ICOT Technical Report: TR-301

TR-301

GHCによる簡単な プログラミングシステムの記述

太田祐紀子(富士通SSL)
田中二郎

September, 1987

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

GHC による簡単なプログラミングシステムの記述

田中　二郎　　太田祐紀子
（ICOT）　　（富士通SSL）

本報告では、並列論理型言語GHC の簡単なプログラミングシステムについて考察を行う。プログラミングシステムとは、その上でユーザがプログラムを入力したり実行したりできる環境の事であり、オペレーティング・システムのユーザ・インタフェースにあたるものである。本報告では、まずプログラミングシステムの基本機能を提供するメタコールの機能及び実現法に関して考察を行なう。次にプログラミングシステムにおける入出力について述べ、ウインドウ、キーボード・コントローラ、データベース・サーバなどの考察を行なう。最後にシェル機能の記述を行なった後、これらの要素を組み合わせ、プログラミングシステムを構成する。

なお、本報告はプログラミングシステムの構成法について提案を行なうものであると同時に、広く並列問題解決や協調問題解決に関する技法のベースを提供する。

A Simple Programming System Written in GHC

Jiro Tanaka and Yukiko Ohta†

Institute for New Generation Computer Technology
1-4-28, Mita, Minatoku Tokyo 108 Japan
†Fujitsu Social Science Laboratory

A programming system can be defined as an environment where one can input/execute programs. In this paper, we try to describe a simple programming system written in GHC. We try to capture the function of "metacall" first. Input/Output problem in GHC are also considered. After describing "shell," we try to assemble these parts into a simple programming system. This paper presents us not only the composing methods of a programming system, but also the basis for parallel/distributed problem solving.

1.はじめに

並列論理型言語には、Clark とGregory によるRelational Language [Clark 81]やParlog [Clark 85] 、ShapiroによるConcurrent Prolog [Shapiro 83]などがある。GHC [Ueda 85, Furukawa 87]はこれらの言語の改良版として提案されたものである。これらの並列論理型言語では、言語のレベルでプロセスを動的に生成でき、プロセス間の同期の記述なども容易である。これらの特徴から、並列論理型言語でオペレーティング・システムを書いてみたり、並列システムのシミュレーションをやってみたらどうかということを思いつく。Parlogでは PPS(Parlog Programming System) [Foster 86, Clark 87] 、Concurrent Prolog では Logix [Silverman 86, Hirsch 86]というオペレーティング・システムがそれぞれ開発されている。本稿では、GHC を用いた簡単なプログラミングシステムについて考察を行う。

本稿の構成は以下の通りである。まず2.でプログラミングシステムの基本機能を提供するメタコール及びその記述に関して述べる。3.ではプログラミングシステムにおける入出力の扱いについて述べる。4.ではシェルおよびプログラミングシステムの全体構成について述べる。5.では、まとめ及び今後の課題を述べる。

2.メタコール及びその記述

ユーザ・プログラムはプログラミングシステムの上で実行されるが、ユーザ・プログラムの実行が失敗したときシステム全体が失敗してはこまる。そこで、システムをユーザ・プログラムの失敗から保護するメタコール機能が必要となる。（このメタコールの事を、ICOTでは荘園(Sho'en)と呼んでいる。）メタコールとは、与えられたゴールを実行し、その実行が成功したか失敗したかを知らせてくれる特殊な述語である。メタコールでは、実行されるゴールが失敗しても、メタコール自体が決して失敗することがない。

メタコールには様々な種類があるが、一番簡単なのは次の二引数メタコールである。

call(G,R)

このメタコールは、ゴールG を実行し、そのゴールが成功するとR にsuccess 、失敗するとR にfailure をかえす。しかしこのメタコールでは簡単すぎてあまり役には立たない。次に考えるのは、メタコールのゴール実行をシステムがコントロールしたいということである。すなわち、ゴールの実行を途中で一時中断(suspend) したり、再開(resume)したり、またその仕事を放棄(abort) したりしたい。そのために考えられているのが、以下の形式の三引数メタコールである。

call(G, C, O)

