記 号 処 理 44－3
（１９８８．１．２２）

並列Lisp上でのＧＨＣの実現

寺田　実，岩崎英哉，田中哲朗，
小西弘一，白川健治，和田英一

東京大学工学部計数工学科

並列論理型言語ＧＨＣの処理系は，従来はガードに制限のあるフラットＧＨＣの処理系がほとんどであり，またプロセス間で論理変数を共有する方式のものが多かった．

本研究では，並列処理記述可能なLisp言語mUtilisp上にＧＨＣ処理系を構成する．mUtilispはプロセス間でのオブジェクトの共有を排除し，メッセージによるプロセス間通信だけが可能である．

このような方式によるＧＨＣの実現は，ゴールのスケジュールなどは容易になる反面，各プロセスに分散した論理変数の扱いなどが問題になってくる．

実際の処理系は未完成であるが，本発表では，実現の基本方針，制御アルゴリズム，論理変数の扱いなどの問題点のいくつかについて報告する．

"Implementation of GHC on the concurrent lisp" (in Japanese)

by Minoru TERADA, Hideya IWASAKI et al.

Department of Mathematical Engineering and Information Physics
Faculty of Engineering, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, Japan

In this paper we describe the outline of our implementation of GHC on the concurrent lisp language "mUtilisp". Each goal is interpreted by an mUtilisp process, which is communicating by messages, not by shared objects. Most implementation of GHC limits the language by not allowing user-defined goals in the guard part. (This restricted version is known as Flat GHC or FGHC.) In our implementation we tried not to restrict the language as far as possible.

1．はじめに

筆者らは，Utilispに並行動作記述を可能にしたLisp処理系mUtilisp[1]を作成してきた．本研究では，そのmUtilispによって，並列論理型言語ＧＨＣ[2]のインタプリタの記述を試みる．ＧＨＣ処理系もすでにいろいろ発表されているが，その多くはガード部にユーザ定義述語を許さないフラットＧＨＣ（ＦＧＨＣ）を対象としており，また各ゴールを実行するプロセスが変数を共有することを前提としている．それに対し，今回われわれが作成しようとしている処理系の方針は以下の通りである：

＋　共有メモリを仮定せず，メッセージによって通信するプロセスの集合体として実現する．

＋　ＧＨＣの言語仕様に対し，できるだけ制限をしない．特に，ガード部にも一般の述語を用いている，いわゆるフラットでないプログラムも処理可能にする．

＋　記述言語mUtilispの並列性を生かす．ａｎｄやｏｒでつながれた各ゴールをそれぞれmUtilispのプロセスとして実現し，スケジューリングもmUtilispに任せる．

本処理系は並行処理言語で記述するため，プロセスの制御・スケジュールはそれほど問題にならず，むしろ変数の保持の方法，ユニフィケーションによるプロセスのサスペンドの実現法などが問題となる．

2．記述言語mUtilisp

本実現は，Lispの一方言であるmUtilispで記述する．mUtilispは，Utilispに並列動作を導入したもので，以下のような特徴をもっている：

1．　動的なプロセス生成

2．　プロセス間のLispオブジェクト共有の排除

3．　プロセスの親子関係によるシンボルの諸属性の継承

4．　メッセージによるプロセス間通信

特徴２は，プロセスごとに異なるシンボル表を用意することにより，プロセス間での（変数としての）シンボルの衝突を避けようとするものである．その結果，同じ印字名をもつシンボルでもプロセスによってその値が異なることになり，共有シンボルによる干渉（そして通信）が排除される．

特徴４のプロセス間通信について説明する．システム関数sendは，(send aProcess aMessage) という形式で非同期に（つまり相手がreceiveしていなくても封鎖されずに）メッセージを送る．相手の指定方法は，Lispオブジェクトとしてのプロセスを直接指定するか，各プロセスに一意につけられたプロセス名（ストリング）を指定するかのどちらかである．

一方システム関数receiveは，(receive aProcess) あるいは (receive) の形式で呼びだす．前者は相手を特定した場合，後者は不特定の相手からのメッセージを待つ場合である．受信すべきメッセージがないときにはそれが到着するまでreceiveを実行したプロセスは封鎖される．

