社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
COMP96-58, SS96-35(1997-01)

擬データと関数による並行プロセス群の記述

石井 裕一郎　　武市 正人

東京大学 大学院 工学系研究科 情報工学専攻

〒113　東京都 文京区 本郷 7 の 3 の 1　東京大学 工学部 計数工学科 武市研究室

電話　03-3812-2111　(内 7412)

電子メール　yishii@ipl.t.u-tokyo.ac.jp　takeichi@ipl.t.u-tokyo.ac.jp

あらまし　遅延評価 関数型言語では、プロセスを決定的な関数として表現する手法がよく用いられ、この際入出力の処理は関数の外側にある OS 核が担当する。しかしこの手法では、非決定的処理を表現することも複数のプロセスからの要求が到着した順に処理をするサーバや OS 核そのものの記述もできない。そこで我々は、関数型言語に一般的な単一代入ができる擬データを導入し、関数として表現されたプロセスが相互に通信する並行プロセス群全体をも関数として取り扱う機構を提案し、その処理系 gofjava を作成した。コンソール入出力やアプレット等の GUI もこの枠組で扱うことができる。本稿では、このような機構による並行プロセス群の記述とその実現法を述べる。

キーワード　関数型言語,　擬データ,　並行プロセス,　非決定的処理,　入出力,　OS,　GUI

Describing a Group of Concurrent Processes with Pseudo Data and Functions

Yuichiro ISHII　　Masato TAKEICHI

Department of Information Engineering, Graduate School of Engineering, University of Tokyo

Takeichi Laboratory, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, Bunkyo-ku, Tokyo 113, Japan

tel:　03-3812-2111　(ex. 7412)

e-mail: yishii@ipl.t.u-tokyo.ac.jp　takeichi@ipl.t.u-tokyo.ac.jp

Abstract　In lazy functional languages, processes are often represented by functions where input and output are handled by an operating system kernel which is outside of the functions themselves. However this cannot deal with nondeterministic programs, such as a server or an operating system. In this paper, we introduce functional programming with pseudo data, where the whole group of concurrent processes, which communicate with each other, is described by a combination of functions. Practically, we have made the gofjava system which can compile our functional programs to JAVA code. We shall discuss how console input/output, GUI like applets, a server, and an operating system are described and implemented as concurrent functional processes.

key words　functional language, pseudo data, concurrent processes, nondeterminism, input and output, OS, GUI

1 序論

関数でプロセスを表現すると、プロセスの動作証明や最適化が容易であるという利点がある。遅延評価 関数プログラミングでは、受信メッセージを引数、送信メッセージを返り値とする関数でプロセスを表現する [9, 13, 4, 10]。これを一般化して、複数のプロセスが相互に通信するプロセス群についても定型的に動作の説明ができる枠組が求められる。

我々は、これらの記述のために並行して動作するプロセスの処理を分析し、(1) 待ち合わせ (2) 並行計算の起動 (3) 非決定的処理の 3 種が基本処理であるとの結論を得た。そこで、プロセスを関数で表現する際にこの基本処理をそれぞれ (1) 遅延評価と引数部や case 式のパターン照合 (2) 関数 par や spec [17, 19] (3) 擬データ処理 [24, 25] という手法で記述することとした。従来の関数プログラミングによるプロセス表現との違いは、擬データ処理の導入にある。

我々は、すでに非決定的処理を記述する擬データ関数プログラミングを提案し [24, 25]、擬データを扱うことのできる関数プログラミング処理系 gofjava を開発した。gofjava は関数プログラムを JAVA へ翻訳するもので、並行処理や GUI の実験用の処理系である。これを用いると、例えばアプレットを関数プログラムで宣言的に記述し、前述の関数プログラムの利点を生かした応用プログラムの開発が可能である。

本稿では擬データを用いて並行プロセスを関数で表現する新しい手法について述べ、本手法に基いてプロセス群や OS を記述する。さらに、入出力処理やアプレットの記述も本手法の枠組で把握できることを示す。最後に、今後の課題について述べる。

2 プロセスの関数表現

関数型言語には、動作の証明が数学的にでき [1, 20]、さまざまな最適化技法が利用できる [22, 3, 8] という利点がある。「同じ文脈の同じ式は同じ値を意味する」からである。これは参照透明性と密接な関係がある。

プロセスを関数で表現した場合には、上記の利点が保持され [10, 13, 4, 21, 9]、手続き的なプロセス記述では得られない利点が享受できる。しかし、この場合、プロセス側で非決定的処理は表現しない。入出力等の副作用が必要な場合は、プロセスの外部にある OS に非決定的処理を委ね、OS との通信によって行うこととするのが従来の考え方である。

例えば Dialogue モデルでは OS への要求を返り値とし OS からの応答を引数とする [Response] → [Request] 型の関数でプロセスを表現する。OS からの応答は時と場合によって異なるが、これを引数側に置くことによりプログラム上は非決定性を隠すことができる。

