KL1 入門

近山 隆

目次

1 KL1 の実行機構 **5**

1.1 プログラムの形式と基本実行機構 5

1.1.1 簡単なプログラム 5

1.1.2 プログラムの実行 6

1.1.3 ユニフィケーション 6

1.1.4 節の形式と基本実行機構 7

1.1.5 並列実行, 通信, 同期の機構 8

1.1.6 実行機構の補足 9

1.2 KL1 の実行機構の特徴 10

1.2.1 手続き型言語と KL1 10

1.2.2 関数型言語と KL1 10

1.2.3 他の論理型言語と KL1 11

1.2.4 オブジェクト指向言語と KL1 11

2 KL1 で扱えるデータ **12**

2.1 変数に型はない 12

2.2 アトミックなデータ 12

2.2.1 記号アトム 13

2.2.2 数値アトム 13

2.3 構造を持ったデータ 14

2.3.1 ベクタ 14

2.3.2 ストリング 15

2.3.3 コンス 15

2.3.4 不完全データ構造 16

2.3.5 構造データのユニフィケーション 16

3 プロセスとストリーム通信 **18**

3.1 略記法 18

3.1.1 ガード・ユニフィケーションのヘッドでの表記 18

3.1.2 ゴール true の省略 19

3.2 プロセス 19

3.2.1 KL1 のプロセスとは 19

3.2.2 リスト要素の和 20

3.2.3 自然数のリスト 21

3.3 ストリーム通信 22
3.3.1 プロセスの複合 22
3.3.2 ストリーム通信 23
3.3.3 プロセスの状態 24
3.3.4 複雑なメッセージ 25

4 プロセス・ネットワーク **27**
4.1 フィルタ 27
4.2 ストリームの連結 28
4.3 マージャ 30
4.4 組込みのマージャ機能 32
4.5 ディストリービュータ 33
4.6 サーバ 33

5 実行の仕方の指定 **35**
5.1 KL1 の並列実行指定の方針 35
5.1.1 負荷分散はプログラムで指定する 35
5.1.2 ふたつの並列性の分離 36
5.2 ゴール分散プラグマ 36
5.3 ゴール優先度指定プラグマ 37
5.4 節優先度指定プラグマ 37

はじめに

KL1 は並列論理型言語 Guarded Horn Clauses (GHC)[5] に基づいて設計された, 並列処理の記述に適したプログラム言語で, 第五世代計算機プロジェクトが目標としている並列推論システムの中核となる並列推論マシンの共通核言語として, オペレーティングシステム [1] から種々の応用プログラムの記述にまで用いられている.[6] 本稿はこの KL1 の言語仕様とプログラミング技法について解説するものである.

ひとことで KL1 という言語の特徴を言うならば:

KL1 は Guarded Horn Clauses に基づく並列論理型言語である

ということになるだろう. しかし, これではこの分野に首を突っ込んでいる者以外には何のことかさっぱりわかるまい. もう少し普通の言葉でいえば, KL1 は:

KL1 記号処理を並列に行なうための言語である

ということになろう.

記号処理では複雑に絡み合うなデータを取り扱う必要が生じることが多い. このため, こうしたデータをどのようなデータ構造で表現するかが重要になる. また, そうしたデータ構造をどのようにメモリ上 (あるいはディスク装置上) で管理するかも問題である. こうした管理に多くの労力を割くとより本質的なプログラミングにかけられる労力が少なくなってしまうので, 言語システムで基本的な機能を用意しよう, というのが記号処理言語である. 具体的には, 一意性のあるものに名前をつけると自動的に一意な表現を与えてくれる**記号アトム**の機構や, 標準的なデータ構造についてその割り付けや解放についてのプログラミング労力を減らしてくれる**自動メモリ管理**機構などが, 代表的な特徴になる. この点, KL1 は代表的な記号処理言語である Lisp と同様の機能を提供している.

並列処理のためには, 全体の処理をを複数の部分処理に分割し, それらが必要なところでは同期を取りながら計算を進める必要がある. このために普通の逐次処理言語に並列実行を指定する機構と同期のための機構を追加し, 並列処理にも使えるようにした言語 (あるいは OS の機能まで含めたシステム) は数多い.

しかし, こうしたアプローチにはふたつの大きな問題点がある. ひとつには, 原則は逐次処理のままなので, 最初のプログラミング時から並列に実行できる部分を指定しなければならないことである. このため, 同じプログラムを並列度の大きく異なるハードウェアで共通に使えてしかも効率良く動くようにすることは難しく, ハードウェアごとにプログラム全体を書き直す必要が生じる. まして, どのような並列処理が適当か良くわからない問題について, 並列処理の仕方を模索し実験を積み重ねながらプログラミングしていく際など, そのつどプログラムを書き直すのはそのつどデバッグし直すことになり, 多大な労力を必要とする.

もうひとつは同期処理の面倒さである. 同期の必要性を意識してプログラミングするのは非常に厄介であるし, 同期にバグが入ると, そのバグがどのように表面化するかに再現性がない (実行するたびに違う現象が起きる) ので, デバッギングが非常に困難になる.

KL1 は逐次処理に並列実行のための機構を付加するという方法ではなく, 最初からすべては並列に実行できることを前提とした言語になっている. こうすると非常に頻繁に同期が必要になるのだが, **データフロー同期機構**を導入して同期を自動化することによって, プログラマによる同期の誤りが入り込む可能性を排除している. 物理的な並列実行の指定は別途**プラグマ**と呼ばれる記述によって行なうこととし, プラグマはプログラムの正当性[1]を変えないように設計してあるので, 並列処理の仕方を変えるたびにデバッグし直す必要がない.

こうした KL1 の特徴のおかげで, 並列処理ソフトウェアの研究開発の労力は非常に軽減される. 同じプログラムのプラグマ部分を変更するだけでさまざまな並列実行の仕方を指定できるので, いったんバグを取ってしまえば並列実行指定を変えるたびにバグに悩まされる必要はない. このため, 並列処理のもっとも困難な課題である負荷分散方式の研究を, 数多くの実験を伴いながら実証的に進めることができるようになるわけである.

[1] 正確には停止性を除いた部分正当性.

第 1 章

KL1 の実行機構

本章では KL1 の実行機構の概要を説明する.

1.1 プログラムの形式と基本実行機構

まず KL1 のプログラムの形式と, 基本的な実行機構について, 簡単な例を用いて解説しよう.

1.1.1 簡単なプログラム

もっとも簡単な KL1 プログラムの例として, 入力の 0, 1 を反転して出力するインバータを考えてみる.

```
:- module inverter.
:- public not/2.

not(In, Out):- In = 0 | Out = 1.
not(In, Out):- In = 1 | Out = 0.
```

最初の二行はプログラムのモジュール化のためのおまじないで, ここでは気にしなくて良い. 後の二行がプログラムの本体である. それぞれ:

- もし not の第一引数 In が 0 だったら, 第二引数 Out は 1 にする
- もし not の第一引数 In が 1 だったら, 第二引数 Out は 0 にする

と読む. このふたつの行はそれぞれ**節** (clause) と呼び, プログラム定義の単位である.

それぞれの冒頭部分 "`not(In, Out)`" は, この節が述語 not に関する定義であること, 引数はふたつで, この定義中ではそれぞれ In, Out と呼ぶことを宣言している. この部分を**ヘッド** (head) と呼ぶ. **述語**という名称を使うのは論理型言語に共通だが, ここでは単に手続き, あるいはサブルーチンのことだと思ってさしつかえない.

ヘッドに続く ":-" の後, 縦棒 ("|") の前までを**ガード** (guard) と呼び, この節を選んで良いかどうかの条件を指定する部分である.

縦棒の後, フルストップ (".") までを**ボディ** (body) と呼び, 実際何をするかを指定する部分である. ボディにはいろいろなものが書けるが, この例では出力用の引数の値を決める操作である "`Out = 1`" などを行なっている.

1.1.2 プログラムの実行

前掲のインバータのプログラムを会話的に実行すると, 以下のようになる.

```
?- inverter:not(1, X).
X = 0
yes.
```

会話的な実行にはプロンプト記号 (“?-”) の後に呼び出したい述語名と引数並びを書き, フルストップで終る. 述語名 not の前の “inverter:” は述語を定義するモジュールの指定である. 述語名と引数の並びを合わせて**ゴール** (goal) と呼ぶ. ゴールは KL1 プログラムの実行の単位である.

ここでは引数に 1 と X を与えた. 整数 1 は普通の整数値 1 を表す. KL1 では大文字で始まる名前は変数を表す. ここでは他にどこにも出てこないまっさらの変数 X を第二引数に渡したわけである.

プログラムの実行は以下のようになる.

1. 第一引数の In は会話的な呼び出しの時に与えた引数 1 に対応するので, ふたつあった節のうち後の方の選択条件を満たしている. そこで, この節を選ぶ.

2. 節が選ばれるとそのボディを実行する. ボディでは第二引数 Out (ここでは会話的に呼び出した時に書いた X に対応づけられている) の値として 0 を与える.

3. 他にすることはないので, これでおわり.

この実行の結果, 引数に与えた X の値が 0 に決まったので, それが次の行に表示されている.[1]その後の “yes” は実行がつつがなく終了したことを示すものである. [2]

最初に与える第一引数を 0 に変えれば, 当然:

```
?- inverter:not(0, Y).
Y = 1
yes.
```

のように結果は 1 になる.

1.1.3 ユニフィケーション

前掲のインバータのプログラムの中の “In = 0” や “Out = 1” のような, 等号 (“=”) の両辺に値を書いた形のゴールを**ユニフィケーション** (unification) と呼ぶ. 等号を用いているように, これは両辺の値が等しいことを意味するのだが, より詳しくはユニフィケーションがどこに現れたかによって異なる.

---

[1]KL1 の変数は C のような手続き型言語でいう『変数』とは大きく異なるものである, という点である. 手続き型言語では『変数』は値の格納場所であって, 計算の進行に伴って格納されている値は変化する. KL1 の変数はもっと数学でいう変数に近いもので, 値は決まっていないか決まっているかのどちらかで, いったん値を決めたら後で変わることはない.

[2]後に述べるが, プログラムの実行はつつがなく終了するとは限らない.

