は parse 中に cons された structure words の総数であり、両アルゴリズムの実質的なスペースコストを表している。

この実験結果にもみられるように Tomabechi のアルゴリズムは一文中の parsing 中の unification が約６０％が失敗に終るような一般的なグラマーを用いた場合、 Wroblewski の２～３倍の早さで走ることが確認されている。また unification の成功率が１００％である場合でも Wroblewski よりわずかに早くなると思われる。なぜなら、 Wroblewski が unify2 のなかで complement-arcs を 2 回用いているのに対し Tomabechi では 1 回しか用いていないためで、一般に arc の set-difference を調べるのはコストのかかる操作であるからである。

4 unification の並列処理

一般に効率の良い並列処理のためには並列プロセスに共有される global 変数の lock/unlock をできるだけ避けねばならないのは当然である。また、ある程度のプロセス間の synchronize も最小限度にする必要がある。この点から見ると、２で述べた従来の手法はいずれも並列化するのが困難なアルゴリズムであることがわかる。

なぜなら、Pereira,Karttunen & Kay および Karttunen のアルゴリズムではいずれも copy を避ける代わりにすべてのノードの情報を global 変数の中に分散させているため、並列化した場合、複数のプロセスがこれにアクセスしようとするたびに lock/unlock の over-head は膨大なものになると考えられる。また、Wroblewski のアルゴリズムでは、ノードの copy を作る時にそこから出発するすべての arc をその 1 つの copy ノードに copy するため、この部分は serial 処理にならざるを得ない。これがノードの数だけ incremental におこるのであるから、もはや並列処理とはいえなくなってしまうのみならず、一つのノードへの成功した各 sub-graph の copy を並列的に付加する時の lock/unlock のコストは極めて大きくなる。一般に、graph unification algorithm では、shared-arc への recursion の結果を待たなければ現在の unification の結果を得ることができない。例えば Wroblewski では、shared-arc への recursion がすべて成功した時のみ copy に現在の unification 中書き込みを行なうが、このためには現在の unification はすべての再帰的な unification の結果を待たなければならない。このように synchronization の問題も重要なボトルネックとなっている。

Tomabechi のアルゴリズムでは、copy への書き込みは graph 全体の unification が成功した後のみに行なわれるので synchronization の問題は本質的に存在しない。したがって、shared-arc における再帰的な呼びだしを並列に行なうことが可能である。しかしその場合新たに 2 つの問題点が考えられる。

1. ノードの forwarding は sub-graph の unification がすべて成功してはじめて行なわれなければならないが、下のサブプロセスとは並列に処理が進んでいるので、この結果を待つという synchronization を行なうと並列処理する意味が全くなくなってしまう。そこで、sub-graph の forwarding 操作を lazy evaluation を用いてすべての unification が終了するまで遅延（delay）させる。そして、unification が成功した場合にのみこれを行なう。これにより、unification の並列プロセスが走っている間はノードへの同時書き込みが一切なくなるので、lock/unlock の問題は起こらない。ただし、もし graph が loop を含んでいる場合、forward の情報を利用せずに unification を進めるので、loop の数だけ余計に unification のチェックを行なわなければならない。しかし並列プロセッサの数が充分大きい場合はこれらは並列的にチェックが行なわれるので特に問題とならない。

2. ある unification プロセスが局所的に unification に失敗した時は他の実行中のプロセスをすべて中止させ、トップのプロセスに失敗を知らせれば良いが、一方、各ローカルな並列プロセスには全体の graph の大きさがわからない以上 unification が成功したということはすべてのプロセスが正常に終ったことでしか判断不可能である。しかし、トップのプロセスが unification の結果をまって block している間に他のすべてのプロセスが終ってしまうとトップのプロセスは止まったまま動かなくなる。そこでこれを避けるために unification のプロセスとは別にすべての再帰的な unification のプロセスが終ったかどうかを常にチェックするプロセスを走らせ、成功した場合にはトップのプロセスを unblock して再び走らせ、遅延させておいた forward の操作を行ない、成功した結果の graph を返すという操作を行なう必要がある。