プロセスと OS、又はプロセス同士が 1 対 1 の通信をする場合には上述のように 1 引数の関数として表現すればよい。これを一般化すれば、外部の複数のプロセスと通信をする場合には、一般に $a_1 \to \cdots \to a_n \to (b_1, \cdots, b_m)$ という型を持つようにすればよいと推察される。これは他のプロセスとの間で n 種のメッセージ $a_1, \cdots, a_n$ を受け取り、m 種のメッセージ $b_1, \cdots, b_m$ を送るものである。

例えば 2 つのプロセス double と treble の両方から整数を受け取り、その和を double と treble に送信するプロセス adder がある場合、これら 3 つからなるプロセス群 dta は以下のように表現できる (図 1)。

```
> adder :: [Int] -> [Int] -> [Int]
> adder (n:ns) (m:ms) = (n+m):adder ns ms
> adder _      _      = []
>
> double, treble :: [Int] -> [Int]
> double xs = map (2*) xs
> treble ys = map (3*) ys
>
> dta = let ns = double as
>           ms = treble as
>           as = 1:adder ns ms
>       in as
```

adder では 2 つの引数のパターン照合で待ち合わせが表現され、2 つのメッセージ n と m の両方が到着して初めてメッセージ n+m を送信することができる。

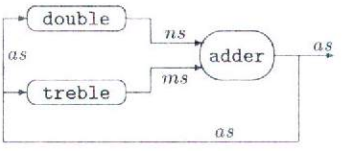

図 1: プロセス群 dta

この類推で ns から来た整数は 1 増やし、ms から来た整数は 2 増やして、到着順に返すプロセス increment を作ればプロセス群 dti は以下のようになる。

```
> increment (n:ns) (m:ms)
>   = (n+1):(m+2):increment ns ms
> increment ns     []     = map (1+) ns
> increment []     ms     = map (2+) ms
>
> dti = let ns = double as
>           ms = treble as
>           as = 1:increment ns ms
>       in as
```

上記 dti では double、treble ともメッセージを絶えず送信するので問題は発生しない。しかし、increment に対してメッセージを送る一方のプロセスが無限ループに陥った場合、プロセス群全体が全く送信をしなくなるという問題がある。このような従来の関数型言語の枠組では、キーボードの押下又はマウスのクリックで動作の種類を切り替えるプロセス ([KEY] → [CLICK] → ···) すらも表現できない。

関数型言語に並列論理型言語 [18, 26] で用いられるような非決定的パターン照合を導入すればこの問題は回避できる [13]。しかし、この手法では、「同じ文脈の同じ式は同じ値を意味する」という性質が失われ、関数型言語の利点が十分に生かされなくなってしまう。

一方、我々が提案した擬データ関数プログラミングでは、上記性質を維持しつつ、従来の関数プログラミングの枠組に極めて近い形で非決定的処理を表現できる [24, 25]。

上記のプロセスに相当するものは、以下の increment' のように表現すれば無限ループやユーザ入力待ちにも対応できるようになる。

```
> increment' ms ns = map inc . mergeR ms ns
>   where inc (A n) = n+1
>         inc (B m) = m+2
```

mergeR は非決定的にリスト ms と ns をマージする。ms と ns のいずれから値を取得したかという記録が隠された擬データ引数に記録されるため、「同じ文脈の同じ式は同じ値を意味する」性質が維持されるのである。

また、従来の関数プログラミングでは OS そのものの記述もできない。入出力という非決定的処理を伴うからである。しかし、擬データ関数プログラミングでは、上記の mergeR と同様、入出力の履歴を擬データに保存するという手法でこれを解決しており、OS そのものを表記することも可能である。

次節では、この擬データ関数プログラミングについて詳述する。

3 擬データ関数プログラミング

3.1 擬データの確定と未確定

擬データは神託 (Oracle) [2, 11] や Dialogue モデルの応答リスト [9, 6, 7] を一般化したものである。擬データに対しては一般的な単一代入ができる。擬データは、内部的には未確定 (未代入に相当) と確定 (代入済に相当) の 2 つの状態を持つが、プログラムからは この判別はできない。

a 型の値を確定する擬データの型は $\ll a \gg$ である。型によって、従来の関数プログラミング以上の機能を用いる場所が明確に表わされる。

確定した擬データの値を取得するには後置演算子 ·? :: $\ll a \gg \to a$ を用いる。未確定の擬データに対して演算子 ·? を適用すると確定するまでブロックするが、プログラムからは計算に時間がかかることと区別できない。