ガード・ユニフィケーション　節の適用条件を指定するガードに現れるユニフィケーションは, 両辺が等しいことが適用条件の一部であることを示すものである. これは単に等しいかどうかを試すだけなので受動的なユニフィケーション (passive unification) とも呼ばれる. 前掲の例の "`In = 0`" では In が 0 かどうかを調べるわけである.

ボディ・ユニフィケーション　実行を指定するボディに現れるユニフィケーションは, 両辺を等しいものにするという操作を意味し, 能動的なユニフィケーション (active unification) とも呼ばれる. 前掲の例の "`Out = 1`" では, まだ値の決まっていなかった Out の値を 1 に決めてしまうことによって, 両辺を積極的に等しいものにしてしまっているわけである.

1.1.4　節の形式と基本実行機構

プログラムを定義する節は, 一般には以下のような形をしている.

述語名 (引数, ...) :- ガード | ボディ.

それぞれの部分の役割は以下の通りである.

述語名: 節で定義 (の一部) を与える述語 (手続き) の名前.

引数: 述語の引数と節の中で使う変数名の対応づけをする仮引数並び. KL1 では変数名の有効範囲はひとつの節の中 (および, ひとつの会話的な実行指示の中) だけで, 同じ名前でも別の節にあればまったく別のものとみなされる.

ガード: 節の適用条件を指定する部分. カンマで区切ってゴールをいくつでも書け, すべてを満足した場合にだけ適用条件を満たしたものとする. ガードにはユニフィケーションと, 言語で定義するある決まった種類の述語の呼び出しだけが書ける.

ボディ: 節を選んだときに実行すべき部分. やはりカンマで区切ってゴールをいくつでも書け, その節が選ばれたらすべてを実行する. ボディにはユニフィケーションや任意の述語を呼び出すゴールが書ける.

KL1 で呼び出せる述語には, あらかじめ言語で定義する**組込述語** (built-in predicates) と, プログラム中に定義する**ユーザ定義述語** (user-defined predicates) の二種類がある.

組込述語の多くは種々のデータについての基本操作や基本的な値の検査のためのもので, それらがどのように動作するかは KL1 言語の仕様の一部として定義される.

特殊な組込述語として, 引数のない述語 "`true`" がある. これはガードに現れれば常に真 (つまり無条件), ボディに現れれば何もしないこと (no operation) を意味する.

ユーザ定義述語についてのゴールの実行では, その述語を定義する節がいくつかあるとすると, まず各節の適用条件であるガードの真偽を試す. 次にガードが真であるとわかった節をひとつ選び, そのボディのゴール (ユニフィケーションを含む) をすべて後に実行すべきゴールとする. これを繰り返すのが KL1 プログラムの実行過程である.

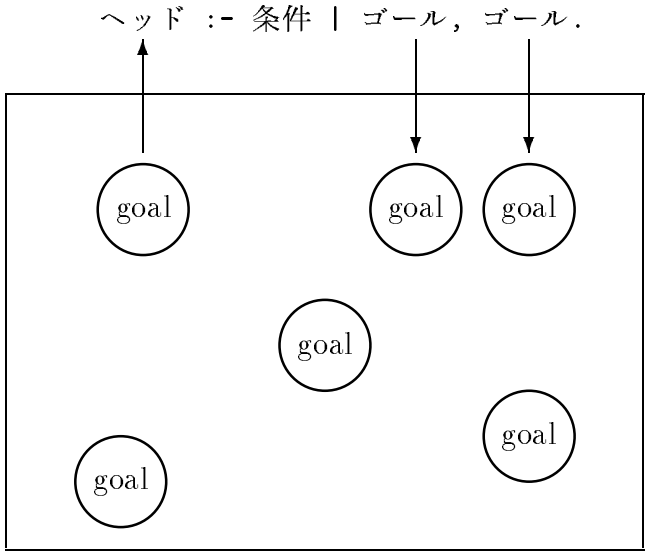

図 1.1: KL1 の実行機構

1.1.5 並列実行, 通信, 同期の機構

節はある条件 (ガード) を満たすゴールを, ボディのゴールに書き換える書換ルールであると見ることもできる. 図 1.1 に示すように書き換えるべきゴールの集合[3]から任意のものを取り出し, この書換えを施した結果をゴールの集合に戻すのが実行の 1 ステップになる. 書換えを繰り返すことによって最終的にはすべてのゴールが何もしないゴール "`true`" にまで書き換えられて実行が終了するとすると[4], この書換えは**簡約化** (reduction) であると考えられる.

書換えはかならずしも逐次的にひとつずつ行なわなくとも良い. 複数のゴールについていっぺんに行なえば, 実行は並列になる.

KL1 では複数のゴールが同じ**変数を共有**することがある. あるゴールが変数の値を決めると, それを共有する他のゴールがその値を使って, どの節を使って簡約化するかを決めることもできる. これが KL1 の通信機構である.

前掲のインバータのプログラムについて, 以下のようなゴール群を実行することを考えてみよう.

```
?- inverter:not(1, X), inverter:not(X, Y).
```

このように実行すると, X で示す変数はふたつのゴールで共有することになる.

KL1 では, 複数のゴールがあるときにはどの順番に実行するかは特に決めておらず, 処理系の都合で決めて良いことになっている.[5]

[3]同じものがあってもよいので, 正確にはマルチ集合.

[4]KL1 ではいつまで待っても実行が終らないが有用なプログラムというものも考えられる.

[5]後述の優先度指定機構はあるが, 絶対的なものではない.

仮に, まず左のゴールから実行するものとしてみよう. すると第一引数は 1 だから, 第二引数 X の値は 0 に決まる. この X は右のゴールも第一引数として共有しているので, 右のゴールの第一引数は 0 になったわけである. そこで, 右のゴールを実行すると Y は 1 に決まるわけである. したがって, このゴール列の実行の結果, X は 0 に, Y は 1 になる.

実行の順序が変わったらどうなるだろう. 最初に右のゴールを実行しようとしたとする. すると, 第一引数 X の値は決まっていないから, 述語 not のふたつの節の適用条件であるガードの, それぞれ `In = 0`, `In = 1` の In の値が決まっていないことになる. これではどの節を使って良いのかわからない. このような場合, 当面の間このゴールの実行は見合わせることにする. これを実行の**中断** (suspension) と呼ぶ.

右のゴールが実行できないとなると, 左のゴールしかすることはない. そこで左のゴールを先に実行すると X の値が 0 に決まる. これで先程実行を中断していた右のゴールの中断理由はなくなった. そこで, 右のゴールの実行を再開する. こんどはガードを真偽を調べるのに障害はなく, 無事 `In = 0` の節が選ばれ, Y の値は 1 に決まる. これが KL1 の通信と同期の機構である.

要約すると以下のようになる.

- **通信は共有変数**を用いて行なう
- **同期はガード**で必要な値を検査すると自動的に行なわれる

KL1 では選択実行 (if-then-else のようなもの) の条件はすべてガードの真偽を調べることによって決める. その判断に必要な値について, まだ値が計算できていない場合は実行を中断して待ち合わせるという自動的な同期が行なわれる. KL1 の変数はいったん値が決まってしまえば, 後からそれが別の値に変わるということはなく, ずっと同じ値を持ち続ける. したがって KL1 プログラムでは単純な計算順序の誤りによって誤った選択をしてしまうという危険性は皆無なのである.

1.1.6 実行機構の補足

以上が KL1 の基本的な実行機構なのだが, ここまでの内容でいくつかまだよく説明していないことがあるので, ここで補足しておこう.

複数の節が使えたら? もしふたつ以上の節のガードがみな真だったら, どの節を選ぶのだろう. KL1 の言語仕様としては, このような場合にどの節を選ぶかあえて決めていない.[6]処理系の自由でどう決めても良いことになっている.[7] したがって, 正しい KL1 プログラムでは, ひとつの述語についてガードを排他的に書くか, 排他的でない場合にはどれを選んでも正しく動くように書くかしなければならない.

[6]節間の優先度を与える機構もあるが, これは絶対的なものではない.

[7]実際, Multi-PSI のような非共有メモリシステム上の処理系では, 近くにデータが揃っていてプロセサ間通信をしなくてもガードの真偽を判定できるようなう節があればそれを選ぶので, 毎回どの節を選ぶかが変わったりする.

どの節も使えなかったら? もしどの節のガードもみな偽だとわかったら, どうするのだろう. たとえば前掲の述語 not を "`not(3, X)`" のような引数で呼んだらどうなるのだろう. この場合はゴールの実行は**失敗** (failure) になり, 正常な実行から外れる. 失敗は演算のオーバフローなどと同様に, 例外として処理する (例外の扱いについては後述する).

ボディ・ユニフィケーションで, 既に値の決まっている変数の値をまた決めようとしたら? 既に決まっている値と同じ値なら何もしなかったのと同じになる. 違う値にしようとしたのならユニフィケーションの失敗になり, やはり例外として処理する. ただし, 両辺とも値が決まっているようなボディ・ユニフィケーションを行なうのは, 推奨するコーディング・スタイルではない. そのような操作の実行効率は良くないし, プログラムもわかりにくくなる.

1.2 KL1 の実行機構の特徴

ここまで述べてきた KL1 の仕様の特徴を, 他の言語と比べて振り返ってみよう.

1.2.1 手続き型言語と KL1

KL1 では変数がその値を変えていくという, 手続き型言語に見られるような考え方は存在しない. KL1 の変数は値が決まっていないか, 決まっているかのいずれかで, 値が決まったらそれがその後に変わるということは決してない. このため, 時々刻々変化する計算状態というものがなくなり, 計算順序についての自由度が大きく上がっている. この特徴は並列処理を記述する際に大きな利点となる.

1.2.2 関数型言語と KL1

時間変化する計算状態が問題にならないという意味で, KL1 は関数型言語に近い.[8] 並列処理時の利点も共通している.

関数型言語との最大の違いは, KL1 には**非決定性** (non-determinacy) があるということである. つまり, まったく同じ計算を複数回行なわせると違う結果が得られる可能性があるのである. この非決定性は複数の節のガードが真である場合に生じる. これはデバッグを考えると欠点なのだが, 効率良く問題を解かせるためには利点になる. 関数型言語のソフトウェアはどんなハードウェアの上でも同じ計算をするが, KL1 のプログラムはハードウェアに応じて, ことにハードウェアにどのような並列性が得られるかに応じて自在の動きをするようなソフトウェアを作れるのである.