以上のような考察を踏まえ、Tomabechi のアルゴリズムを並列化した手続きを以下図 5、6 に示す。

```
function UNIFY-DAG (dag1,dag2);
 clear all locks';
 clear *unify-lwps*, *delayed-forward*, *unify-lwps-count*;
 with-spin-lock(*unify-result-lock*,
   *unify-result* :== 'NOT-DONE;)
 make-lwp(unify0,dag1,dag2);       ;;; unify0-lwp
 do forever
   with-spin-lock(*unify-result-lock*,
     unless *unify-result* == 'NOT-DONE
        return from loop;)
 increment *unify-global-counter*
 with-spin-lock(*unify-result-lock*,
   if (*unify-result* == 'UNIFY-FAIL)
      then return nil;
      else return *unify-result*;
end;

function UNIFY0 (dag1,dag2);
 with-spin-lock(*unify-lwps-count-lock*,
    increment *unify-lwps-count;)
 make-lwp(check-success,dag1,:lwpname checker-lwp);
 with-spin-lock(*unify-lwps-lock*,
  push make-lwp(unify1,dag1,dag2,:lwpname
                                *unify-global-counter*)
       into *unify-lwps*;
end;

function CHECK-SUCCESS (dag1);
 do forever
  with-spin-lock(*unify-lwps-count-lock*,
   if (*unify-lwps-count* == 0)
      then return from loop;)
 FORCE-DELAYED-FORWARD();
 with-spin-lock(*unify-result-lock*,
  *unify-result* :== COPY-DAG-WITH-COMP-ARCS(dag1));
 return *unify-result*;
end;

function SENDFAIL();
 if (*unify-global-counter* == lwpname of the process
                               evaluation this function)
    then kill all lwps in *unify-lwps* except this lwp;
         kill checker-lwp;
         with-spin-lock(*unify-result-lock*,
            *unify-result* :== 'UNIFY-FAIL);
end;

function UNIFY1 (dag1-underef,dag2-underef);
 DAG1 :== DEREFERENCE-DAG(dag1-underef);
 DAG2 :== DEREFERENCE-DAG(dag2-underef);
 LWPNAME :== lwpname of the process evaluating this function;

 if (DAG1 == DAG2)
    then decrement *unify-lwps-count*;
         return '*T*;

 else if (DAG1.type == :bottom) ;; i.e., variable
         then FORWARD-DAG(DAG1,DAG2,LWPNAME,:temporary);
              decrement *unify-lwps-count*;
              return '*T*;

 else if (DAG2.type == :bottom)
         then FORWARD-DAG(DAG2,DAG1,LWPNAME,:temporary);
              decrement *unify-lwps-count*;
              return '*T*;

 else if (DAG1.type == :atomic and DAG2.type == :atomic)
         then
             if (DAG1.arc-list == DAG2.arc-list)  ;;;contains atomic values
                then FORWARD-DAG(DAG2,DAG1,LWPNAME,:temporary);
                     decrement *unify-lwps-count*;
                     return '*T*;
             else SEND-FAIL();
```