メッセージとしては任意のLispオブジェクトを用いることが可能であるが，共有排除の方針にしたがい，それらは翻訳という過程をへてから伝達される．翻訳とは本質的にはコピーの操作で，コンスセルの場合はコピーが，シンボルの場合は相手プロセスでの同一印字名をもつシンボルへの変換がおこる．整数はポインタコーディングされているため，そのまま伝達される．

プロセス間通信についてまとめておく：

＋　通信は１対１あるいはＮ対１で，ブロードキャストはない．

＋　翻訳操作によって，メッセージはそのコピーが伝達される．共有はできない．

＋　プロセス間に遠近の差はなく，通信のコストは一定である．

＋　メッセージの大きさによるコストの違いは，リストの場合その長さによる．すなわち印字表現の長さとおおむね対応する．

mUtilispを計算機アーキテクチャに対応させていえば，共有メモリをもたないマルチプロセッサに対応すると考えられる．メッセージは印字表現としてバスを経由して伝達されると考えるのが適当である．

3．実現上の制限

節は，ヘッド，ガード部，ボディー部からなるが，本来これら同士も並列に実行されてよいものである．（もちろんコミット以前は，ボディー部といえども上位の変数を具体化できないが．）しかし本実現では，これら三部分を順に実行するという制限を設けた．ヘッドのユニフィケーションが完了してからガード部の実行を開始し，それらがすべて成功してコミットしたのちはじめてボディー部に着手する．こうした制限を設けた理由は，未コミット状態のままボディー部の実行を開始してしまうと，コミットの事実を多くのプロセス（ボディー部のプロセスとその子孫たち）にブロードキャストする必要が生じるからである．

また，現在のところ，対象とするデータは，定数としては整数，構造をもったものはリストに限っている．定数に文字列や記号アトム（シンボル）を追加することや，構造として複合項を追加することはそう困難なことではない．

4．制御方式

本実現では，並列言語mUtilispの記述能力を生かし，極力プロセスに分割して実行する方式をとる．具体的には，あるゴールが与えられたとき，それに対して複数存在する節のそれぞれをｏｒとして並列に実行する．またその節に含まれるゴールのそれぞれをａｎｄとして並列に実行する．図１に，簡単なプログラムの実行例を図式的に示す．

以下にｏｒ・ａｎｄのアルゴリズムを示す．

ｏｒ
引数：ゴール
1．　ゴールに対応する節集合を選ぶ．空集合なら親に失敗を報告．終了．
2．　節集合のそれぞれに対して，それを実行するａｎｄを子プロセスとしてｆｏｒｋする
3．　子からのコミット報告を待つ
4ａ．　成功の報告があれば，その子にコミット許可を与える．他の子には終了を命じる．
4ｂ．　子全部から失敗の報告があれば，親に失敗を報告．終了．
5．　コミットした子の完了を待ち，それを自分の結果として，親に報告．終了．

ａｎｄ
引数：ゴール，節ひとつ
1．　ヘッドのユニファイ．
2．　ガード部を実行するｏｒを子プロセスとしてｆｏｒｋする．
3．　子の完了を待つ．
4ａ．　子がひとりでも失敗すれば，親に失敗を報告，終了．
4ｂ．　子全部から成功の報告があれば，親にコミット報告を行う．
5．　親からのコミット許可を待つ．
6ａ．　コミット許可が得られなければ，終了．

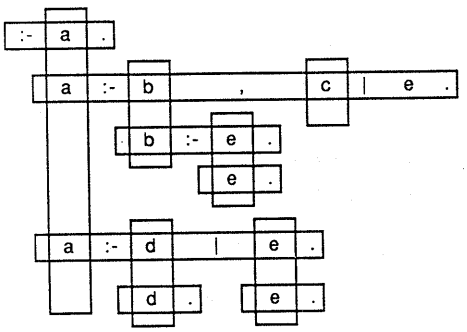

図１．実行例

```
a：－b，ｃ｜ｅ．
a：－ｄ｜ｅ．
ｂ：－｜ｅ．
ｄ．
ｅ．
```

というプログラムのもとで，

```
：－ａ．
```

なる入力を実行した例．横長の長方形がａｎｄを，縦長の長方形がｏｒを表わしている．

6 b．　コミット許可が得られれば，ボディー部を実行するｏｒを子プロセスとしてｆｏｒｋする．
7．　子の終了を待つ．
8 a．　子がひとりでも失敗すれば，他の子を終了させ，親に失敗を報告，終了．
8 b．　子全部から成功の報告があれば，親に成功を報告．終了．