ここでC とO はそれぞれストリームである。C はコントロール・ストリームといい、システムからメタコールへの制御情報の入力、O はアウトプット・ストリームと呼ばれメタコールからシステムへの通信に使われる。

例えば、このメタコールの実行中にユーザがゴールの実行を一時中断したくなったらC を[suspend | C'] と具体化する。この時メタコールは休止状態になり、メタコールのコントロール・ストリームだけが次の指令をまつ。メタコールを再開したくなった時にはC'を[resume | C'']と具体化する。メタコール自体が不要となったときには、コントロール・ストリームを[abort | C'] と具体化する。また、メタコールの現在の作動状態を知りたいときには、コントロール・ストリームを[state | C'] と具体化する。そうするとアウトプット・ストリームから、メタコールの作動状態 (run, suspended, aborted)を示す出力が上がってくる。

また、メタコールの実行が成功するとアウトプット・ストリームは[success] 、失敗すると[failure] に具体化される。アウトプット・ストリームからは、この他にメタコールの実行中に生じた例外やユーザへの出力など各種の情報が上がってくる

さて、以上がメタコールの概要であるが、次にこの三引数メタコールをどのように実現するかということを考える。このメタコールは効率の上からは組込みの述語として用意する事が望ましいが、組込述語として処理するにはあまりにも複雑である。また組込述語として用意すると、例外やユーザへの出力などの扱いの柔軟性が失われてしまう。そこで、この矛盾の解決策として、メタコールを二つの要素に分けて実現することを考える。すなわちメタコールの基本機能を実現する部分（Exec部分）とその他のサービスを行う部分(Process_server部分）である。

まずExec部分であるが、Exec部分は組込述語として用意することが望ましいが、効率を忘れればメタ・インタプリタの形式で以下のように記述することもできる。

```
exec(true.In.Out) :- true  | Out=[success].
exec(false.In.Out) :- true | Out=[failure(false)].
exec((A.B).In.Out) :- true | exec(A.In.O1),
    exec(B.In.O2).omerge(O1.O2.Out).
exec(A.In.Out) :- sys(A).var(In) |
    sys-exe(A.In.Out).
exec(A.In.Out) :- reflect(A).var(In) |
    Out=[A | Out1].
exec(A.In.Out) :- user(A).var(In)  |
    reduce(A.In.Body.Out.NewOut),
    exec(Body.In.NewOut).
exec(A.[susp | In].Out) :- true | wait(A.In.Out).
exec(A. [abort | In].Out) :- true | Out=[aborted].
wait(A.[resume | In].Out) :- true | wait(A.In.Out).
wait(A. [abort | In].Out) :- true | Out=[aborted].
```

このexecの特徴は、常に引数でメタ・レベルとの連絡がストリームとして確保されていることである。（すなわち、ゴールの成功や失敗という概念は、絶対的な概念ではなくメタ・レベルへのメッセージと考えられている。）ここでは入出力述語はreflectiveな述語として扱われ、そのままアウトプットストリームに送られる。（またこのプログラムの中で使われているvar(In) は、あくまでチェックした時点で情報が来ていない事を確かめているだけであることに留意されたい。）

なお本稿では述語reduceの詳しい記述は省略するが、与えられたゴールA がユーザ定義述語であるとき、述語reduceは、以下のようにふるまう。

(1) ユーザ定義ゴールA に頭部がユニファイ可能な候補節を捜す。
(2) 候補節が無かった時にはBodyをundefined(A)に具体化する。
(3) 候補節があったときには候補節のガードを実行する。
(4) ある候補節のガードが成功すればBodyをその候補節のbody部分に具体化する。
(5) 候補節のガードがすべて失敗したときにはBodyをfailure(A)に具体化する。

また、その他のサービスを行う部分(Process_server部分) は以下のように記述できよう。

```
process_server(State.G.El.[success | EO].In.Out):-var(In) |
        Out=[output([success.goal=G])].
process_server(State.G.El.[failure(G)  | EO].In.Out):-var(In) |
        El=[abort].
        Out=[output([failure.failed_goal=G])].
process_server(State.G.El.[G | EO].In.Out):-var(In).reflect(G)  |
        Out=[G  | Out1].
        process_server(State.G.El.EO.In.Out1).
process_server(State.G.El.[undefined(G)  | EO].In.Out):-var(In) |
        Out=[input([undefined_goal=G, expected_result?].NG) | Out1].
        G=NG.
        process_server(State.G.El.EO.In.Out1).
process_server(State.G.El.EO.[C  | In].Out):-true |
        control_receiver(C.State.G.El.EO.In.Out).
```