[8] ここでいう意味では, 広く使われている Lisp は関数型ではなく, 手続き型言語である.

1.2.3 他の論理型言語と KL1

Prolog と KL1 KL1 のシンタクスは同じホーン論理[9]に基づく論理型言語である Prolog に非常に良く似ている. プログラムを論理式として解釈できる点も同様で, その際にプログラムをどのような論理式に対応づけて解釈するかもまったく同じである. しかし, プログラム言語として見た時, KL1 と Prolog はまったく異なる言語であるといって良い. 両者は共にホーン論理に基づきながら, まったく違う方向に発展した言語なのである.

Concurrent Prolog, Parlog, GHC と KL1 Concurrent Prolog[3], Parlog[2], Guarded Horn Clauses (GHC) は, いわば同じ穴のムジナで, ほぼ同じ方向を向いた言語である. KL1 はこれら committed choice 型などとも呼ばれる並列論理型言語のうち, もっともシンプルな言語である GHC をさらに簡単にした Flat GHC と呼ばれる言語をその基礎としている.

1.2.4 オブジェクト指向言語と KL1

KL1 自身はオブジェクト指向言語ではない. だが, KL1 や他の並列論理型言語で多用されるプログラミング・スタイルは, オブジェクトをプロセスとして実現するオブジェクト指向の (あるいはエージェント指向ともいえる) プログラミング・スタイルである. オブジェクト指向プログラミング・パラダイムのかなりの部分は, ごく素直に KL1 で記述できるのである. このプログラミング・スタイルについては次章以降に解説する.

まとめ

KL1 の言語仕様のもっとも基本的な部分である実行機構について概説した.
次章では KL1 で取り扱うデータについて述べる.

[9]機械的な定理証明が比較的容易で表現力もかなり大きいような一階述語論理のサブセット. Prolog (カットなどを含まないピュアなもの) も KL1 も, この論理についての不完全だが健全な自動証明系とみなせる.

第 2 章

KL1 で扱えるデータ

ここまでの例では具体的なデータとしては 0, 1 などの数値しか出てこなかった. KL1 では他にもさまざまな型のデータが扱える. 本章では KL1 の基本的なデータ型とそれらに対する操作について述べる.

2.1 変数に型はない

KL1 では変数についてどんな型のデータが入るかは宣言しない. ソース・プログラム中の同じ変数が, あるときはひとつの型のデータ, 別のときには別の型のデータを値として持つことがある. このあたりは Prolog や Lisp と同様である.

この仕様には利点も欠点もある. プログラム中の変数にはどんな種類の値が入り得るかが, 原則としては入力データが与えられるまでわからないため, 処理系が最適化しにくいことは欠点のひとつである.[1]また, 人間がプログラムを読むときにも, 変数の型という理解の助けになる情報が欠けている分だけ, 強い型付けをする言語に比べると不便だろう.

一方利点としては, C の union のような特別の記法を用いなくてもさまざまな型のデータを混在させることができること, Ada の generic package のような面倒な宣言をしなくても, いろいろな型のデータに対して共用できるプログラム・モジュールを簡単に作れることなどがある.

全体として, 変数に型がないことはプログラムを読むのには不便だが, 書くときには便利である. これは, 実験的なプログラムを何度も書き直しながら開発していくような場合には有利になる. これは AI 研究に変数に型をつけなくて良い Lisp が広く使われてきた理由のひとつであろう.

2.2 アトミックなデータ

アトミックなデータとは, 内部構造を持たず, その値自身だけに意味があるようなデータである. KL1 で扱えるアトミックなデータ型としては以下のものがある.

記号アトム: 特に何の値を表すわけでもなく, 自己同一性だけに意味がある識別子である. プログラム中で必要になる概念などを一意に表すのに用いる.

[1] この欠点はコンパイル時の解析によって型を推論すれば, 少なくともある程度はカバーできる.

整数: 普通の整数値を表し, 表記も通常は普通の十進記法 (に必要なら符号がついたもの) を用いる (“3”, “-15” など).

浮動小数点数: 実数値の近似値を表す. 表記には小数点を用いた十進記法 (“3.14” など) と, 指数部を加えた記法 (“0.314e+1” など) がある.

2.2.1 記号アトム

KL1 の記号アトムは Lisp などと異なり, 属性を持つことはない. また, 記号アトムの名前の管理もコンパイラやオペレーティングシステムが行なうものであって, 言語自体では定めない. したがって, 記号アトムについて可能な演算はガードでの同一性の比較だけである.

アトムの表記は Edinburgh Prolog のアトムの表記と同様で, 以下のいずれかである.

- 英小文字から始まり, 英数字および下線 “_” の任意個の並び. たとえば “lpc”, “multiPSI”, “psi_3” など.
- 特殊文字 (“~”, “+”, “-”, “*”, “/”, “\”, “^”, “<”, “>”, “=”, “‘”, “’”, “:”, “.”, “?”, “@”, “#”, “$”, “&” のいずれか) の任意個の並び. たとえば “+”, “:-” など.
- ふたつの引用符 “’” でくくられた任意の文字列. ただし, 引用符を含む場合は引用符を二回続ける. たとえば “’Hello world’”, “’’’quoted’’’” など.
- 特殊なアトム. 具体的には “!”, “;”, “[]” の三種類.

2.2.2 数値アトム

整数, 浮動小数点数の数値に対する演算や比較は, 言語のプリミティブとして用意した述語である**組み込み述語** (built-in predicates) を用いて行なう.

整数に対する演算としては加減乗除の四則演算があり, ガードの条件として大小比較ができる. 浮動小数点数に対しても加減乗除の四則演算, ガードでの大小比較ができるが, これらは整数に対するものとは別の演算になっている. 通常, これらの演算, 比較は, 組み込み述語を直接用いるよりもマクロ記法を用いるのが便利である.

整数についてのガードでの大小比較には “<”, “>”, “=<”, “>=” を用いる. たとえば “X =< Y” は「X は Y と等しいか, または Y よりも小さい」を表す. 相等, 不等の条件には “=:=” と “=\=” を用いる.

たとえば第一引数と第二引数のどちらか大きい方を第三引数の値とするようなプログラムは, 以下のように書ける.

```
max(X, Y, Z) :- X >= Y | Z = X.
max(X, Y, Z) :- X =< Y | Z = Y.
```

このプログラムでは X と Y が等しい場合, どちらの節のガードも真である. 前述の通り, このような場合はどちらの節が選ばれるかは言語仕様としては規定しない. このプログラムではどちらの場合でも結果は変わらないので問題ないわけである.

整数の演算のためのマクロは通常の加減乗除の記号 (“+”, “-”, “*”, “/”) と括弧を用いて算術式を表記し, これを “:=” の右辺に, 結果を値にしたい変数を左辺に書く.[2]

たとえば第一引数と第二引数の和を第三引数の値とするような述語は以下のように定義できる.

```
sum(X, Y, Z) :- true | Z := X + Y.
```

ここでガードが “true” となっているが, ガードに現れる “true” は常に真であるような条件, つまり無条件を表す.

浮動小数点数についても同様のマクロがあるが, 浮動小数点数用のものは先頭に “$” を付けて区別する. 上述のふたつの述語 max と sum を浮動小数点用に書くとしたら, 以下のようになる.

```
max(X, Y, Z) :- X $>= Y | Z = X.
max(X, Y, Z) :- X $=< Y | Z = Y.

sum(X, Y, Z) :- true | Z $:= X + Y.
```

2.3 構造を持ったデータ

構造を持ったデータとは, 要素となるデータをなんらかの形で集めてできているデータのことである. KL1 で扱える構造を持つデータには以下のものがある.[3]

ベクタ: 任意の型のデータを要素として任意個持つような構造. 要素番号を用いて各要素にアクセスできる. 要素の型はひとつひとつ異なっていても良い.

ストリング: ある範囲の値の整数値を要素として任意個持つような構造. 要素として許される値の範囲の応じて 1, 8, 16, 32 ビットのストリングがある. 基本的にはベクタと同様のデータ構造だが, 要素の型と値の範囲が限定されているため, 基本操作やメモリ上での表現を効率化しやすい利点がある.

コンス: 任意の型のふたつの要素を持つ構造. コンスひとつでは二要素のベクタと変わらないのだが, 複数のコンスを組み合わせることによってリスト構造などを柔軟に表現できる.

各々の構造データについては後に詳述する.

2.3.1 ベクタ

ベクタは任意の型のデータを要素として任意個持つような構造である.

[2] この他にビットワイズの論理和, 論理積, 論理排他和, 反転などの演算も用意されているが, ここでは詳しく述べない.

[3] 他にも実行可能コードを表すデータ構造などがあるが, ここでは述べない.

ベクタの表記は, 中括弧対 (“{” と “}”) の間に要素をカンマで区切って並べて書く. たとえば, 三要素 0, 1, 2 を持つベクタは “{1, 2, 3}” と表す. ベクタの要素はどんな型でも良いのだから, もちろんまたベクタになっていても良く, たとえば “{a, b, {0, 1, 2}, d}” は四要素を持ち, 要素のひとつが三要素のベクタになっているものを表している.

ベクタの要素は 0 から順に番号づけする. たとえば “{a, b, {0, 1, 2}, d}” の要素番号 1 に対応する要素は “b” である.

最初の (要素番号 0 の) 要素が記号アトムであるようなベクタには, そのアトムの名前, 開括弧 (“(”) 最初のもの以外の要素をカンマで区切って並べたもの, そして閉括弧 (“)”) という特別な表記法が用意されている. たとえば前述の “{a, b, {0, 1, 2}, d}” は “a(b, {0, 1, 2}, d)” と書き表しても良い. これはさまざまな構造を区別したい時に, 構造に名前をつけて, その名前と詳細な内容で書き表すようなスタイルに便利である. このような構造を特に**ファンクタ** (functor) と呼ぶ. ベクタ構造をファンクタ構造とみなす場合には, 第 0 要素のアトムをファンクタ名, 他の要素を引数と呼ぶ. たとえば “a(b, {0, 1, 2}, d)” というファンクタは, ファンクタ名は a, 第一引数は b である.