図 5: 並列化した時間差準破壊型単一化アルゴリズム（その１）

```
else if (DAG1.type == :atomic or DAG2.type == :atomic)
        then
            SEND-FAIL();

else  NEW :== COMPLEMENTARCS(DAG2,DAG1);
      SHARED :== INTERSECTARCS(DAG1,DAG2);
      increment *unify-lwps-count* by length of SHARED;
      for each ARC in SHARED do
         send to mail box
           (destination of the shared arc for dag1,
            destination of the shared arc for dag2);
         push make-lwp(UNIFY1,dag1 received from mailbox,
                              dag2 received from mailbox,
                        :lwpname LWPNAME)
              into *unify-lwps*;
       delay evaluation of (FORWARD-DAG(DAG2,DAG1,LWPNAME:temporary);
       DAG1.comp-arc-mark :== *unify-global-counter*;
       DAG1.comp-arc-list :== NEW
       decrement *unify-lwps-count*
end;

function FORWARD-DAG (dag1,dag2,lwpname,type);
 if (lwpname == *unify-global-counter*)
    then if (type == :temporary)
            then dag1.forward :== dag2;
                 dag1.forward-mark :== *unify-global-counter*;
         if (type == :permanent)
            then dag1.forward :== dag2;
                 dag1.forward-mark :== 9;
end;

function COPY-DAG-WITH-COMP-ARCS (dag-underef);
 DAG :== DEREFERENCE-DAG(dag-underef);
 if (DAG.copy is non-empty
       and
     DAG.copy-mark == *unify-global-counter*)
    then return the content of DAG.copy;   ;;; i.e. existing copy

 else if (DAG.type == :atomic)
         COPY :== CREATE-NODE();     ;;; creates an empty node.
         COPY.type :== :atomic;
         COPY.arc-list :== DAG.arc-list;    ;;; this is an atomic value
         DAG.copy :== COPY;
         DAG.copy-mark :== *unify-global-counter*;
         return COPY;

 else if (DAG.type == :bottom)
         COPY :== CREATE-NODE();
         COPY.type :== :bottom;
         DAG.copy :== COPY;
         DAG.copy-mark :== *unify-global-counter*;
         return COPY;

 else
        COPY :== CREATENODE();
        COPY.type :== :complex;
        for all ARC in DAG.arc-list do
              NEWARC :== COPY-ARC-AND-COMP-ARC(ARC);
              push NEWARC into COPY.arc-list;
        if (DAG.comp-arc-list is non-empty
                  and
            DAG.comp-arc-mark == *unify-global-counter*)
           then
              for all COMP-ARC in DAG.comp-arc-list do
                  NEWARC :== COPY-ARC-AND-COMP-ARC(COMP-ARC);
                  push NEWARC into COPY.arc-list;

        DAG.copy :== COPY
        DAG.copy-mark :== *unify-global-counter*;
        return COPY;
  end;

function COPY-ARC-AND-COMP-ARC (input-arc)
 LABEL :== label of input-arc;
 VALUE :== COPY-DAG-WITH-COMP-ARCS(value of input-arc);
 return a new arc with LABEL and VALUE;
 end;
```

図 6: 並列化した時間差準破壊型単一化アルゴリズム（その２）

5 実験と考察

5.1 並列処理実験について

上に述べた並列アルゴリズムの Implementation を並列マシンＳｅｑｕｅｎｔ（shared memory multiprocesser）上の並列 Common Lisp を用いて行なっており、３．２での serial 処理実験との比較を実験中である。現在、並列アルゴリズムは unification の結果には問題がないが、並列 Common Lisp のスケジューラーに問題があり、現在 Common Lisp コンパイラーのバグ取り中であり、この終了を待って複数プロセッサー利用時のデータを集める予定である。また、実際の並列化では４で述べたほかに次のような問題を考慮しなければならない。

1. 並列プロセスの世代重複

   一つの unification が失敗した時には現在存在しているプロセスのリストを参照して他のすべてのプロセスを強制的に終らせなければならない。しかしプロセスが実際に止まるまでにはスケジューラを介するために常にタイムラグがあるので、殺されるプロセスが止まる直前に新しい子プロセスをすでに生んでいる場合、これを殺すことはできないという問題がある。この場合、トップのプロセスは失敗を知るとすぐに、世代を更新させ新しいトップのプロセスが unification をはじめるがこの間にも完全に止まらなかった前の世代のプロセスの残りが新たに孫のプロセスを生みだし、新しい世代の unification の最中にこれらが fail や forward などの side effect を起こす可能性がある。この様子を図７に示す。

   これを防ぐためには、side effect のある処理をプロセスが行なう前にプロセスがどの世代のトッププロセスに属するものかをチェックしなければならない。

2. 子プロセスへのデータ受渡し

   再帰的に処理が進む場合でも、子プロセスを作る時は関数内の動的な変数の束縛などのローカルなスコープはすべて無効になる。つまり、引数を渡す場合でもプロセス間で共有できるデータ構造（global 変数、mailbox など）を用いなければならない。