実際には，プロセスの仕事にはこのような制御関係だけでなく，以下に述べるような変数の値とユニフィケーションに関するものが含まれる．

5．変数とユニフィケーション

5．1．変数の値

本実現においては，変数はそれが出現した節を実行しているａｎｄプロセスに属しており，変数の値は，そのプロセスが管理する．またリストを構成するコンスセルについても，所属するプロセスがきまっている．

変数の値には，以下の３通りがありうる（図２）：

1．未束縛
2．他の変数とユニファイしている
　2 a．自プロセスの変数
　2 b．他プロセスの変数
　2 c．他プロセスに属するコンスセルのｃａｒセル・ｃｄｒセル
3．値がすでにきまっている
　3 a．定数
　3 b．自プロセスに属するコンス
　3 c．他プロセスに属するコンス

5．2．他プロセスの変数の参照

自プロセスの変数が値として自分以外のプロセスの変数を指す場合には，やや困難がともなう．mUtilispのプロセスの特徴である空間の分離によって，直接ポインタで指すことができないからである．従って，以下のようなレコードをもちいて他プロセスの変数を表現する：

（＜属するプロセス名＞　＜変数名＞　＜コミット値＞）

他プロセスのコンスも同様の方法で表現する：

（＜属するプロセス名＞　＜コンス番号＞　＜コミット値＞）

変数の参照の方向については，ユニファイ時に適当な方法を用いることで，自分の先祖にあたるプロセスの変数に参照を限る．つまりユニファイによる変数のチェーンを，子から親への一方向とする．そうせずに親から子への変数チェーンを許すと，子プロセスが完了したのちも変数チェーンのためだけに子プロセスが存在しつづけなければならず，空間効率を悪化させるおそれがある．

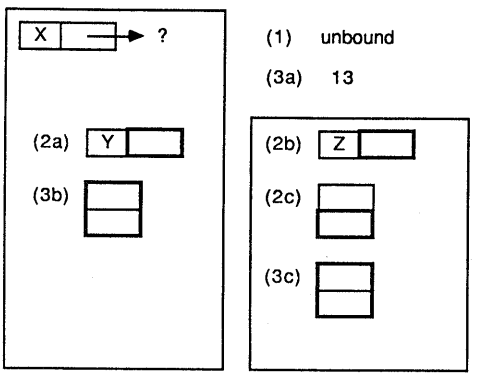

図２．変数の値
外側の枠はプロセスを表わしている．
左の枠は変数Ｘが属しているプロセス，右の枠はそれ以外のプロセスである．枠の外にあるものはプロセスに属さない定数などである．

5．3．値の伝播

変数チェーンに属するある変数が，定数（あるいは構造）とユニファイすることによって値が定まった場合，その値はその変数チェーンに属する他の変数にも伝播する必要がある．そのための方法として，以下のようなものが考えられる．

・全プロセスに対するブロードキャスト

・変数チェーンの各変数にメッセージによって値を伝播する．

・他の変数が値を必要とした時点で値を送る．

本実現では，上の第三の方法をとることにした．第一の方法はブロードキャストが必要となる点で問題がある．第二の方法は，値そのものを必要としないプロセスにまで伝播することになって無駄が生じると考えられるためである．（例えば，ヘッドで引数として受取った変数をそのままボディー部のゴールに使用するだけであれば，その節ではその変数の値を必要としていないことになる．）

5．4．上位変数の具体化によるサスペンド

ＧＨＣにおいては，コミットしていない節が自分より上位の変数を具体化しようとした場合，その上位の変数の値が定まるまでサスペンドさせられる．

```
[ｐ１]　ａ（Ｘ）：－ｔｒｕｅ｜ｂ（Ｙ），．．．．
[ｐ２]　ｂ（Ｚ）：－Ｚ＝２｜．．．
```