2.3.2 ストリング

ストリングはある範囲の値の整数値を要素として任意個持つような構造である. 要素の整数として文字コードを並べ, 全体として文字列を表すような使い方が一般的である. こうした文字列は “"abc"” のように, 二重引用符の間に文字を並べて表記する. 現在の Multi-PSI などでの KL1 システムでは, 特に要素の値の範囲についての指定がなければ 0 から 65,535 (符号なし 16 ビットで表せる範囲) となり, 文字コードとしては JIS 16 ビットコードを用いる.

ストリングについても要素には 0 からの要素番号がつけられる. たとえば “"abc"” の要素番号 1 の要素は文字 “b” に対応する JIS 16 ビットコードの整数値 (十六進で 2662) である.

2.3.3 コンス

コンスは任意の型のふたつの要素を持つ構造である. ふたつの要素は Lisp に習って car, cdr と呼ぶ. 表記には鈎括弧対 (“[” と “]”) の間に car, cdr の要素を縦棒 (“|”) で区切って書き並べる. たとえば “[a|b]” は car がアトム a, cdr がアトム b であるようなコンスを意味する.

コンスそれ自体はふたつの要素だけを持つ構造データだが, これを組み合わせてさまざまなデータ構造を作れる. その代表がリスト構造である. コンスの car がひとつの要素, cdr がリストの残り, 要素がひとつもないリストはアトム “[]” 表すものと取り決めれば, ふたつの要素 a, b を持つリストは “[a | [b | []]]” と, ふたつのコンスを用いて表すことができる (図 2.1). このままではリストが長くなると多数の括弧が必要になるので, これを “[a, b]” のように書き表す.

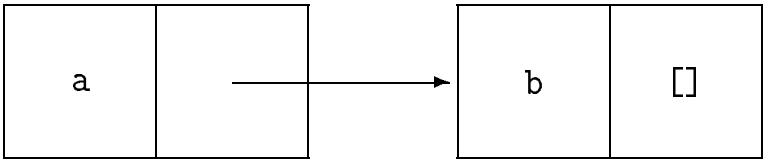

図 2.1: コンスによるリストの表現

2.3.4 不完全データ構造

ベクタやコンスのように任意の型の要素を持てる構造データの場合, 要素の値がまだ決まっていないことがあっても良い. このような構造を**不完全データ構造** (incomplete data structure) と呼ぶ.

不完全データ構造の値の決まっていない要素は値の決まっていない変数として書き表わす. たとえば "[a|X]" は car がアトム a, cdr が X という名前を持つ変数であるようなコンスを表す.

不完全データ構造の中の変数も, 構造体全体を持っているゴールと別のゴールで共有することができる. 変数の値を決めるのはこの別ゴールであっても良い. つまり, 後で誰かが詳細を決めることになっているデータ構造, ということになる. こうしたデータ構造は KL1 のプログラミング・テクニックのさまざまな局面で重要な働きをし, 柔軟なプログラミング・スタイルができる源となっている.

2.3.5 構造データのユニフィケーション

ここまでの例ではアトミックなデータのユニフィケーションしか出てこなかったが, 構造を持つデータについてもユニフィケーションはできる.

ガード・ユニフィケーション ガードでのユニフィケーションは両辺の値の一致を調べるものだが, 構造体同士の場合は以下のような再帰的な規則になる.

1. まず両辺が全体として同じ型の構造体で, 同じ個数の要素であることを確かめる. そうでなければ値は不一致である.

2. 次に両者の対応する (同じ要素番号の) 各要素について, この規則を適用する. すべてについて一致するときのみ, 全体が一致するものとする.

要素のユニフィケーションの際に片方が引数に渡されたもの[4]以外の変数なら, その変数にはもう一方の構造体の対応する要素が渡される. たとえば, 以下のプログラムは第 1 引数に渡されたコンスの car と cdr を, 第 2, 第 3 の引数に返すような述語の定義になっている.

```
carcdr(Cons, Car, Cdr) :- Cons = [X|Y] | Car = X, Cdr = Y.
```

[4]あるいは, この規則によって既に値を渡されたもの.

ボディ・ユニフィケーション　ボディに現れるユニフィケーションで一辺が変数の場合は, もう片方が構造を持っていてもいなくても同じで, その変数の値をもう一辺の値に決めるものである.

両方ともが構造体の場合のユニフィケーションの規則は, やはり以下のような再帰的なものなる.

1. まず両辺が全体として同じ型の構造体で, 同じ個数の要素であることを確かめる. そうでなければユニフィケーションは失敗である.

2. 次に両者の対応する (同じ要素番号の) 各要素について, この規則を適用する.

なお, 前述したように, 両辺とも値が決まっているようなボディ・ユニフィケーションを行なうのは, 推奨するコーディング・スタイルではない.

まとめ

KL1 で扱えるデータについて概説した. 中でも要素が何なのかまだ決まっていない不完全データ構造を扱えるのが大きな特徴である.

次章ではこの不完全データ構造を利用したプログラミング手法である, プロセスとストリームについて述べる.

第 3 章

プロセスとストリーム通信

前章までに KL1 の基本的な言語仕様を解説した. 本章では, KL1 で良く使われる**プロセス** (process) とプロセス間をつなぐ**ストリーム** (stream) による通信を用いるプログラミング・スタイルについて述べる.

なお, このプログラミング・スタイルは Shapiro と竹内によって最初に提案されたもので [4], 現在までに KL1 でオペレーティング・システムや種々の応用プログラムを比較的スムースに開発してこられたのはこのスタイルに負うところが大きい.

3.1 略記法

本題にはいる前に, いくつかの便利な略記法について述べる.

3.1.1 ガード・ユニフィケーションのヘッドでの表記

ガードで引数との間でユニフィケーションを行なう場合, ヘッドの対応引数位置に直接ユニフィケーションの相手を書いてしまうような略記ができる. たとえば, 前述の例にあった

```
not(In, Out) :- In = 1 | Out = 0.
```

という節は

```
not(1, Out) :- true | Out = 0.
```

と略記することができる.

この例の 1 のような具体的な値ではなくとも, 引数同士のユニフィケーション (引数の値が同じ) をガードで行なう場合, たとえば

```
same(X, Y, Ans) :- X = Y | Ans = same.
```

という節は

```
same(X, X, Ans) :- true | Ans = same.
```

と, 同じ変数名を複数回ヘッドに書くことによって表せる.

3.1.2 ゴール true の省略

ガードが無条件の場合, つまりガードが引数のない述語 true の呼び出しのみからなる場合, ガード部全体をボディとの区切りである縦棒ごと省略することができる. たとえば, 前述のインバータの例は

```
not(1, Out) :- Out = 0.
```

と書いても良い.

ガードだけでなく, ボディも true だけならば, さらにヘッドとの区切りである ":-" ごと省略して良い. たとえば:

```
one(1) :- true.
```

という節があったら

```
one(1).
```

と書いても良い.

3.2 プロセス

前述の通り, KL1 の基本実行機構は簡約化の (並列な) 繰り返しである. しかし, この概念だけではある以上程度複雑な計算をわかりやすく記述するためには不足で, 複数の簡約化操作をまとめあげるような概念が有用である. 本章ではそのような役割を果たす**プロセス**の概念について述べる.

3.2.1 KL1 のプロセスとは

ボディに同じ述語を呼び出すゴールをひとつ含むような節は, 同じことを (引数は変わるが) 繰り返し行なうループを表していると見ることができる. たとえば, 与えられた引数から始めて, 0 になるまでカウントダウンしていくような述語は

```
count_down(0).
count_down(N) :- N > 0 | M := N-1, count_down(M).
```

というふたつの節で定義できる.

【注意】 この例で "N := N-1" とはなっていないことには気をつけていただきたい. 前にも説明したが, KL1 の変数は値の起き場所を示すものではなく, むしろ値そのものにつけた名前なのである. だから N + 1 には M という, N とは異なる名前をつけているのである.

この例のボディには “M := N - 1”, つまり引き算を行なうゴールと, “count_down(M)” という再帰呼び出しのゴールのふたつがある. KL1 ではボディ中のゴールの記述順序には特に意味がないので, 両者は実行順はどうなるかわからない. 引き算よりも前に再帰呼び出しの実行を始めることもあるし, 両方同時に実行することもある. しかし, 述語 count_down を定義するふたつの節は両方ともガード (ないしはそれを略記したヘッド) で選択条件として引数の値を調べているので, 実行はその値が決まるまで待たされる. この引数の値は引き算の結果になっているので, 実際には必ず引き算の方が先に実行されることになる.

基本的な実行機構である簡約化のひとつひとつを見ると細切れの操作なのだが, この繰り返しを全体として見ると, ある程度の大きさを持った連続性のある計算過程であると考えることができる. このような計算過程を**プロセス**と呼ぶ.[1]以降にプロセスとみなせるようなまとまった計算を行なう述語の例をいくつかあげ, KL1 のプロセスとはどんな概念なのか, どういう特徴を持つかをもう少し詳しく説明していこう.

3.2.2 リスト要素の和

整数を要素とするリストに対して, 要素の総和を計算するような述語は以下のように書ける.

```
sum([], PSum, Sum) :- Sum = PSum.
sum([One|Rest], PSum, Sum) :-
    NewPSum := PSum + One,
    sum(Rest, NewPSum, Sum).

sum(List, Sum) :- sum(List, 0, Sum).
```

このプログラムはふたつの述語の定義からなる. 両者は同じ sum という名前だが, 引数個数が異なる. KL1 では同じ名前でも引数個数が異なれば違う述語として扱う. この例でもそうだが, 同じ名前で引数個数が異なる述語は補助的な述語の名前として使うことが多い.

最初のふたつの節で定義する三引数の述語が実際の計算をする述語である. この述語を単独に見ると, 第一引数に与えられるリストの要素すべてと, 第二引数に渡される数とを足し合わせ, 第三引数にその結果を返すものになっている. 最初の節は, 空のリストについては要素がないのだから第二引数をそのまま返せば良いことを表している. もうひとつの節は, リストが空でない場合, 最初の要素が One なら, この One と第二引数 PSum との和 NewPSum とリストの残り部分 Rest 要素との和が求める総和であることを表す.