擬データに対して対やリストを確定する場合には、以下のように特殊なパターン<< ? >> を用いる。

```
> mapR :: (<<a>> -> b) -> <<[a]>> -> [b]
> mapR f <<ra?:ras?>> = f ra:mapR f ras
> mapR f <<[]>>       = []
>
> connectR :: (<<x>> -> m) -> (m -> <<y>> -> n)
>          -> <<(x,y)>> -> n
> (f `connectR` g) <<(rx?,ry?)>> = g (f rx) ry
>
> seqsR :: [<<a>> -> ()] -> <<[a]>> -> ()
> seqsR (f:fs) <<ra?:ras?>>
>   = case f ra of () -> seqsR fs ras
> seqsR []     _            = ()
```

<<·>> :: $a \to \ll a \gg$ は構成子関数に相当し、演算子 ·? の逆関数である。上記パターン照合でのみ用いることができ、通常の式の中では使用できない。

式 mapR f ros 等、未確定の擬データ ros :: $\ll [a] \gg$ に対して<<ra?:ras?>> のパターン照合を行うと (図 2)

1. 未確定の擬データ ra :: $\ll a \gg$ と ras :: $\ll [a] \gg$ が生成される。
2. 擬データ ros を<<ra?:ras?>> に確定させる。ros? を評価すると ra?:ras? になる。

擬データ ros が確定している場合は、通常のパターン照合と同様に処理される。この点で、複数回の代入が行なわれた場合にエラーとなる I-structure[15, 16] と異なる。擬データは、並列論理型言語 [26, 18] 等の論理変数 (logical variable) に類似している。

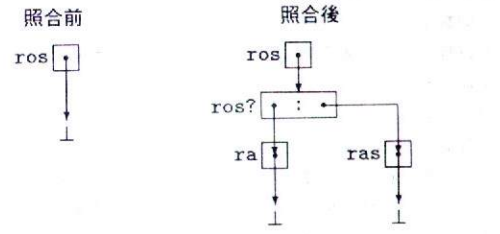

図 2: 未確定の擬データ ros のパターン照合

3.2 擬データと入出力

擬データでは副作用を伴う入出力も表現できる。例えば以下のようなコンソール入出力関数が用意されている。

```
> primitive putchar "primPutchar"
>                   :: Char -> <<()>> -> ()
> primitive getchar "primGetchar"
>                   :: <<Int>> -> Int
```

式 getchar ri は、擬データ ri が未確定ならばコンソールから 1 文字読み込んでその文字コードを ri に確定させ、その文字コードを返す。式 putchar c ru は、文字 c を評価し、擬データ ru が未確定ならばコンソールへその文字を表示して ru を () に確定させ、() を返す。擬データ ri や ru が確定している場合は、副作用は実行されず、それぞれ以前の入出力の履歴である ri?、ru? を返す。

3.3 擬データと非決定的処理

入出力が一度発生すると、その処理を取り消すことはできない。これは、未確定の擬データが確定するともはや別の値に確定することはありえないことに対応している。

入出力のほかによく用いられる非決定的処理は、2 つの式 x と y を並行計算して、結果を同じ擬データ r に競争して書き込む処理である。これを行う関数 race は以下のように定義できる。なお、並行計算は関数 par 等によって起動され、式 a `par` b は、a の計算を並行に起動しつつ b の値を返すものである。

```
> race x y ra = assign (strict A x) ra `par`
>               assign (strict B y) ra `par` ra?
>
> choice o x y = detag (race x y o)
>   where detag (A x) = x
>         detag (B y) = y
```

式 assign ra a は、a の値を計算してから擬データ ra への確定を試み、確定 (パターン照合) に成功すれば True、失敗すれば False を返す関数である [25]。

raceでは、タグA又はBを書き込んで排他制御を行いつつ、いずれを選択したかを表示する。例えば式race 1 2 raの結果はA 1かB 2のいずれかである。上では、McCarthyのambに類似するBurtonのchoice関数[2]も定義している。amb x y == amb x yは成立しないことがあるが、choice o x y == choice o x yは神託(非決定的選択の履歴)oを共有するので常に成立する。

3.4 擬データの性質

関数main :: ≪a≫ → ()を実行する際には、初めに1つだけ種となる未確定の擬データra :: ≪a≫が暗黙のうちに生成され、式main raが評価される。評価は遅延評価に基いて行われる。

評価の途中で新たに生成された擬データはすべて初めに1つだけ生成された種の擬データraから到達することができる。逆に、種の擬データra以外の未確定の擬データを生成するにはパターン照合を用いる以外に方法はない。生成された擬データはすべて種の擬データraに埋め込まれる。

擬データ パターン照合を、通常のパターン照合と同様に扱えば、種の擬データraを神託(Oracle)として解釈することもできる。対話的プログラムに対して⊥を含む引数を与えることによってその振舞いを証明する手法[1, 20]があるが、上記解釈の下ではこの手法がそのまま利用できる。