ａを実行するプロセスをｐ１，ｂを実行するプロセスをｐ２とすると，ｐ２の変数Ｚはヘッドのユニフィケーションによって，ｐ１のＹを値として持つ．Ｚ＝２のユニフィケーションがそのサスペンドの例で，ｐ１（かｐ１のｐ２以外の子）によってＹが束縛されるまでｐ２は計算を中断する．

このようなサスペンドを実現するため，以下のような手順を用いる：

1．　ｐ２はｐ１に対してＹの値を要求し，返事待ちにはいる．

2．　ｐ１はその問いあわせに対し，回答を保留する．（変数Ｙにキューしておく．）

3．　のちにＹの値が定まったときＹのキューをみて，返事を待っているプロセス（ｐ２）にその値を送る．（もしＹがもっと上位のプロセスの変数とユニファイし，結果としてＹからさらに上位へのチェーンができた場合は，キューも上位へ委譲する必要がある．）

ＦＧＨＣにおいては，ｐ１が回答を保留する条件は

1）Ｙが未束縛である；

2）ｐ２がまだコミットしていない；

のふたつで，比較的単純であった．

しかし，フラットでないＧＨＣでは，もうすこし事態が複雑である．

```
[ｐ１]　ａ（）：－｜ｂ（Ｘ），．．．．
[ｐ２]　ｂ（Ｙ）：－ｃ（Ｙ）｜．．．．
[ｐ３]　ｃ（Ｚ）：－｜Ｚ＝２，．．．
```

ｐ３はすでにコミットしているにもかかわらずＺ＝２のユニファイはサスペンドされなければならない．ｐ２がコミットしていないためである．つまり，サスペンドの判定に必要な情報が，ｐ１－ｐ２－ｐ３というプロセスの親子関係の中に散在しており，要求を出したプロセス（ｐ３）と，管理者であるプロセス（ｐ１）だけでは判定がくだせないのである．

5．5．コミット値

これを解決するために，他プロセスの変数に対し，コミット値という整数値をつけることにした．

変数は新しく生成されたときコミット値０を持つ．すなわち自プロセスの変数はすべてコミット値は０である．変数を子プロセスに渡す際，コミット値は１増やされる．その子プロセスがコミットすると，コミット値は１減る．

つまりコミット値とは，変数の属するプロセスから自プロセスまでのプロセス列の中のコミットしていないプロセスの個数である．あるプロセスにとって，ユニファイによって値を変えてよい変数は，自プロセスの変数か，コミット値０のものに限られることになる．

上例でいえば，ｐ３のＺの値となっているｐ１のＸのコミット値は，ｐ１からｐ２に渡った段階で１になり，ｐ２からｐ３に渡った段階で２になるが，ｐ３がコミットしたために１になる．

実現上の制限の項で，コミットに成功するまではボディー部の実行を開始しないと述べたが，そのことによってこのコミット値が意味をもってくる．プロセスの親子関係の途中にあるプロセスの状態が変化しないので，親からもらったコミット値がずっと有効なのである．

なお，コミット値は他プロセスのコンスに対しても全く同様に定義する．

５．６．コミット値の１ビット化

前項のコミット値は，その定義のままでは自然数の値を取る．しかし，コミット値は０か１以上かの区別だけが重要であるので，これを０であるかどうかの１ビットで表現可能である．いいかえると，変数の属するプロセスから，自プロセスまでのプロセス列の中にひとつでもコミットしていないプロセスがあるかどうかを示すフラグが各変数についていればよいのである．

ローカル変数はこのフラグを偽としてうまれる．ボディー部で子プロセスに上位変数をわたす場合はこのフラグをそのままわたし，ガード部でわたす場合のみ，このフラグを真にしてわたせばよいのである．

この方式で，ユニフィケーションによってサスペンドの可能性があるのは，

１．未コミットのプロセスが上位変数をユニファイする場合；

２．コミットしたプロセスが，フラグが真の上位変数をユニファイする場合；

のふたつになる．

５．７．構造（リスト）のユニフィケーション

変数のチェーンにおける参照の方向をさだめたのと同じ問題がリストのコンスセルについても生じる．

```
[ｐ１]　ａ（）：－｜ｂ（Ｘ），．．
[ｐ２]　ｂ（Ｙ）：－｜Ｙ＝［Ｙ１｜Ｙ２］，．．
```