最後の節で定義する二引数の述語がもともと定義したかった述語で, リストの要素の総和と 0 の和を計算すれば, リストの要素の総和そのものを計算することになる, という意味になる.

[1] KL1 ではプロセスという概念は言語仕様の一部ではなく, あるプログラミング・スタイルにつけた名前に過ぎないことに注意されたい.

注意されたいのは, 二番目の節でボディに書いてある足し算と再帰呼び出しのふたつは, 並列に動いてもかまわないということである. 再帰呼び出しの実行では節の選択条件は第一引数のリストが空かどうかだけに依っており, 足し算の結果は節の選択に関係しない. だから, 再帰呼び出しだけ先にどんどん実行してしまい, 足し算は後からやっても構わないのである.[2] ただし, 次々に呼ばれる足し算の引数のひとつ (PSum) は, 一段前の呼び出し時に呼んだ足し算の結果になっている. だから, 要素個数と同じ回数行なわれる足し算については実行順が自動的に決まってしまう.

このように KL1 でプロセスと呼び慣わしている計算過程は必ずしも逐次的な過程ではなく, それ自体の中に並列性を持っていることも少なくない. プロセスと考えるかどうかはまったく見方の問題に過ぎないのである.

3.2.3 自然数のリスト

ある数未満の自然数すべてを要素とするリストを作るような述語は以下のように書ける.

```
naturals(N, M, List) :- N >= M | List = [].
naturals(N, M, List) :- N < M |
    List = [N|Rest],
    N1 := N + 1,
    naturals(N, M, Rest).

naturals(M, List) :- naturals(0, M, List).
```

このプログラムも同様に二引数の本来定義したかった述語と, 補助的な役割を果たす三引数の述語からなっている.

最初のふたつの節で定義する述語は, 第一引数の値から始めて, 第二引数未満の整数すべてを小さい順に要素に持つリストを第三引数に返すものである. 最初の節は, 第一引数が第二引数より小さくなければ, 空のリストを返せば良い, という意味である. 次の節は, 第一引数の方が小さければ, 第一引数を先頭要素とするリストを返せば良く, リストの残り部分は第一引数よりひとつだけ大きい値から始めて第二引数未満の整数すべてを小さい順に要素に持つリストにすれば良い, という意味である.

最後の節で本来定義したかった述語を定義していて, 三引数の方の述語を 0 から始めるように呼べば, 与えられた引数未満の自然数すべてを要素にするリストが作れる, ということを書いてあるわけだ.

この例でも, 第二の節のボディのユニフィケーション, 足し算, 再帰呼び出しのみっつはどんな順で実行しても (並列に実行しても) 構わない.

[2] 並列に行なって良いということは必ずしも実際に並列に行なえば速くなるということではない. 通信のためのコストなどがあるので, 普通は足し算のような簡単な操作を並列に行なうメリットはない. 実際, KL1 の処理系ではこのような足し算を並列に行なおうとしたりはしない. ただし, この例は整数の足し算という簡単な操作だったのだが, 足し算の代わりにもっと複雑な操作を各要素について行なう必要がある場合には, 実際に並列に行なうメリットがでてくる場合もある.

3.3 ストリーム通信

本節では前節に述べたプロセスをモジュールとして結び合わせ, より複雑な計算を記述する方法について述べる.

3.3.1 プロセスの複合

ひとつのプロセスの計算結果を用いて, 別のプロセスが計算をするように, プロセスを組み合わせるプログラムを組むことができる. たとえば, 与えられた数未満の自然数の和を求めるには, 前掲のふたつのプロセスを組み合わせる以下のようなプログラムを使えば良い.

```
sum_up_to(N, Sum) :- naturals(N, List), sum(List, Sum).
```

このプログラムでは変数 List がふたつのプロセスから共有されており, naturals が作ったリストを sum に渡す手段になっている.

このふたつのプロセスは同じボディの中に書かれているのだから, どんな順序で実行しても良い. 二引数の述語 sum はガードの条件がないから, List の値が決まっていなくても実行でき, 三引数の方の述語 sum を呼び出す. こちらの方は引数の値によってどの節を選ぶか決めなければならないので, 値が決まるまで待たされるわけである.

さて, ここで, 先ほどの自然数のリストを作る述語の定義をもう一度見てみよう.

```
naturals(N, M, List) :- N >= M | List = [].
naturals(N, M, List) :- N < M |
    List = [N|Rest],
    N1 := N + 1,
    naturals(N, M, Rest).
```

前にも書いたように, 第二の節のボディのユニフィケーション, 足し算, 再帰呼び出しのみっつはどんな順でも実行できる. ユニフィケーションが他のゴールと並列に実行できるということは, まだ計算が終らないのに答を返せるということである. もちろん答は全部求まったわけではなく, 結果を返す引数とユニフィケーションしているコンス構造の要素である cdr は再帰呼び出しゴールの実行結果が入るまでは変数のままだから, 前述の不完全データ構造である. しかし, 計算結果が全体としてはコンスであることはこのユニフィケーションで決まってしまっている. また, その car, つまりリストとして見たときの最初の要素は, 最初の呼び出しの第一引数は 0 なのだから, もうそれに決まっている.

では総和を求める方のプロセスの本体である, 三引数の sum 定義をもう一度振り返ってみよう.

```
sum([], PSum, Sum) :- Sum = PSum.
sum([One|Rest], PSum, Sum) :-
    NewPSum := PSum + One,
    sum(Rest, NewPSum, Sum).
```

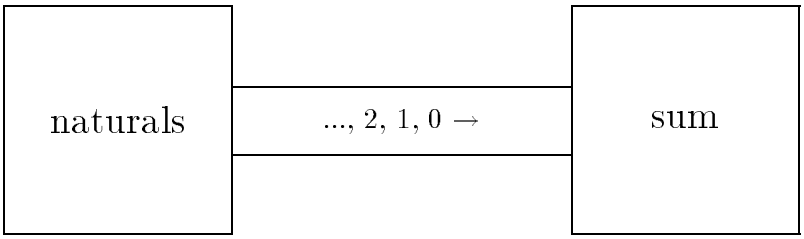

図 3.1: プロセス間の通信路

述語 naturals の結果がコンスであることまでもう決まっているとすると, ガードの条件 (実際にはヘッド中に略記されている) の判定はもうでき, ふたつめの節を選べる. そしてその car (ここでは変数 One で受けている) が 0 であることも決まっている. 第二引数の PSum は最初は 0 にして呼んでいるのだから, もう足し算の引数はふたつとも決まっているわけである. だから $0 + 0$ という足し算は, naturals の実行が少し進んだだけでもうすぐにでも実行できるようになっているわけだ.

再帰呼び出しの方はそうはいかない. まだ naturals が最初の一段階しか進んでいないとすると Rest の値は変数のままである. すると, 再帰的に呼び出された sum では第一引数がまだ決まっていないわけで, どの節を選べば良いかの判定ができない. この Rest は naturals と sum のふたつのプロセスの間で共有する変数になっており, naturals がその値を決めることになる. だから sum の再帰呼び出しの実行は naturals の方の計算がもっと進むまでは待たされる. この後に naturals の方がもう一段進めば sum の方ももう一段進めるようになる.

このように, 要素が不完全にしか決まっていない構造体を結果として返し, 後からその要素を確定していくことができる, という仕組は, KL1 の大きな特徴のひとつである. もし要素がすべて決まるまでは結果を返すことができないのだとしたら, naturals の実行が終るまで sum の実行を始めることができない. つまり, それだけ並列度が低くなってしまうわけである.[3]

3.3.2 ストリーム通信

前節に述べたふたつのプロセスは, 述語 naturals で表されるプロセスが次々に作り出す値を, 述語 sum で表されるプロセスが次々に使っていく, という関係にある. その仲介役を果たすのが要素が段々に具体的な値に決まっていくようなリストという不完全データ構造だった.

このリスト構造をデータが流れていくような通信路だと考えることもできる. この考え方では, プロセス naturals が次々にこの通信路に自然数を流して行き, プロセス sum はその通信路の出口からデータを読み取っては計算を進める (図 3.1). データがまだ到着していなければ読み取り側のプロセスは待たされる.

通信路を実現しているのはデータ構造コンスである. その car にはストリームを流

[3]繰り返しになるが, プログラムに並列性があるということと, その並列性を実際の並列実行として実現するかどうかは別の問題で, 実際に並列に実行するか逐次に実行するかは通信コストなどの要因を考えて決めるべきである.

れるデータが入り, cdr はストリームの続きを表す. 空のリスト [] は, 必要な通信が終りまで来てしまい, もうこれ以上メッセージがないことに対応する. このように通信路がデータ構造になっているので, データの流れる順序はどのようなデータ構造を作るかによって明確に決まってしまい, 計算の実行順序によってデータの順序が違ってしまうようなことは決して起きない. このような通信路を**ストリーム** (stream) という.[4]

ストリームを流れるデータはプロセス間で受け渡される**メッセージ** (message) であるとも考えられる. つまり, naturals というプロセスは sum というプロセスに次々に, この数も足してくれ, この数も, というメッセージを送っているわけである.

3.3.3 プロセスの状態

プロセスをメッセージを受けてそれに従って仕事をするように書けることを説明したが, もし仕事の内容が完全にメッセージだけで決まるのならわざわざそんな書き方をする必要はない. プロセス作ってそれにメッセージを送る代わりに, 必要な仕事をする述語を定義してそれを呼び出してしまえば良い.

メッセージ駆動のプロセスとして記述する意義は, プロセスには**状態** (state) を持たせることができるということにある. たとえば, 整数値を保持し, メッセージによってそれを増減するようなカウンタの機能を持つプロセスは, 以下のように書ける.

```
counter(Stream):- counter(Stream, 0).

counter([], Count).
counter([up|Stream], Count) :-
    NewCount := Count + 1,
    counter(Stream, NewCount).
counter([down|Stream], Count) :-
    NewCount := Count - 1,
    counter(Stream, NewCount).
```

このプログラムで定義するプロセスは, 第二引数としてその時々のカウンタの値を保持している. 初期値は 0 で, 後から up, down というメッセージが来るたびにそれを増減して再帰呼び出しの引数に渡すことによって, 状態を更新しているわけである. このように KL1 のプロセスは**状態を引数値として保持**する.