擬データ関数プログラミングでは、生成される擬データがすべて種の擬データに埋め込まれ、非決定的処理の履歴がすべて擬データに保存されるので「同じ文脈の同じ式は同じ値を意味する」という性質が保持される。また、擬データを表現するために新たに導入した要素が、パターン照合における構築子<<·>>の逆関数·?のみなので、従来の関数プログラミングとの親和性が高い。例えば近年注目されているdeforestation、fusion、hylomorphism[22, 3, 8]等のプログラム変換がそのまま適用できる。

3.5 プログラミング例

前記の高階関数を利用すれば、遅延文字列読込gets、文字列表示putsは以下のように定義できる。

```
> gets = map chr . takeWhile (-1/=) . mapR getchar
> puts = seqsR . map putchar
```

getsでは以下のように処理が進む。

1. mapRがgetcharを繰り返し起動(する予約を)し、入力された文字コードをリストにして返す。
2. takeWhileが、リストの先頭から文字コード-1(読込終了)が検出されるまでを返す。
3. map chrが整数リストを文字列に変換する。

擬データ関数プログラミングでは、遅延評価を基礎とする。上記の処理は、結果の文字列が必要となるまで起動されない。また、文字列のうち必要な部分までの入力がされれば、それ以降の文字を読もうとしてブロックすることはない。このような遅延入力は、従来の関数プログラミングにおける継続モデル、Dialogueモデル、monad[9, 14, 23]等では表現できない。一方、例えば式puts "hello!"では

1. map putcharが以下のリストを作成する。

```
[putchar 'h', putchar 'e', putchar 'l',
 putchar 'l', putchar 'o', putchar '!']
```

2. seqsRがリスト内の処理を順に実行する。

以下のechoはキーボード入力をそのまま画面に表示するプログラムである。

```
> echo  = gets `connectR` \ cs -> puts cs
> echo' = gets `connectR` puts
```

connectRは2つの擬データ処理関数を結合する。getsの結果の文字列はcsに接続(connectR)され、putsに渡される。echoではラムダ式\ cs -> ...を用いているが、echo'のようにconnectRでcsを明示的に用いずに関数を接続することもできる。

関数プログラミングでは高階関数を用いて処理を抽象的にモジュール化する手法が用いられるが、その手法が擬データに対しても応用できる。また、一般のプログラマはライブラリとして用意された関数を用いることにより特殊なパターン<< ? >>を使わずにプログラムが作成できる。

4 プロセスの表現

4.1 プロセス型

一般に副作用を伴うプロセスの型は以下のようになる。

$$a_1 \to \cdots \to a_n \to \ll s \gg \to (b_1, \cdots, b_m)$$

$a_1, \cdots, a_n$は他のプロセスから受け取るメッセージに、$b_1, \cdots, b_m$は他のプロセスへ送り出すメッセージに相当する。プロセスが行う副作用の記録が≪s≫に残される。新たな擬データは既存の擬データへの埋め込みとして生成されるので、≪···≫型の引数は1つでよい。

4.2 プロセス群の表現

上記のechoは2プロセス系であって、プロセスgetsからプロセスputsへメッセージcsが送られていると見ることができる。これらの型に注目すると、

```
gets  ::  ≪[Int]≫ → [Char]
puts  ::  [Char] → ≪[()]≫ → ()
echo  ::  ≪([Int],[()])≫ → ()
```

のように、いずれも上記のプロセス型に一致することがわかる。すなわち、複数のプロセス(gets、puts)をconnectRで結合したプロセス群全体(echo)を1つのプロセスとして表現していることになる(図3)。

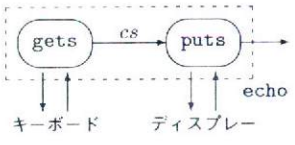

図 3: getsとputsの結合

4.3 リストの非決定的マージ

複数のプロセスと応答するサーバでは、リストの非決定的マージ処理が不可欠である。2つのリストを非決定的にマージする関数mergeRは、以上の機能を用いて定義することができる[25]。リストasとリストbsが以下のような要素を持つ場合に、

as = $[a_1, a_2, a_3, a_4, \ldots]$
bs = $[b_1, b_2, b_3, b_4, \ldots]$

以下の式では、非決定的に as と bs をマージして

```
mergeR as bs `connectR` \ cs -> ...
```

例えば以下のようなリストが cs に束縛される。

[A a_1,A a_2,B b_1,A a_3,B b_2,A a_4,B b_3,...]