この例において，Ｘの値となるコンスセルをどちらのプロセスに所属させるかが問題となる．ｐ２に属させた場合，Ｘから下むきに（子にむかって）ポインタが生じ，ｐ２が完了しても領域を完全に解放することができない．そこで本実現においては，このコンスセルをｐ１に所属させるようにした．つまり，Ｙの値はｐ１のコンス１番，Ｙ１の値はｐ１のコンス１番のｃａｒとするのである．（図３）

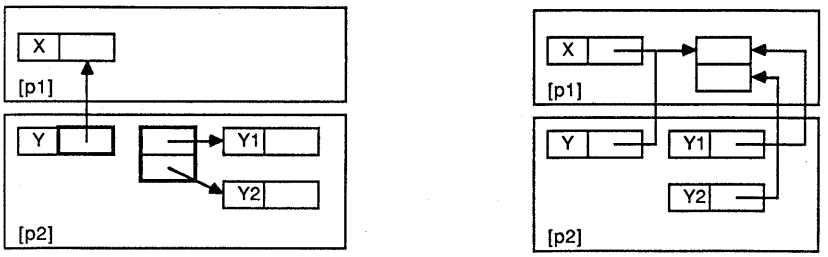

図３．リストとのユニフィケーション
左はユニフィケーション前で，太枠同士をユニファイしようとしている．
右はユニフィケーション後で，コンスセルはｐ１の側に移った．すべてのチェーンは上位プロセスに向いている．

5．8．上位変数同士のユニフィケーション

上位変数同士を，コミットしていない下位プロセスでユニファイする場合，上位へのメッセージによって値を要求するという方式ではうまくいかないことがある．

```
[p1]    a () :- | b (X, X), . .
[p2]    b (U, V) :- U=V | . .
```

上例において，ｐ２がＵとＶのユニファイを試み，まず現在のＵの値であるｐ１のＸの値を要求する．ｐ１はｐ２が未コミット状態であるから，Ｘが具体化するまで回答を保留し，結果としてｐ２はサスペンドしてしまうが，それは誤りである．

この例だけについてこの問題を回避するのであれば，ｐ２はｐ１に問いあわせる前にＵ，Ｖの値（どちらもｐ１のＸである）を比較し，無条件にユニファイ成功とすればよい．しかし，変数チェーンのもっと上の方で同一変数となっている場合などはうまくいかない．

```
[p1]    a (X) :- | b (X, Y), . . , X=Y, .
[p2]    b (U, V) :- U=V | . .
```

ｐ１がＸ＝Ｙのユニファイをするまで，ｐ２はサスペンドしている必要がある．つまりＸかＹのいずれかのキューにはいっている必要がある．ここで，変数チェーンに方向を定めたことが役にたつ．Ｘはｐ１よりさらに上位のプロセスにつながっているから，もしＸとＹがユニファイすることがあれば，そのチェーンの方向はＹからＸへとなる．従ってこの例では下位のプロセスに属するＹにキューしておくのがよいことになる．

ところが，ｐ２からではＵ，Ｖどちらのチェーンがより上位までつながっているのか判断できないことがある．それを判断するためには，メッセージがチェーンを上にたどっていくことが必要である．

6．おわりに

ＧＨＣをこのようにメッセージ通信によって実現した場合，性能向上のために何が必要になるかはわからないことが多い．たとえば変数チェーンをたどるタイミングをとっても，子プロセスにわたす段階で最終的な値（すなわち最上位の変数の値）までたどってしまうべきか，あるいは子プロセスがほとんどすぐ死ぬことをみこしてチェーンのままでわたすべきかは難しい問題である．本処理系は，当面このような問題を解析するためのひとつの実験台としようと考えている．

本稿を書いている１９８７年１２月の段階では，まだアルゴリズムの検討を行なっており，実際にプログラムを書くに至っていない．mUtilisp処理系はすでに安定して動いているので，今後は実際の処理系の作成をとおして，本方式の検証を行なっていく予定である．

参考文献

[1] 岩崎：Lispにおける並列動作の記述と実現，情報処理学会論文誌，
Vol.28, No.5, pp.465-470(1987).

[2] K. Ueda: Guarded Horn Clauses, In Logic Programming '85, E. Wada(ed.),
Lecture Notes in Computer Science 221, Springer-verlag(1986).