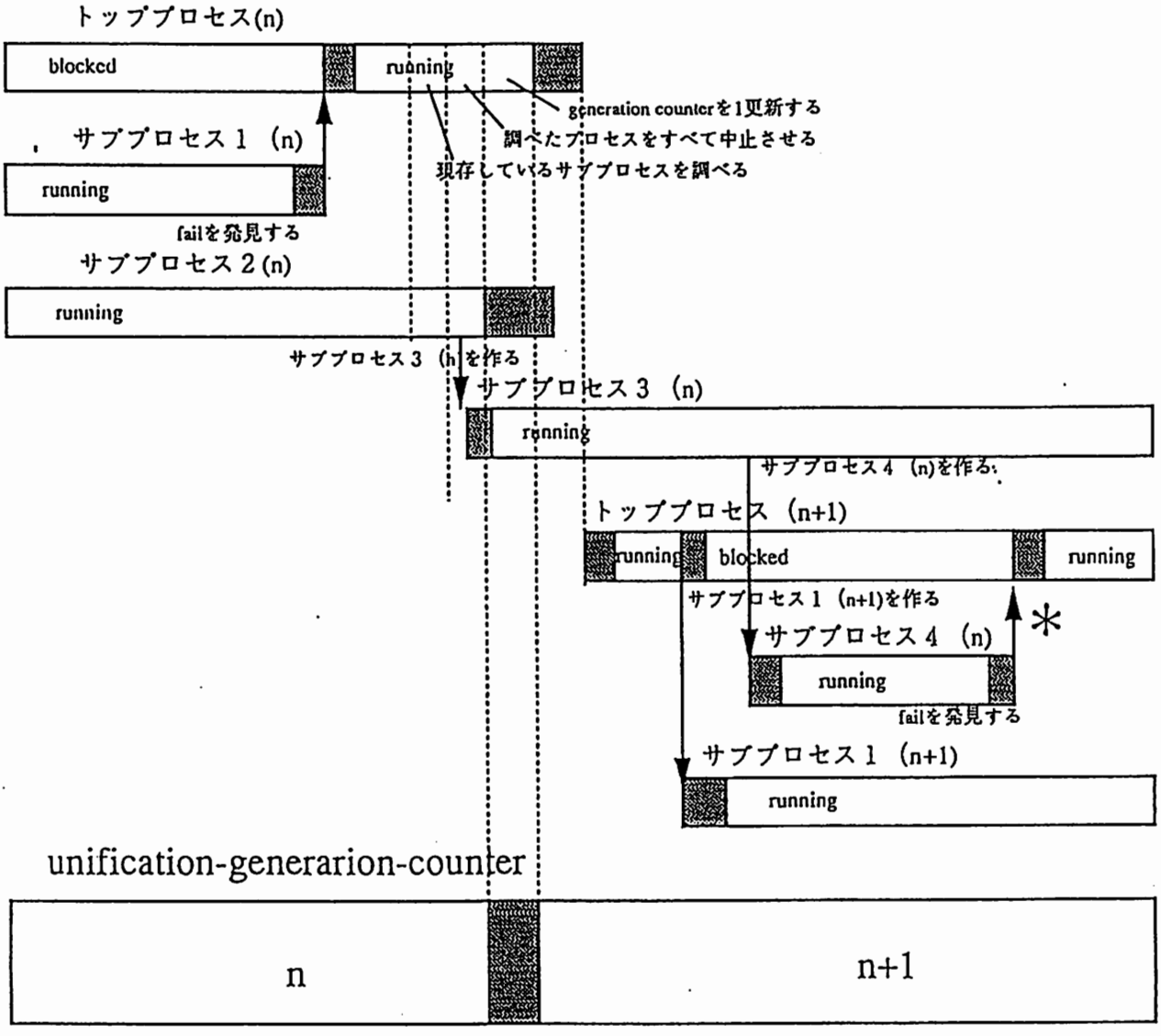
3. 共有データの lock

   複数のプロセスが同じデータを使う時、read/read の衝突についてはマシンのアーキテクチャ上一般には問題にならないとされている。しかし、read/write,write/write の場合では、ハードウェアからコンパイラまでのいずれかのレベルで排他制御を行なうことが避けられないため、実際のアルゴリズムにはすべての共有データに対して同時のアクセスが起こり得ないことが極めて重要である。

4. 局所的な forward

   unification の最中では sub-graph が存在している場合は前述のように forward を遅延させるが、局所的に成功した場合（同一のノード同士の unification, 或はどちらか一方または両方が変数の時）は遅延させずにその場で forward しなくてはならない。もしこの場合も遅延させるとそれによって unification の結果が誤ってしまうことが起こりうる。この例を図８に示す。