A と B は、引数のいずれから要素を取得したかを示す。

mergeR では、上記 race 同様単一代入関数 assign を用いて書込みの排他制御を行っている。従来、リストの非決定的マージは組込関数として用意されることが多く、並行計算の起動と非決定的選択の 2 つの処理の分割が不十分であったが [10, 13]、擬データ関数プログラミングではこれらの役割が整理されているため、mergeR を組込ではなくユーザ定義関数として自然に定義できる。

5 応用例

5.1 アプレット

gofjava では、擬データ関数プログラミングにより GUI の記述、アプレットの作成ができる (図 4)。

GUI の記述にはイベント指向のプログラミング手法をとることが多い。例えば言語 JAVA[5] もその一つである。この手法では、あるイベント専用の処理関数 (イベント ハンドラ) を登録することにより、プログラムを記述する。したがって、コンソール入出力関数とイベント ハンドラとは、まったく異なる概念といってよい。

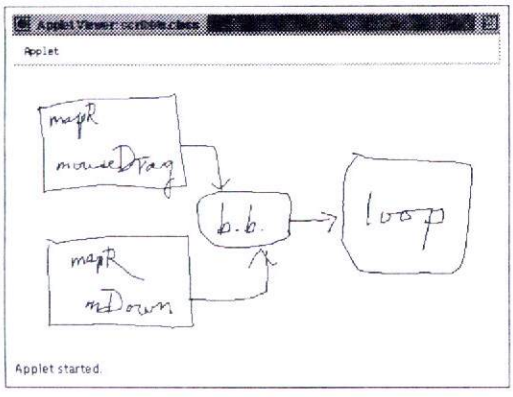

図 4: 落書アプレット scribble

一方 gofjava では、以下に基いてイベントを扱う。

1. イベントを監視する関数は、そのイベントが発生するまでブロックするという性質を有する。
2. イベント監視関数を並行起動することにより、イベント ハンドラの登録に相当する処理を行う。

コンソール入出力関数もイベント ハンドラも、ユーザが何かイベントを発生させるまでブロックする関数として同様に表現される。イベント ハンドラか否かは並行起動されるか否かによって区別される。gofjava は JAVA 等の記述と比較して処理分担がより明確になっていると考えられる。

例えばマウスのクリック 1 回分の監視関数 mouseDown、ドラッグ 1 回分の監視関数 mouseDrag、線描画関数 drawLine、2 つのリストの非決定的マージ関数 mergeR を組み合わせて落書アプレット scribble を記述することができる。

```
> scribble = mapR mouseDown `connectR` \ dws ->
>            mapR mouseDrag `connectR` \ drs ->
>            drawer dws drs
>
> drawer dws drs
>   = mergeR dws drs  `connectR` \ ps@(_:qs) ->
>     seqsR (foldr draw [] (zip ps qs))
>   where
>   draw (A(x,y), B(z,w)) s = (drawLine x y z w):s
>   draw (B(x,y), B(z,w)) s = (drawLine x y z w):s
>   draw _                s = s
```

dws はクリック座標のリスト、drs はドラッグ座標のリストになる。これらを mergeR すると、ps と qs は、クリック座標には A、ドラッグ座標には B なるタグが付いたリストになる。drawLine で線を引くのはクリック–ドラッグ間 (A–B) とドラッグ–ドラッグ間 (B–B) である。

scribble を並行プロセスの記述として見ることもできる (図 5)。mapR mouseDown、mapR mouseDrag、drawer dws drs の 3 つのプロセスが動作し、イベント監視や線描画が個別の関数 (プロセス) で表現される。

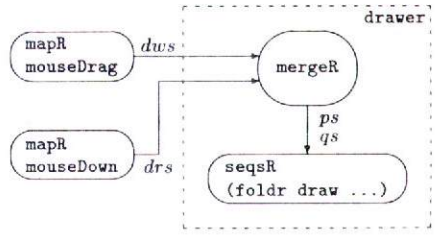

図 5: 並行プロセス scribble

JAVA 等の手続き型言語では、落書アプレットを作成する際には、現在の状態を記憶するオブジェクトを定義することとなる。一方、我々の手法では、処理ごとに別の関数、別のプロセスに分離することができ、これらの関数を宣言的に合成することでプログラムを作成する。現在の状態を明示的に保存する必要もなく、関数間の通信によって状態が伝達されるため、極めてモジュール性が高い。

更に、mapR mouseDown 等は、マウス ボタンなる装置の処理を担うデバイス ドライバと見ることもできる。このように考えると、関数 scribble は全体で、マウスのボタン、マウスのローラ、ビットマップ画面という 3 つの装置を処理する 1 つのデバイス ドライバとなる。

このように我々の手法では、単純な装置を扱うデバイス ドライバを複数結合して 装置群を処理する 1 つのデバイス ドライバを構成的に作成できるという特長を有している。

5.2 OS

echo や scribble では複数のプロセスがそれぞれデバイス ドライバとなって並行動作し、全体として 1 つのプロセス、1 つのデバイス ドライバとなっている。複数のプロセス、複数のデバイス ドライバの管理を行うものが OS であ

るとすれば、scribbleやechoを全体としてOSと見ることも可能である。ただしこれらでは、プロセス数は変化しない。

本節では、プロセス数が動的に変化するより本格的なOSを記述する。OSでは、OS核が資源を管理し、各プロセスは相互にメッセージの送受を行うほか、OS核と要求や応答のメッセージを送受する。