プロセス・サーバは六つの引数を持ち、第一引数がプロセスの作動状態、第二引数が実行されているゴール、第三、第四引数はexecへの入出力、第五引数、第六引数が外の世界への入出力である。この中で第一、第二引数は、主とし

て外からの問合せに答えるときの情報として使われる。

また本稿では詳しい記述は省略するが、コントロール・レシーバは以下のようにふるまう。

(1) 動作状態の問合わせに対しては自分の状態を見て答える。

(2) abort 、susp、resumeなどのメッセージについては、メッセージにより自分の状態を変更し、execにメッセージを転送する。また自分の状態の変更が困難であれば、 already_suspendingや already_resumed などの情報を出力する。

このexecと process_serverによりメタコールは以下のように表現できる。

```
call(G,In,Out) :- true |
    process_server(run,G,E1,E0,In,Out),
    exec(G,E1,E0).
```

すなわちexecと process_serverの対でメタコールの機能を果たす事に留意する必要がある。

3.入出力

入出力の扱いであるが、これについては、入出力デバイスに対応するプロセスを仮想的に考え、入出力デバイスに対応する仮想プロセスがシステムと一本のストリームでつながっていると仮定する。入力と出力とを区別して、仮想プロセスとシステムとが二本のストリームでつながっていると仮定する事も可能であるが、その場合入力と出力とのタイミング合わせが大変になる。一本のストリームでつないだ場合には、デバイスに対応する仮想プロセスは常にストリームを消費していると考える。すなわち入力についても、システムから入力の要求を送ってやる必要がある。

一般に、入出力デバイスに対応する仮想プロセスをプログラムの中で作るにはcreate述語を実行すれば良い。create述語は特殊な述語で、第一引数でデバイスの種類を指定すると、第二引数で仮想プロセスへのストリームをつかむことができる。入出力には、この仮想プロセスへのストリームにメッセージを送ってやれば良い。メッセージは、前に挙げた process_serverの記述でもわかるよう、Input(出力メッセージリスト、X)、output(X) の形をしている。(Inputの場合、メッセージリストの内容を出力してから入力モードになり、入力された情報がX に束縛される。まずメッセージリストの内容を出力するのは、実際の便利さを考えてのことである。) またデバイスに対応する仮想プロセスはX に [] が具体化されたとき消滅する。

3.1 ウインドウ

ウインドウは通常のビットマップ・ディスプレイの上に生成され、その上から入出力が行える。例えば、create(window,X)が実行されると、ウインドウが一つ生成される。そのウインドウへの入出力にはX にメッセージを送れば良い。(実際のウインドウ入力は、まずマウスによりカーソルを入力を行うウインドウの上にのせ、キーボードから文字を打ち込んでやれば完了する。) またウインドウを消滅させるにはX を [] に具体化してやれば良い。

3.2 キーボード・コントローラ

既に述べたように、入出力はプログラムからの要求で行われる。すなわちプログラムからの要求がなければキーボードから文字を打ち込んでやっても受付けない。従って、システムがいつもユーザからの入力を受け付けるようにするためには、常にデバイスに入力を要求するプロセスを作る必要がある。プログラミングシステムでその働きをするのがキーボード・コントローラである。キーボード・コントローラのプログラムは以下のように書ける。

```
keyboard(Out,In):-true |
        Out=[Input([C],T) | Out1],
        keyboard(T,Out1,In).

keyboard(halt,Out,In):-true |
        Out=[],
        In=[].
```

```
keyboard(T,Out,In):-goal__or__command(T) |
        In=[T | In1],
        Out=[input([0],T1) | Out1],
        keyboard(T1,Out1,In1).
```