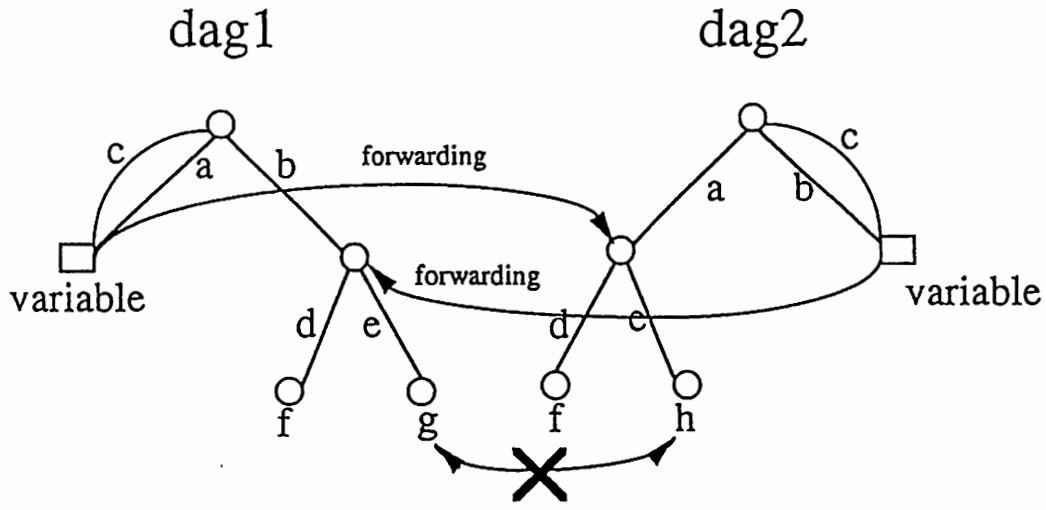
<プロセス (n)はunification-generation n に属するプロセスであることをあらわす。
また ▒▒ の部分は処理が実行されるまでのタイムラグをあらわす。>



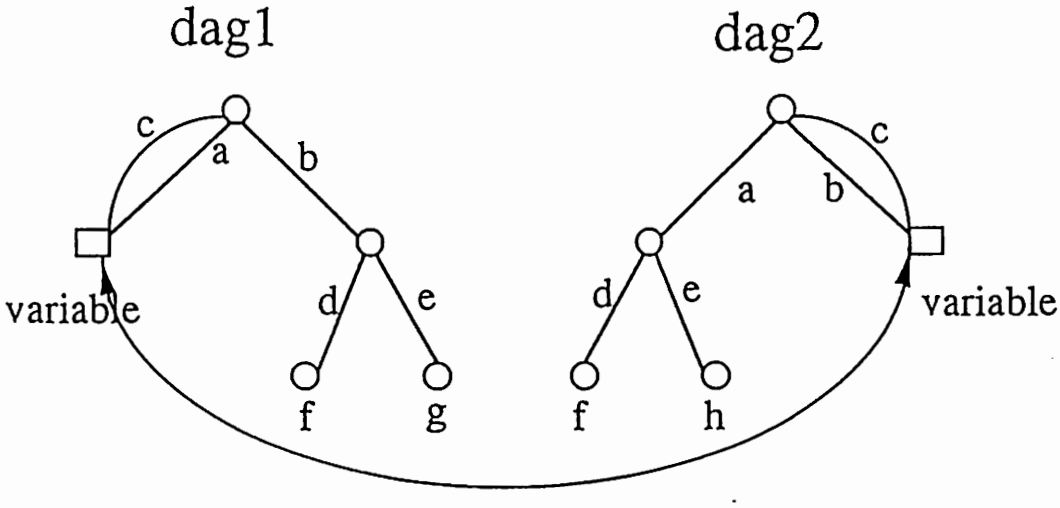
＊ サブプロセス１ (n)はunificationのfailを発見するとトッププロセス (n)にそれを知らせトッププロセス (n)は現存するサブプロセスm(n)をすべて調べ,中止させる。しかしサブプロセス2 (n)は中止させられる直前にサブプロセス３ (n)をつくり、これがgeneration-countが更新されてからサブプロセス４ (n)をつくる。よってサブプロセス４ (n)がunificationのfailを発見するとトッププロセス (n+1)にfailを知らせてしまう。

図 7: 並列プロセスの世代重複

この場合arc a,arc bに対するunification は,片方がvariableなので局所的に成功する。



図(a)arc a,bに対するunificationが成功したあと局所的にforwardするとarc cについてunificationを行なった結果,全体で正しい結果(fail)が得られる.