OSを関数型言語で記述した例は過去にもあるが[13, 10, 4]、参照透明性を損う関数を用いているため、関数型言語の利点が生かされていない。

擬データ関数プログラミングでは、リストの非決定的マージを用いても「同じ文脈の同じ式は同じ値を意味する」という性質を保持する。以下に示すOSでは、従来のOSの記述では適用できなかった最適化の適用ができ、動作の説明が従来より詳細にできるようになっている。

ここでは、[13]のOSを例にとって、我々の手法で記述する。この記述によるOSは、gofjava処理系の上で実際に動くものである。

5.2.1 OSとプロセス

ここで説明するOSは図6の構成をとる。OS全体は1つのメッセージ環で構成され、各プロセスはsplitter(図中の ⓝ 等)とmerger(図中の ◹)を経由してメッセージ環に接続される。プロセスProcはメッセージMesgを受け取ってメッセージMesgを送る関数として表現される。メッセージには宛先あるいは送付元の識別子PIdが添付される。メッセージ環を流れるデータは(PId,PId,Mesg)型で、順に(送付元, 宛先, 内容)である。

```
> type PId  = Int
> type Proc = [(PId,Mesg)] -> [(PId,Mesg)]
> data Mesg = Create Proc
>           | New PId
>           | Kill PId
>           | Stop
>           | Syserr String
>           | MNone
>           | MInt Int
>           | MStr String
```

OS核の機能は、プロセスマネージャpm(識別子0)と、デバイスドライバの集合体であるデバイスマネージャdmに分割されており、OS自体は起動プロセスstartupから開始される。OS全体の定義を以下に示す。

```
> os dm startup <<(rms?,rds?)>> = ring
>   where
>   ring  = pm pidq ring' rms
>   ring' = dm (r:ring) rds
>   r     = (0,0,Create startup)
>   pidq  = initqueue 2
```

rmsは非決定的マージのための神託、rdsは装置の処理のための擬データである。擬データが追加されている点とdmが追加されている点が[13]との違いである。

識別子nのプロセスが関数fで表わされる子プロセスを起動するには、メッセージ(0,Create f)を送信する。これはmergerによって(n,0,Create f)の形でメッセージ環にマージされる。pmがこれを受信すると、キューpidqから新しい識別子mを取り出し、splitter mとmerger mでfを狭んで自分と並列に接続する。更にメッセージ環に(0,n,New m)を流す。このメッセージはsplitterでプロセスnに分配される。

プロセスnが(m,MStr "hello!")を送信すれば、メッセージ環を(n,m,MStr "hello!")の形で流れ、splitter mがこれを切り出して子プロセスfに(n,MStr "hello!")を分配する。

(0,Kill m)を送信すれば子プロセスf(識別子m)を終了させることができる。pmがこのメッセージを受信すると(0,m,Stop)をメッセージ環に流す。merger mとsplitter mがこれを検知するとfの出力/入力との接続を切る。Stopがメッセージ環を1周すると、識別子mが回収されpidqに登録、再利用される。

プロセスが装置を利用する場合は負の識別子を有するデバイスドライバにメッセージを送る。例えば(-2,MStr "foo\n")を送信すると画面に文字列が表示される。

5.2.2 メッセージの分配

関数splitterは、メッセージ環から自分用のメッセージのみを切り出す。splitterは[13]と同様である。

```
> splitter myid ((0,destid,Stop):ring)
>   | destid == myid = ((myid,0,Stop):ring,[])
> splitter myid ((srcid,destid,msg):ring)
>   | destid == myid = (r,(srcid,msg):p)
>   | otherwise      = ((srcid,destid,msg):r,p)
>   where
>   (r,p) = splitter myid ring
```

Stopを検知するとそれ以降はメッセージをプロセスに分配しなくなり、プロセスマネージャにmyidを回収させる(第1定義)。第2定義は通常のメッセージの分配である。

5.2.3 メッセージ環へのマージ

関数mergerは、メッセージprocにプロセスのmyidを付加してメッセージ環ringにマージする。擬データrsに対して、wrProcがprocを、wrRingがringを、競争して書き込む。mergeRと同様、メッセージの送付元識別子を書込の排他制御に用いる。

```
> merger myid ring proc rs
>  = wrProc proc rs `par` wrRing ring rs `par` rs?
>  where
>  wrProc [] <<r?:rs?>>
>    | assign (0,myid,Stop) r = True
>    | otherwise              = wrProc [] rs
>  wrProc ((destid,msg):proc) <<r?:rs?>>
>    | assign (myid,destid,msg) r = wrProc proc rs
>    | otherwise
>      = wrProc' r? ((destid,msg):proc) rs
>
>  wrProc' (0,destid,Stop) proc rs
>    | destid == myid           = False
>  wrProc' _               proc rs = wrProc proc rs
>
>  wrRing ((srcid,destid,msg):ring) <<r?:rs?>>
>    | assign (srcid,destid,msg) r
>      = wrRing ring rs
>    | otherwise
>      = wrRing ((srcid,destid,msg):ring) rs
```