トップレベルの述語keyboardは二引数述語であり、第一引数はデバイスへのストリーム、第二引数はシステムの入力ストリームである。二引数keyboardは、デバイスからの入力をバッファしてすぐ三引数keyboardを呼び出す。バッファされた入力がhaltであればkeyboardプロセスは終了するが、そうでなければ、バッファされた情報をシステの入力ストリームに入れ、再び入力を要求し、入力情報をバッファしてから三引数keyboardを呼び出す。

3.3 データベース・サーバ

プログラミングシステムでは、ユーザがプログラムの定義を入力したり、それを更新したりする必要がある。こうしたデータベースへのアクセスも、広い意味では入出力の一種と考えられる。こうしたデータベースへのアクセスを扱うには、データベースに対応する仮想的なプロセス（データベース・サーバ）を考え、データベースへの参照、入力、更新などはすべてこの仮想的なプロセスへのメッセージとして実現することが考えられる。

実際のところ、[Clark 87]などに見られるよう、PPS ではデータベースを（少なくともみかけでは）そのように実現することをめざしている。しかしながら、その場合、exec　でのデータベースへの参照にもいちいちデータベースへのストリームを持ち運ぶことになり面倒である。簡単化のため、本稿ではストリームと副作用を併用することとした。その場合データベース・サーバのプログラムは以下のように記述できる。

```
db__server([add(Code) | In],ready,Out):-true |
        add __difinition(Code,Done,Out,Out1),
        db__server(In,Done,Out1).
db__server([delete(Name) | In],ready,Out):-true |
        delete__definition(Name,Done,Out,Out1),
        db__server(In,Done,Out1).
db__server([definition(Name) | In],ready,Out):-true |
        definition(Name,Done,Out,Out1),
        db__server(In,Done,Out1).
```

ここでトップレベルの述語db__serverは三引数述語であり、第一引数は入力、第二引数はアクセスの逐次性を保障するためのフラグ、第三引数は出力である。（このプログラムは極度に簡単化してある。むろんフラグを複雑化してして定義の検索が並列に起るよう改良することは可能である。）

4.プログラミングシステムの構成

前節までで、プログラミングシステムにおけるメタコールの記述および入出力の扱いについて考察した。次の課題は如何にしてプログラミングシステムを構成するかということである。そこで本節では、まずプログラミングシステムの中核となるシェルのプログラムを示し、次にプログラミングシステムの構成例を示す事にする。

4.1 シェル

ユーザからの入力によりユーザ・タスクを起動したり、プログラムを入力したりするのがシェルの機能である。シェルについては[Shapiro 83, Clark 84]でも既に触れられている。シェルの記述例を以下に示す

```
shell([],Val,Db,Out):-true |
        Val=[],
        Db=[],
        Out=[].
```

```
shell([goal(Goal) |n].Val.Db.Out):-true |
        Val=[ record_dict(Goal.NGoal) | Val1].
        create(window.WOut).
        keyboard(KO.PI).
        process-server(run.NGoal.EI.EO.PI.PO).
        exec(NGoal.EI.EO).
        shell(In.Val1.Db.Out).
        merge(KO.PO.WOut).
shell([db(Message) | In].Val.Db.Out):-true |
        Db=[ Message | Db1].
        shell(In.Val.Db1.Out).
shell([binding(Message) | In].Val.Db.Out):-true |
        Val=[Message | Val1].
        shell(In.Val1.Db.Out).
```

このシェルは、ストリームで変数辞書へつながっていて、ユーザが勝手に変数とその束縛値を定義できるようになっている。上のプログラムでシェルは四つの引数を持つが、第一引数は入力ストリーム、第二引数は変数辞書へのストリーム、第三引数はデータベース・サーバへのストリーム、第四引数は出力ストリームである。

このシェルのプログラムは次のように動く。

(1) 入力ストリームが[]であれば、それは入力の終了を意味するので、第二引数～第四引数の出力ストリームを閉じる。

(2) 入力ストリームからゴールがgoal(Goal)という形式で入力されると、まず変数辞書へGoalを送る。変数辞書ではGoalに含まれている変数の束縛値を調べてゴールNGoal を作る。また入力されたゴールに対応するexecとプロセス・サーバを起動する。これと同時に、キーボード・サーバとウインドウも起動される。