図(b) arc a,bに対するforwardを遅延させるとarc cに対するunificationは両方variableなので成功してしまう.

図 8: 局所的 forward

5.2 今後の実験課題

まず、一般的なgrammarにおいて並列プロセッサの数（１～６４？）と処理速度の関係を実験することが必要である。さらに、並列化することによるスケジューリングと並列プロセスの同期化のover-headに対し、どのようなgraphの大きさや形によって並列化の利点が得られるかを調べ、データを取って並列化に適するgraph即ちgrammarの特性を得たい。これがある程度特定できれば、grammarの書き方やgrammarの定義そのものをserial/parallelのどちらかに最適化するアルゴリズムが考えられると思われる。また、Kasparなどのdisjunctionのアルゴリズムを利用したgraph unificationの並列化についても考察する。

6 結論

Tomabechiの時間差準破壊型単一化アルゴリズムが並列化に適しているのはunification中のgraphのdestructiveな変更をlazy evaluationを使うことによって完全になくすことが可能であるためlock/unlockによる問題を最小限にできるからである。また、各unificationはshared-arcへの再帰的な呼び出しの結果を待たずに処理を進められるためunification中のsynchronizationの問題が発生しない。これらは、時間差準破壊型アルゴリズムの特徴であり、Wroblewskiなどのアルゴリズムでは本質的に実現不可能である。このように、Tomabechiのアルゴリズムを並列化した場合は各並列プロセスがsychronizationやノードへの書き込みのコスト無しに、子プロセスを並列的に生みながらshared-arcを下っていけるので、unificationの失敗を極めて早く発見することが可能である。このように並列化を行なうことによりTomabechiのアルゴリズムの特徴であるunificationの失敗の早期発見がより効果的に行なわれる。 以上のように、本稿で述べられたいくつかの手法を用いることによりTomabechiのアルゴリズムを効果的に並列化できることがわかった。

今後、並列自然言語処理を進めるに当たり、Earley,Active Chart,LR等のparserとgraph unificationとを統合的に並列化することも考えられる。ただし、これらのparserはtableやchartをグローバルに利用しているため、これまで考察した種々の問題により顕著な並列効果は期待できない。また今回はunification algorithmのみの並列化を行なったが、Unification-based Grammarのように言語理論そのものが元々serial的な発想に基づいている場合、これをアルゴリズムのレベルのみで並列化しても並列化不可能な部分も多く、自ずと限界があるのではないかと思われる。したがって、並列自然言語処理という観点からすれば言語理論的な制約自体が並列的であるような理論に基づいたアルゴリズムを利用した並列自然言語処理も検討すべきであろう。

参考文献

[Bresnan, 1982] Bresnan, J. *The Mental Representation of Grammatical Relations*. MIT, 1982.

[Godden, 1990] Godden, K. "Lazy Unification" In *Proceedings of ACL-90*. 1990.

[Karttunen, 1986] Karttunen, L. *Development Environment for Unification-Based Grammars*. Report CSLI-86-61. Center for the Study of Language and Information, 1986.

[Karttunen and Kay, 1985] Karttunen, L. and Kay, M. "Structure Sharing with Binary Trees". In *Proceedings of ACL-85*. 1985 .

[Kasper, 1987] Kasper, R. "A Unification Method for Disjunctive Feature Descriptions". In *Proceedings of ACL-87*. 1987.

[Kogure, 1989] Kogure, K. *A Study on Feature Structures and Unification.* ATR Technical Report. TR-1-0032. 1988.

[Pereira, 1985] Pereira, F. "A Structure-Sharing Representation for Unification-Based Grammar Formalisms". In *Proceedings of ACL-85.* 1985.

[Pollard and Sag, 1987] Pollard, C. and Sag, A. *Information-based Syntax and Semantics.* Vol 1, CSLI, 1987.

[Tomabechi, ms] *Quasi-Destructive Graph Unification.* Submitted to The Second International Workshop on Parsing Technologies. Manuscript.

[Yoshimoto and Kogure, 1989] Yoshimoto, K. and Kogure, K. *Japanese Sentence Analysis by means of Phrase Structure Grammar.* ATR Technical Report. TR-1-0049. 1989.

[Wroblewski, 1987] Wroblewski, D."Nondestructive Graph Unification" In *Proceedings of AAAI87.* 1987.