このプロセスを呼び出す時に

```
?- counter(Stream), some_other_process(Stream).
```

のようにして, メッセージ・ストリームを媒介して他のプロセスと通信できるようにするわけだが, このストリームがこのカウンタのプロセスとそれ以外のプロセスとを結ぶ唯一の通信手段である. 状態として保持しているカウンタの値を直接他のプロセ

[4] この例の場合は合計を求めるのが目的なので, データの到着順は問題ではないのだが, 一般には順序が問題になることが多い.

スが操作することはできない. つまり, プロセスの状態として値を保持することは**情報を隠蔽**していることになる.

では, この隠された情報にはどうやってアクセスしたら良いのだろう. 上のプログラムではメッセージを送ってカウンタを上下することはできるが, カウンタの値が何なのかを読むことはまったくできない. これについては次節で説明しよう.

3.3.4 複雑なメッセージ

これまでの例では, ストリームを流れるメッセージはすべてアトミックなデータだけだった. もちろんメッセージとしてもっと複雑な構造を持つデータを流しても構わない.

よく使われるメッセージとしてファンクタ構造がある. 前述のようにファンクタはベクタの一種だが, その要素番号 0 の要素がアトムで, 構造の種類を表しているようなものである. ファンクタをメッセージとして用いて, 要素 0 のアトムであるファンクタ名でメッセージの種類を, 他の要素 (ファンクタの引数) でメッセージの詳細を表すのが便利である.

前述のカウンタを一度にひとつずつだけではなく, いくつでも増減できるようにするには, 増減のためのメッセージを引数を持つファンクタにして, 以下のように書けば良い.

```
counter(Stream):- counter(Stream, 0).

counter([], Count).
counter([up(N)|Stream], Count) :-
    NewCount := Count + N,
    counter(Stream, NewCount).
counter([down(N)|Stream], Count) :-
    NewCount := Count - N,
    counter(Stream, NewCount).
```

カウンタの値を読み取る機能も, 以下のようなファンクタをメッセージとする節を追加すれば実現できる.

```
counter([show(Value)|Stream], Count) :-
    Value = Count,
    counter(Stream, NewCount).
```

値を読み取るためのメッセージ show は, カウンタの値を返してもらうための引数 Value を未定義のままで送る. 受けとったプロセスは内部状態に応じてその値を決めてやるわけである. このような未定義の部分を含んだメッセージを**不完全メッセージ** (incomplete message) という. 不完全メッセージも不完全データ構造の一種である. このようにすれば, 一本のストリームを介して双方向の通信ができるわけである.

まとめ

プロセスとストリームという概念を用いる KL1 のプログラミング・スタイルについて述べた．このプログラミング・スタイルの実現には不完全データ構造が主要な役割を果たしている．

次章ではこのプロセスとストリームを組み合わせて複雑なプログラムを組み上げていくためのさまざまな手法について述べる．

第 4 章

プロセス・ネットワーク

前章では KL1 のプログラミング・スタイルとしてのプロセスと, その間の通信に用いるメッセージ・ストリームについて述べた. 本章ではプロセスをストリームで結合して組み合わせてプロセスのネットワークを作り, 複雑なプログラムを組み上げていくためのさまざまな手法を紹介する.

4.1 フィルタ

与えられた数未満の自然数の和を計算するプログラムを前章で紹介した. これは自然数のリストを作るプロセスと, その要素の総和を計算するプロセスのふたつからなっていた.

では, 少し問題を変えて, 与えられた数未満の自然数の平方の総和を計算するプログラムを考えてみよう. 前章のプログラムを利用して作るとすると, すぐに思いつく方法は以下のふたつだろう.

- 自然数のリストを作るプロセスを直して, 自然数の平方のリストを作るようにする.
- リスト要素の総和を計算するプロセスを直して, リスト要素の平方の総和を求めるようにする.

これはいずれもふたつのプロセスのいずれかを改修して必要な機能を作ろうという方針である. もちろんこのどちらの方法でもプログラムは作れるし, この問題に限っていえばこれで十分である. しかし, もし両方のプロセスともっと複雑な仕様で簡単には改修できないとしたら, どのような方法があるだろう.

与えられた数未満の自然数のリストを作るプログラムはもうある. リスト要素の総和を求めるプログラムもある. とすると, 整数のリストをもらって, 各要素の平方を要素とするようなリストを作るプログラムをその間に入れてやれば解決である. たとえば以下のようなプログラムを作れば良い.

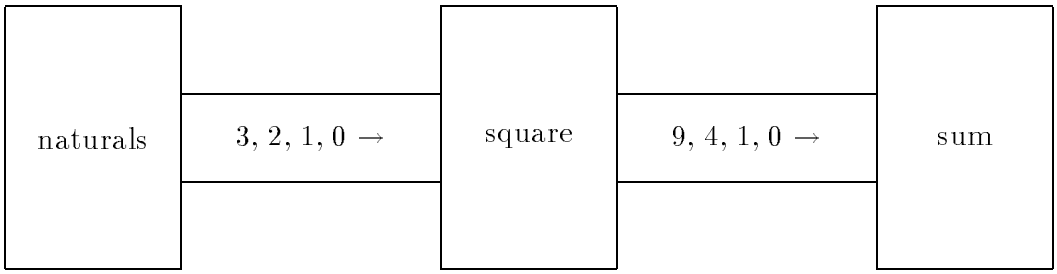

図 4.1: フィルタ

```
square([], Out) :- Out = [].
square([One|Rest], Out) :-
    Square := One * One,
    Out = [Square|OutTail],
    square(Rest, OutTail).
```

この述語を使えば, 全体のプログラムは以下のように書ける.

```
square_sum_up_to(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    sum(Squares, Sum).
```

述語 square はリストを入力としてリストを出力するような述語を定義している. この述語の計算過程をプロセスととらえ, 入出力のリストをストリームと解釈するとどうなるだろう. このような見方をすると, このプログラムは一本のストリームから整数値のメッセージを受け, もう一本のストリームにその平方をメッセージとして送り出すプロセスを表している. このように, あるストリームから受けとったメッセージになんらかの変換を施して, 別のストリームに出力するようなプロセスを**フィルタ** (filter) と呼ぶ. 全体のプログラムはプロセス naturals とプロセス sum が, フィルタ square を通るストリームを使って通信する, という構成になる (図 4.1).

この例ではフィルタとなったプロセス square は状態を持たず, 出力は入力メッセージだけに依存している. 一般にはフィルタの動作は入力メッセージだけではなく, フィルタ・プロセスの状態に依存しても良い. フィルタ・プロセスが入力メッセージに応じて状態を変えることもできるから, 入力の履歴に応じてフィルタの仕方を変えることもできる.

4.2 ストリームの連結

また少し問題を変えて, こんどは与えられた数未満の自然数の平方と立方の両方の総和を計算するプログラムを考えてみよう.

前節と同様にひとつのフィルタで済まそうとすると, 入力 n に対して $n^2 + n^3$ を出力とするようなフィルタを作るという方法はある. しかし, それではせっかく作った square 述語は使わずに, 別に定義しなければならない.[1] そこで, 前述の square はそのままにして, 別に入力メッセージの立方を出力するようなフィルタを作って, これを使うことを考える.

このようなフィルタ自体はどう定義すれば良いかはもうおわかりだろう.

```
cube([], Out) :- Out = [].
cube([One|Rest], Out) :-
    Cube := One * One * One,
    Out = [Cube|OutTail],
    cube(Rest, OutTail).
```

このふたつのフィルタと残りの naturals, sum をどうストリームで結合するかだが, 入力には両方ともプロセス naturals の出力を共通に与えれば良い. これはごく簡単で

```
square_sum_up_to(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    ...
```

のように同じ変数を両方に書けば良い. 問題はどうやって Square, Cubes の二本の出力ストリームに流れるメッセージが共にプロセス sum に渡るようにするかである.

ひとつの方法は, まず片方のストリームのメッセージを送り込み, それが終ったらもう一方のストリームのメッセージを送るようにすることである. そのための交通整理をする述語は以下のように書ける.

```
append([], In2, Out) :- Out = In2.
append([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    append(In1, In2, OutTail).
```

この述語は第一, 第二のふたつの引数が入力ストリームに, 第三引数が出力ストリームになっている. 全体としてはまず第一引数にやってくるメッセージを次々に出力し, それが終ったら第二引数の方のメッセージを流すようにしている.

最初の節は, 第一引数である一方のストリームが終りまで来たら, 後は第二引数であるもう一方のストリームをそのままままとめて出力ストリームとしてしまえば良い,

[1] 老婆心ながら付け加えると, ここには説明の都合上非常に簡単な例を使っているので, これぐらい簡単な計算で良いのなら全部書き直してしまった方が早い. 本当はここに説明するような方法は必要な計算がもっと複雑で, square のような述語を書き直す手間が大きい場合にこそ有効なのである.