(3) 入力ストリームから、データベース・サーバや変数辞書に対するメッセージを受けとると、それぞれのストリームへメッセージを送る。

シェルへの入力ストリームによりこれらのプロセスが次々に起動されていく様子を示したのが図 1である。

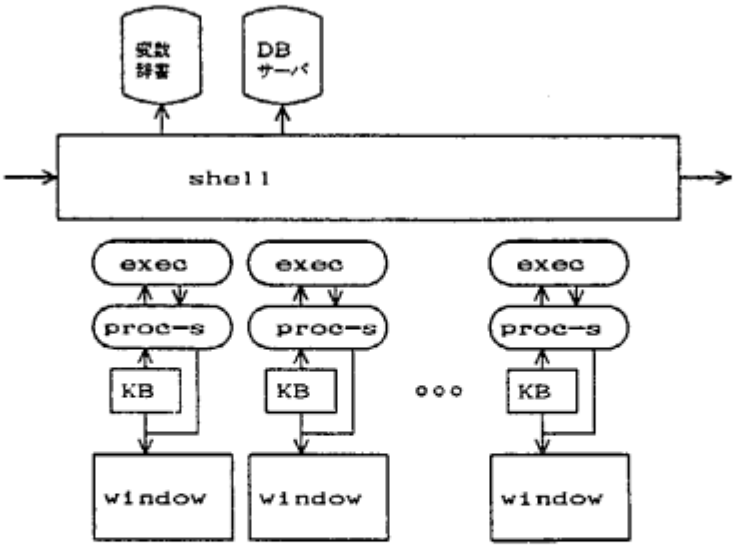

図 1 シエルによるプロセスの起動

ここでは、シェルの入力ストリームからゴールが入力されると、それに対応して exec, process_server, keyboard, windowの四つのプロセスが起動されていく。このプロセスの組は独立したウインドウを持ち、もはやシェルから

独立して動くことに留意されたい。

なお変数辞書を管理するプロセスは次のように記述できる。

```
vr_dictionary([record_dict(G,NG) | In],Dict,Out):-true |
        record_dict(G,NG,Dict,NDict),
        vr_dictionary(In,NDict,Out).
vr_dictionary([set(Var:Value) | In],Dict,Out):-true |
        Out=[output([set_binding(Var:Value)]) | Out1],
        set _value(Var:Value,Dict,NDict,Out,Out1),
        vr_dictionary(In,NDict,Out1).
vr_dictionary([value(Var) | In],Dict,Out):-true |
        look_up_var(Var,Dict,Out,Out1),
        vr_dictionary(In,Dict,Out1).
vr_dictionary([reset | In],Dict,Out):-true |
        Out=[output([reset_binding]) | Out1],
        vr_dictionary(In,[],Out1).
vr_dictionary([reset(Var) | In],Dict,Out):-true |
        Out=[output([reset_binding(Var)]) | Out1],
        reset _var(Var,Dict,NDict,Out,Out1),
        vr_dictionary(In,NDict,Out1).
```

4.2 プログラミングシステムの構成例

これまでに解説した要素を結合し、簡単なプログラミングシステムを構成する例を以下に示す。

```
create_world:-true |
        create(window,Out),
        keyboard(Out1,In),
        shell(In,Va,Db,Out2),
        vr_dictionary(Va,[],Out3),
        db_server(Db,ready,Out4),
        merge4(Out1,Out2,Out3,Out4,Out).
```

このプログラミングシステムを図に示したのが図 2である。

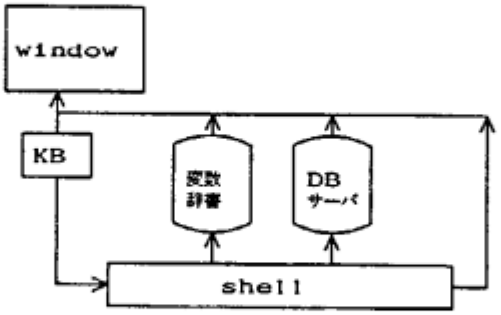

図 2プログラミングシステムの構成例

ここでは、システム・ウインドウ、キーボード・コントローラ、シェル、変数辞書、データベース・サーバがまず起動させられている。システム・ウインドウにはシェル、変数辞書、データベース・サーバからの出力がキーボード