プロセスがメッセージを出さなくなった場合は、splitter経由で(0,myid,Stop)をプロセスマネージャ

に送って後処理をする (wrProc 第1定義)。プロセス マネージャ発の Stop を検知するとそれ以降は proc を無視して接続を切断する (wrProc 第2定義、wrProc')。wrProc は、プロセスの出力を最後までメッセージ環に書き込んだか否 (途中で Kill された) かを返す。

5.2.4 プロセス マネージャ

プロセス マネージャ pm は、プロセスの起動と終了やプロセス識別子 (PId) 等を管理する。以下は pm の定義の一部である。プロセスの起動は第1定義、プロセスの終了は第2定義、識別子の回収と再利用は第3定義にある。

```
> pm pidq ((myid,0,Create f):ring) <<(rm?:rms?)>>
>   = merger newid ((0,myid,New newid):o1) o2 rm
>   where o1       = pm newpidq s1 rms
>         o2       = f s2
>         (s1,s2) = splitter newid ring
>         (newid,newpidq) = headqueue pidq
> pm pidq ((myid,0,Kill pid):ring) rms
>   = (pid,0,Stop):pm pidq ring rms
> pm pidq ((myid,0,Stop):ring) rms
>   = pm (addqueue myid pidq) ring rms
```

5.2.5 デバイス マネージャ

[13] では、各装置にそれぞれ1つのプロセスを割り当ててデバイス ドライバとしている。我々の手法でも splitter と merger を用いて複数のデバイス ドライバを結合する。結合したデバイス ドライバ群全体がデバイス マネージャとなる。何も装置が接続されていないデバイス マネージャ dmzero は以下のように定義できる。

```
> dmzero ((myid,destid,mesg):ring) rds
>   = r:dmzero ring rds
>   where
>   r | destid < 0 = (destid,myid,Syserr "dmzero")
>     | otherwise  = (myid,destid,mesg)
```

destid が非負の場合は通常のメッセージなので素通しし、負の場合は処理できるデバイス ドライバがなかったこととなるのでエラーを返す。

ring 内のメッセージを監視するデバッグ用のデバイス マネージャ dmspy は以下の通りである。表示用の puts (print ring) と上記 dmzero が並行動作する。

```
> dmspy ring
>   = puts (print ring) `connectR` \ u ->
>     u `par` dmzero ring
>   where
>   print (r:ring)
>     = "spy:" ++ show r ++ "\n" ++ print ring
```

以下の関数 mkdrv は、デバイス マネージャとデバイス ドライバを結合する。一般にデバイス ドライバ $d_1,\ldots,d_l$ を結合したデバイス マネージャは以下の式で表される。

(-1,d_1) `mkdrv` $\cdots$ `mkdrv` (-l,d_l) `mkdrv` dmzero

簡単に構成ができるのは、高階関数という機能があるからである。以下にデバイス マネージャの例を示す。

```
> ((id,drv) `mkdrv` dm) ring <<(ra?,rb?,rm?)>>
>   = ring'
>   where o1       = dm s1 ra
>         o2       = drv s2 rb
>         (s1,s2) = splitter id ring
>         ring'    = merger id o1 o2 rm
>
> dm0 = dmzero
> dm1 = (-1,getcdrv) `mkdrv` dm0
> dm2 = (-2,putsdrv) `mkdrv` dm1
```

dm1 や dm2 で識別子 -1 と -2 を割り当てた文字 (列) 入出力のデバイス ドライバを以下に示す。

```
> getcdrv ((myid,MNone):rest) <<ri?:ris?>>
>   | i == i = (myid,MInt i):getcdrv rest ris
>   where i = getchar ri
> getcdrv ((myid,_):rest) ris
>   = (myid,Syserr "getcdrv"):getcdrv rest ris
>
> putsdrv ((myid,MStr s):rest) <<ru?:rus?>>
>   = case puts s ru of
>       () -> (myid,MNone):putsdrv rest rus
> putsdrv ((myid,_):rest) rus
>   = (myid,Syserr "putsdrv"):putsdrv rest rus
```

5.2.6 検討

デバイス マネージャの型は一般に以下の通りである。

[(PId,PId,Mesg)] $\to \ll a \gg \to$ [(PId,PId,Mesg)]

型 a は、処理する入出力の種類を示す。この場合 OS 全体の型は以下のようになる。

$\ll$([[(PId,PId,Mesg)]],a)$\gg \to$ ()

例えば UNIX では、処理する装置の追加や削除の際に、OS 核を再構成 (再コンパイル) する必要があった。関数プログラミングの立場からは、再構成によって OS 核の型が変更されると見ることができる。我々の手法では、OS の再構成は高階関数 mkdrv で表現され、ドライバを付加すれば OS の型も変わり、処理する装置の種類や数は型によって自然に示される。