という意味である．メッセージをひとつひとつ取り出しては中継する必要はないわけである．もうひとつの節は，第一引数のストリームの方にメッセージが来たら，それをそのまま出力することを意味する．[2]

この述語を使って全体のプログラムを書くと，以下のようになる．

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    append(Squares, Cubes, Both),
    sum(Both, Sum).
```

ひとつ注目して欲しいことは，この中の append の定義ではメッセージの中身をまったく見ていないという点である．単にメッセージが来たかどうかだけを見て，来たメッセージを中身を見ずに中継している．このため，どんなメッセージが流れるストリームに対しても同じ append を使うことができる．ここには KL1 の変数に型がない (どんな型のデータでも入れられる) ことの利点が現れているわけである．

4.3 マージャ

前節で紹介したやり方にはちょっと不満が残る．前節のやり方では，ストリーム Squares の出力が終るまではストリーム Cubes にながれるメッセージがプロセス sum にひとつも届かない．もし square の処理にかなり時間がかかるとすると，並列に動いている sum は暇になってしまう．すでに cube の方の処理がある程度進んでいれば，その出力もどんどん足し込んでいけるのに，メッセージが届かなくては何もできない．

そこで，片方のストリームが終りまで来なくても，もう片方のストリームの出力も中継してやれるようなやり方を考えよう．それには，どちらのストリームにでも良いからメッセージが来たら，それをどんどん出力に流していくようなプロセスを作れば良い．このようなことをする述語の定義は以下のようになる．

```
merge([], In2, Out) :- Out = In2.
merge(In1, [], Out) :- Out = In1.
merge([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    merge(In1, In2, OutTail).
merge(In1, [Msg|In2], Out) :-
    Out = [Msg|OutTail],
    merge(In1, In2, OutTail).
```

最初のふたつの節は，どちらか一方のストリームが終りまで来たら，もう一方のストリームをそのまま出力につないでしまえば良い，という意味である．残りのふたつの節

[2]引数をストリームとしてではなく単なるリストとして解釈すれば，この述語はふたつのリストをつなぎ合わせたようなリストを作る述語になっている．

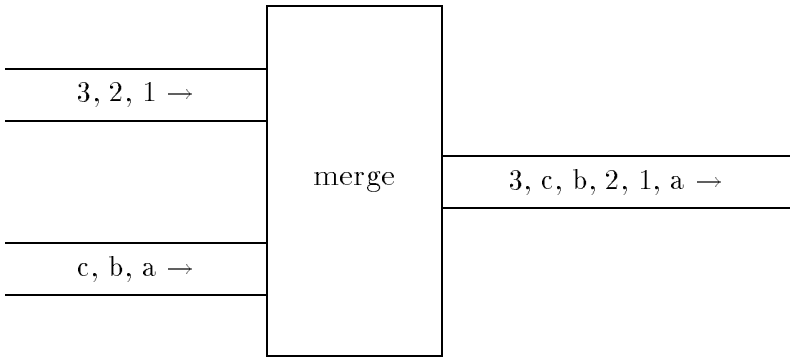

図 4.2: マージャ

が, どちらか一方にメッセージが来た時にそれを中継して出力する, ということをするためのものである.

このように複数のストリームからの入力をひとつのストリームにまとめるようなプロセスを**マージャ** (merger) という. 前節に述べた append もマージャの一種といえるが, まず片方のストリームへのメッセージを流し, それが終ってからもう一方のストリーム, という風になっているので, より制限が強い.

注目して欲しいことは, 両方のストリームともにメッセージが来ている時にこの述語 merge がどのような動作をするかである. この場合, 三番目, 四番目の節は両方とも適用条件を満たしている. このようなときに, どちらが選ばれるかは言語仕様としては決めていない. だから, 同じプログラムを同じ入力データで動かしても, マージャがどのように動くかによって流れていくメッセージの順番は毎回違うかも知れない (図 4.2). これは KL1 の特質のひとつである非決定性が現れている例である.[3] この非決定性はプログラムのデバッグには厄介な性質だが, この例のように並列性を上げるためには必要不可欠な場合がある. なお, 別々のストリームからのメッセージがどんな順で出力されるかは非決定的だが, もともと一本のストリームの中で前後関係があったメッセージは出力中でもその前後関係を保っている.

この述語を使えば, 全体のプログラムは以下のようになる.

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Sum).
```

これなら両方のストリームのどちらからでも, メッセージが来るごとにどんどんプロセス sum に送りつけることができ, 並列性の面で append を使ったプログラムよりも有利になっている. プロセスの全体像は図 4.3 に示すようになっている.

[3] もちろん前節の append のような非決定性のない書き方をすることもできる.

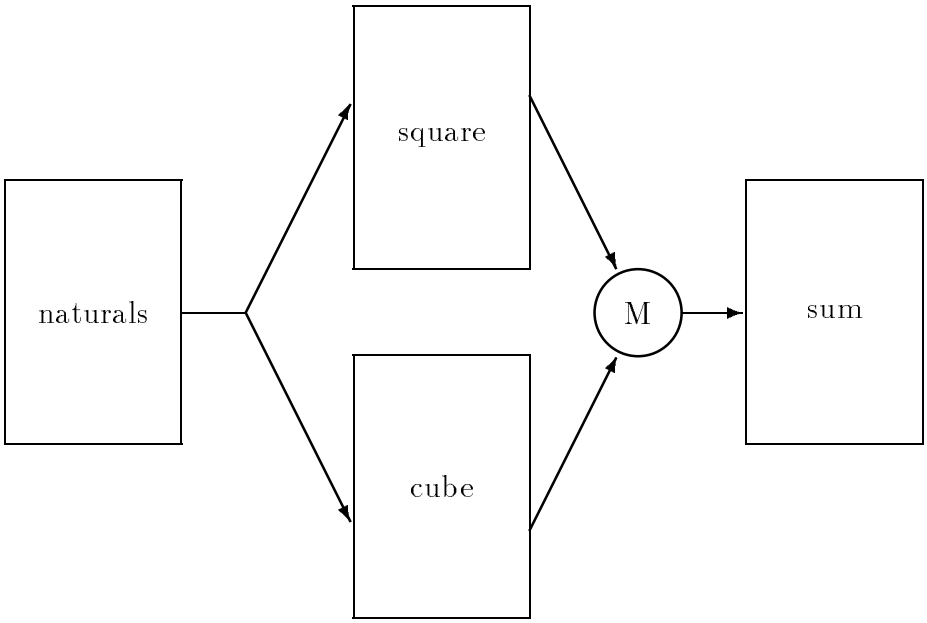

図 4.3: マージャを用いたネットワーク

4.4 組込みのマージャ機能

前節で述べたような KL1 で定義するマージャにはいくつかの問題がある．そのひとつは, 入力ストリームの数である．KL1 では任意引数個数の述語を定義することはできないので, 一定本数の入力を持つようなマージャしか定義できない．だからいくらでも多くの入力ストリームを持つようなマージャを作るためには, 入力ストリームが増えるたびに (あるいは何本か以上増えるたびに) 新しいマージャを入れていき, マージャの木構造を作っていかなければならない．また, マージャを通り抜ける時には木構造の深さ程度の数のプロセスを経由することになるので, 木構造のバランスが悪いとひどく効率の悪いプログラムになる．入力の増減に応じて構造を変えてバランスを保つようにすることはできるが, プログラムは複雑になる．

たとえ木構造をうまくバランスさせても, 入力が n 本のマージャを表すためには最低でも深さ $\log n$ の木構造を作らなければならず, マージャを通り抜けるための手間は $\log n$ に比例する程度になる．一方, 普通の手続き型の並列言語を考えると, ロック操作と破壊的代入の組み合わせで, 一定の手間のマージャを作ることができる．計算の手間のオーダが違うようでは決して効率の良いプログラムなど書けない．

この点を解決するために, KL1 では組込機能としてマージャを用意している．組込述語のマージャは二引数で “merge(In, Out)” のように呼び出す．このままでは第一引数のストリームにメッセージを流すと (つまり, 第一引数を car 要素にメッセージを持つようなコンス構造とユニファイすると) 第二引数に中継する (第二引数をそのメッセージを car に持つような別のコンス構造とユニファイする) だけである．マージャとして働き始めるのは, 第一引数をコンスではなく “$\{In_1, In_2, ..., In_n\}$” のようなベクタ構造とユニファイしたときである．こうするとマージャは n 入力のマージャとして働き始める．入力のひとつ “In_k” をさらにまた同様のベクタとユニファイすれば, いつで

も入力ストリーム数を増やせる. 入力ストリームを減らすには, いらなくなった入力ストリームを閉じれば (単に “[]” とユニファイすれば) よい. すべての入力ストリームが閉じられれば, 出力ストリームも閉じられる (“[]” になる).

組込みのマージャは入力ストリームの数によらず, 常に一定の手間でメッセージを中継できるし, その一定の手間も KL1 でマージャを書いた場合よりもはるかに小さい.

4.5 ディストリービュータ

平方と立方の総和を求めるプログラムでは, 平方を作るフィルタ, 立方を作るフィルタの両者に同じ入力メッセージを与えれば良かった. では, そうはいかない場合 — 入力メッセージの内, あるメッセージはひとつのフィルタを, 他のメッセージは別のフィルタを通したい場合はどうしたら良いだろう.

例として, こんどは与えられた数未満の自然数について, 偶数は平方し, 奇数は立法したものの総和を求めることを考えよう. それには, 入力を偶数か奇数かに応じて別のストリームに出力するようなプロセスを作れば良い. プログラムは以下のようになる.

```
dispatch([], Odds, Evens) :- Odds = [], Evens = [].
dispatch([One|Rest], Odds, Evens) :- One/2 =\= 0 |
    Odds = [One|OddsTail],
    dispatch(Rest, OddsTail, Evens).
dispatch([One|Rest], Odds, Evens) :- One/2 =:= 0 |
    Evens = [One|EvensTail],
    dispatch(Rest, Odds, EvensTail).
```

もうプログラムの細かい説明をするまでもあるまい. この述語で実現されるプロセスは, 入力ストリームが一本, 出力ストリームが二本あり, 入力メッセージの内容に応じてどちらのストリームに出力するか決めている. このようなプロセスを**ディスパッチャ** (dispatcher) と呼ぶ.

これを用いると, プログラムの全体は以下のようになる.