・コントローラからの入出力と合流してつながれている。

キーボード・コントローラはシステム・ウインドウに読み込み要求を常に送っているので、ユーザはシステム・ウインドウから変数辞書やデータベース・サーバの制御コマンドを入力することが出来る。また、ゴールを入力して新たにそのゴールを処理するタスクを次々に起動していくこともできる。

なお、ここで示したのは、あくまでプログラミングシステムの構成の一例であることを指摘しておきたい。

5.まとめ及び今後の課題

以上、プログラミングシステムにおけるメタコールの記述および入出力の扱い、またプログラミングシステムの構成について考察した。しかしながら、ここで示した極度に簡単化したものであり、PPS やLogix 等と比較すれば、まだ基本機能しか供えていない。この基本機能の拡張、例えば、本稿では特に並列ハードウェアを意識していないが、並列環境への拡張 [Tanaka 86, Tanaka 87] や、メタコールを機能拡張して資源の制限や管理も行ない得るようにすること[Foster 87] などについては今後の課題である。

最後に、なぜ今、プログラミングシステムなのかという疑問に答えておかなければならない。並列論理型言語の歴史を振り返ってみると、最初に言語仕様の提案が成され、次に処理系作りが成されている。しかしながら言語設計や処理系作りの時代は既に過ぎ、今は環境作りや問題解決手法に研究の重点が移ってきている。

GHC を使えばプログラミングシステムが極めて自然に表現できる事は本稿でも明らかである。こうしたプログラミングシステムを表現するための技法は、[Ohwada 87] 等でもみるように、並列問題解決、協調問題解決などのための技法と非常に類似している。すなわち、プログラミングシステムの考察とは典型的な並列問題解決、協調問題解決の手法の考察に他ならない。

なお、本研究は第５世代コンピュータ・プロジェクトの一環として行なわれたものである。

[参考文献]

[Clark 81] K.Clark and S.Gregory: A Relational Language for Parallel Programming, In Proceedings of the 1981 Conference on Functional Programming and Computer Architecture, pp.171-178, ACM, Oct. 1981.

[Clark 84] K.Clark and S.Gregory: Notes on Systems Programming in Parlog, In Proceedings of the International Conference on Fifth Generation Computer Systems 1984, pp.299-306, ICOT, 1984.

[Clark 85] K.Clark and S.Gregory: PARLOG: Parallel Programming in Logic, Research Report DOC 84/4 , Dep. of Computing, Imperial College of Science and Technology, Revised 1985.

[Clark 87] K.Clark and I.Foster: A Declartive Environment for Concurrent Logic Programming, Lecture Notes in Computer Science 250, TAPSOFT'87, pp.212-242, 1987.

[Foster 86] I.Foster: The Parlog Programming System (PPS), Vesion 0.2, Imperial College of Science and Technology, 1986.

[Foster 86] I.Foster: Logic Operating Systems: Design Issues, In Proceedings of the Fourth International Conference on Logic Programming, Vol.2, pp.910-928, MIT Press, May 1987.

[Furukawa 87] 古川　溝口編：並列論理型言語GHC とその応用，共立出版，1987.

[Hirsch 86] M.Hirsch et al.: Layers of Protection and Control in the Logix System, Weizmann Institute, Israel, 1986.

[Ohwada 87] H.Ohwada and F.Mizoguchi: Managing Search in Parallel Logic Programming, in Proceedings of the Logic Programming Conference '87, pp.213-222, ICOT, June 1987.

[Shapiro 83] E.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, 1983.

[Silverman 86] W.Silverman et al.: The Logix System User Manual, Version 1.21, Weizmann Institute, Israel, July 1986.

[Tanaka 86] 田中　他: 並列論理型言語に基づくオペレーティング・システム，日本ソフトウェア科学会第三回論文集，pp.73-76, 1986.

[Tanaka 87] 田中: Architecture Abstraction in GHC,　情報処理第35回全国大会，5Q-6，1987年 9月.

[Ueda 85] K.Ueda: Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985.