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    dispatch(Naturals, Odds, Evens),
    square(Evens, Squares),
    cube(Odds, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Sum).
```

プロセスの構成は図 4.4 に示すようになる.

4.6 サーバ

もうひとつの重要なプロセス・ネットワークの構成要素となるプロセスに, 複数のプロセスからアクセスされる共通のデータを貯めておく**サーバ** (server) がある. サー

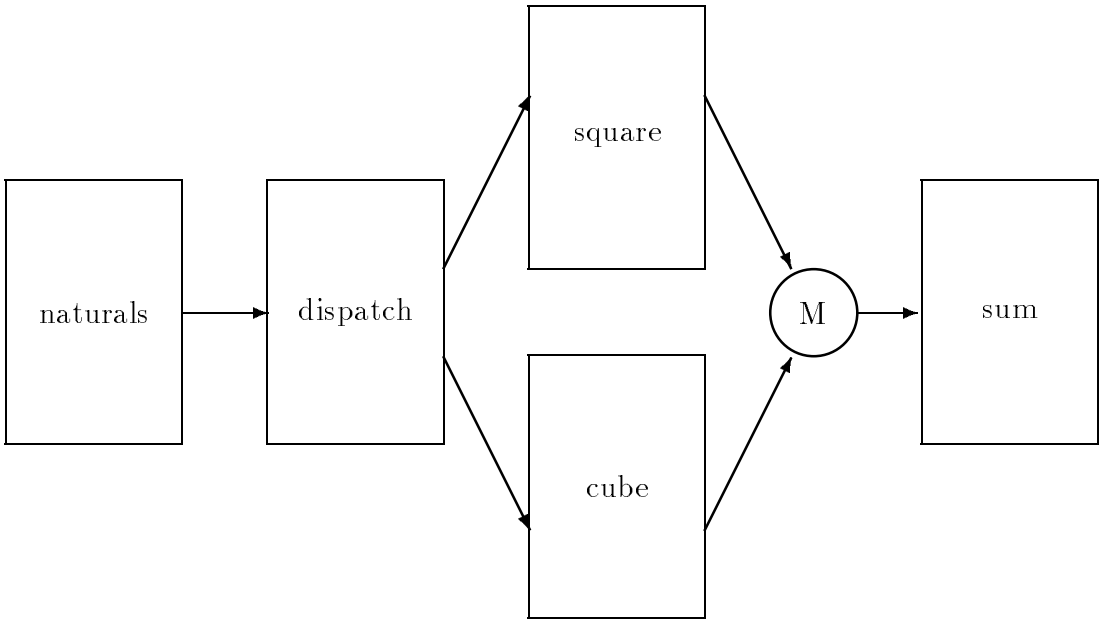

図 4.4: ディスパッチャを用いたネットワーク

バに対して, ストリーム経由でそこにアクセスするプロセスを**クライアント** (client) と呼ぶ.

既に示したカウンタはサーバの一例である.

まとめ

この章では KL1 のプロセスをストリームで結合していくさまざまな手法を紹介した. 実際のプログラムはこうした手法を組み合わせて構成していくことになる.

ここまで並列に動作できる KL1 プログラムをどのようにして組んでいくかについてはひと通り説明してきた. 次章では, ここまでに述べた並列実行の可能性を, 実際にどのようにして物理的な並列実行に結び付けていくかについて述べる.

第 5 章

実行の仕方の指定

前章までに KL1 の基本的な言語仕様と, 並列に動作できるプログラムを書くための手法を解説してきた. 本章では, 並列動作できるように書いたプログラムを実際に並列に動作させるための指定である**プラグマ**について述べる.

5.1 KL1 の並列実行指定の方針

本節では KL1 では並列実行指定についてどのような方針を取ったのか, それはなぜなのかについて述べる.

5.1.1 負荷分散はプログラムで指定する

実際の並列計算機に計算を効率良く行なわせるためには, ふたつのことを考えなくてはならない.

負荷の分散 行なわなければならない計算が一台のプロセサに集中してしまうと, そのプロセサの処理が終るまで計算が終らなくなってしまい, 並列に計算している意味がない. したがって, 全体の問題をいくつかの部分問題に分割し, 多くのプロセサに均等に分散して, どのプロセサにも同程度の負荷がかかるようにしなければならない.

通信の削減 全体の問題をいくつかの部分問題に分割して分散させるには, まず問題を配るための通信が必要で, 最後には結果を集めるための通信が必要になる. また, 効率の良いアルゴリズムを使うには部分問題が完全に独立にできず, 計算途中でも通信が必要になることも多い. 通信が多くなるとそのコストのために返って効率が落ちてしまうこともある. したがって, なるべく部分問題を解くプロセサ間の通信が少なくなるような問題の分割が重要である.

この負荷分散と通信削減は二律排反関係にあり, あまり分散し過ぎると通信が多くなり過ぎ, 通信を減らそうとすると分散できないということになる. この問題が起きないような問題の分割法を見つけ, また, 両者の適当なトレードオフを見つけることは, 並列処理ソフトウェアの研究の最重要課題といえよう.

特に問題の部分問題への分割の仕方は, 解くべき問題が何なのか, どのようなアルゴリズムを使うのかに大きく依存する. 現在のソフトウェア技術では, 部分問題への分割とその分散についての指針を明記していないようなプログラムが与えられても, そ

れに対して自動的に効率的な負荷分散を行なうことはまず無理である．もちろん，たとえば行列のかけ算を多数行なうことなど，あらかじめ行なうべき計算の量を予測しやすい問題領域に限定すれば，かなり効率的な並列化を行なうこともできる．しかし，いわゆる知識情報処理のような，計算結果によって次に行なうべき計算が動的に大きく変わることがあたりまえの分野では，自動負荷分散は非常に難しい．

現在の技術がこういう水準にあることから，KL1 はこの困難な自動負荷分散の方式を研究するためのツールとして役に立てるためのものと位置付けている．したがって，負荷分散は言語処理系で自動的には行なわず，プログラマが指定するものとした．ただし，プログラマがさまざまな負荷分散指定を行なうときに，それができるだけ簡単にできるように工夫している．それについては次節に述べる．

5.1.2 ふたつの並列性の分離

KL1 では二種類の並列性を別々に指定するものとしている．二種類の並列性とは，以下のようなものである．

論理的並列性 プログラムのどの部分が並列に動作しても良いかを指定するもの．KL1 ではデータの流れによって指定する．この並列性はプログラムの正しさに関わるもので，指定が誤っていればプログラムは正しく動作しないかも知れない．KL1 ガードで節の選択条件を指定すると，その選択条件が判定できるまで実行を自動的に送らせることによって，並列に動作して良いかどうかが暗黙に指定される．

物理的並列性 論理的には並列に動作して良い部分の内，どれを実際に並列に動かすかを指定する．この指定は**プラグマ** (pragma) と呼ばれる機能を用いて指定する．この指定による並列動作は，もともと並列動作して良いものの内から指定するのだから，プログラムの正しさには関わらないが，プログラムの実行効率には大きく影響する．

このように論理的並列性と物理的並列性の指定を分離することにより，プログラムの正しさに影響を与えることなく負荷分散の仕方を変えることができるようになる．分散方式の研究を行なうためにはさまざまな分散方式を実験する必要が生じるが，その際にただでさえ困難な並列プログラムのデバッグを毎回改めてやり直す必要がないことは，実験の効率を著しく高めるだろうと考えたわけである．

なお，プラグマはプログラムの正しさとは分離された効率のためだけの指定なので，処理系はこれに従わない方が効率が良いと判断すると必ずしも従わない場合もある．

以下，本章ではこのプラグマの指定方法について概説する．

5.2 ゴール分散プラグマ

計算の分散にはノード指定プラグマを用いる．指定には“*Goal*`@node(`*Node*`)`”のような形式でボディ・ゴールにプラグマを付加する．ここで指定する**ノード** (node) とは，個別のプロセサ，または内部では自動負荷分散するようなプロセサの集まりである．現在のところ，指定にはノードについた一連番号を用いている．

ゴール分散を用いる具体例をあげると，たとえば以下のようになる．

```
p([One|Rest], N, State) :-
    q(One, State, NewState)@node(N),
    N1 := N + 1,
    p(Rest, N1, NewState).
```

この例では, 次々に到着するメッセージについての処理を順番に各ノードに分散しているわけである. 何も指定のないゴール (この場合なら足し算と再帰呼び出しのゴール) は, 元のゴールを実行したのと同じノードで実行する.

5.3 ゴール優先度指定プラグマ

並列に実行できるゴールが複数あり, プロセサの数がそれよりも少なかったなら, どのゴールから順に実行するかが効率に大きく影響する場合が少なくない. そこで, どのゴールを先に実行した方が効率上有利かの示唆を与えるためのプラグマがゴール優先度指定プラグマである. 指定はボディ・ゴールに "*Goal*`@priority(`*Priority*`)`" のようなプラグマを付加して行なう. この *Priority* が具体的な優先度を指定する部分だが, 本稿では詳細は複雑になるので述べない.

なお, 優先度指定プラグマは必ずしも守られるとは限らない. 必ず守ろうとすると効率的な実装が難しくなり, 返って効率が低下する原因となるからである. また, 優先度の指定はひとつのノード内でだけ有効である.

5.4 節優先度指定プラグマ

複数の節の選択条件が真である場合, どの節を選ぶかは言語仕様としては定めていない. どれを選んでも正しく動くようにプログラムを書くべきであることは前に述べた. しかし, どの節を選ぶかによって実行効率に影響を与える場合もある. そこで, どちらの節も選べる場合に, どの節を選ぶと有利かについての示唆を与えるためのプラグマが節優先度指定プラグマである. 節の優先度指定には, まず優先して欲しい節 (複数でも良い) を先に書き, 他の節との間に "`alternatively.`" という節のようなものを書く.

節の優先度も必ずしも守られるとは限らない. やはり必ず守ろうとすると効率的な実装が難しくなるからである.

まとめ

本章ではプログラムを実際に並列に動作させるための指定である**プラグマ**について述べた. プラグマはあくまで処理系に対する効率的実行のための示唆に過ぎず, 処理系の動きを完全に指定するものではないことを再度記しておく.

おわりに

KL1 の言語仕様とプログラミング技法について概説した．KL1 とはどんな言語なのか, どのようにプログラムを書けるのかについて, おおまかな感じだけでもつかめていただけたら幸いである．

KL1 の言語仕様もプログラミング技法も日進月歩であり, 本稿ではまだ書き足りない点が多々ある．今後機会を見てこのチュートリアルにも改訂を加えていきたい．また, 読者諸氏の中からこの日進月歩の活動に加わっていただける方があるようなら, 筆者の大きな喜びとするところである．

なお, 本稿に述べた言語仕様やプログラミング技法は, 第五世代計算機プロジェクトに関わる数多くの研究者の共同の産物である．ここに至るまでの諸氏の多大な努力に感謝し, 今後の研究の一層の進展を願って, キーボードの叩き納めとしよう．

1991 年 6 月
近山 隆

参考文献

[1] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pages 230–251, Tokyo, Japan, 1988.

[2] Keith L. Clark and Steve Gregory. Parlog: A parallel logic programming language. *ACM Transaction on Programming Languages and Systems*, 8(1), 1986.

[3] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[4] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in New Generation Computing, Springer-Verlag Vol.1 No.1,1983.

[5] Kazunori Ueda. Guarded Horn Clauses. ICOT Technical Report TR-103, ICOT, 1985.

[6] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, December 1990.