また、上記の os の定義をよく見ると、pm pidq も dm も、以下の型にマッチする。

[(PId,PId,Mesg)] $\to \ll \cdots \gg \to$ [(PId,PId,Mesg)]

つまり、pm pidq や dm は、前節説明のプロセスに相当する。したがって、複数のプロセス マネージャやデバイス マネージャを直接接続したり、外部 OS へのメッセージと OS を管理する OS マネージャを pm 同様に定義して、個々の OS を splitter や merger で接続すれば、OS ネットワークを記述することも可能である。複数のプロセスから構成されるプロセス群を1プロセスと見ることができるため、OS 全体も1プロセスとして扱えるからである。

ここで示したプロセスの動的生成の技法は、GUI で選択できるメニューやボタンが状況によって変化する場合にも利用できる。

6 結論

本稿では、並行動作しつつ非決定的処理を行う複数のプロセスからなるプロセス群を擬データ関数プログラミングによって記述する手法について説明した。本手法では、コンソール入出力や GUI を同じ枠組で扱うことができ、OS の記述の際にも非決定的処理の結果を擬データに保存するので動作の証明が容易であり、様々な最適化技法が利用できる。今後の課題としては、GUI ライブラリの充実、一般プログラマ用の書きやすいプログラミング スタイルの確立、特にメモリ使用量の最適化の実装があげられる。

なお、gofjava は gofer[12] を元に開発したものである。gofer の作者 Mark P. Jones 氏に感謝する。

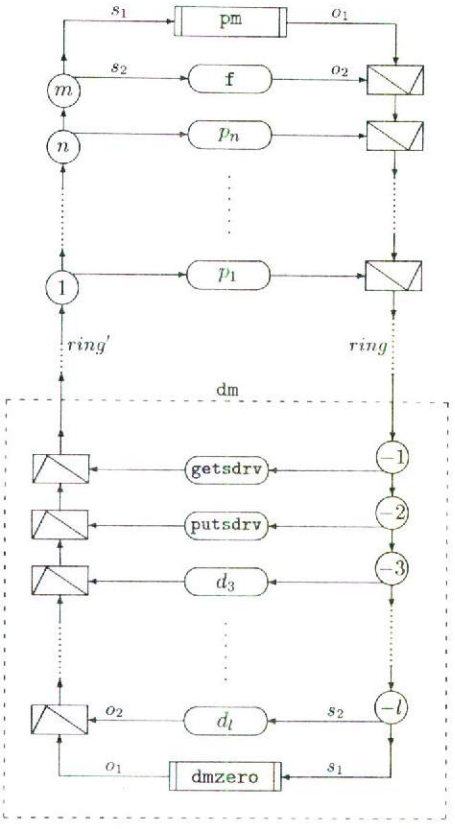

図 6: OS の構成

参考文献

[1] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[2] F. W. Burton. Nondeterminism with referential transparency in functional programming languages. *The Computer Journal*, Vol. 31, No. 3, pp. 243–247, 1988.

[3] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, San Francisco, California, June 1992.

[4] J. Cupitt. *The Design and Implementation of an Operating System in a Functional Language*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, 1990.

[5] David Flanagan. *Java in a Nutshell — A Desktop Quick Reference for Java Programmers*. O'Reilly & Associates, Inc., 1996.

[6] Andrew Gordon. *Functional Programming and Input / Output*. PhD thesis, Computer Laboratory, University of Cambridge, August 1992.

[7] Andrew Gordon. An operational semantics for I/O in a lazy functional language. In *Functional Programming and Computer Architecture*, pp. 136–145, Copenhagen, June 1993. ACM Press.

[8] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pp. 73–82, Philadelphia, PA, May 1996. ACM Press.

[9] Paul Hudak and R. S. Sundaresh. On the expressiveness of purely functional languages. Technical Report YALEU/-DCS/RR-665, Department of Computer Science, Yale University, December 1988.

[10] Paul Hudak, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language version 1.2. *SIGPLAN Notices, Haskell Special issue*, Vol. 27, No. 5, p. Section R, May 1992.

[11] John Hughes and Jon O'Donnel. Expressing and reasoning about non-deterministic functional programs. In Kei Davis and John Hughes, editors, *Functional Programming, Glasgow 1989*, pp. 308–328. Springer-Verlag, August 1989.

[12] Mark P. Jones. Gofer 2.28 release notes, Feb 1993. ftp://ftp.-cs.nott.ac.uk/nott-fp/languages/gofer.

[13] S. B. Jones and A. F. Sinclair. Functional programming and operating systems. *The Computer Journal*, Vol. 32, No. 2, pp. 162–174, April 1989.

[14] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*, pp. 71–84, January 1993.

[15] Rishiyu S. Nikhil and Arvind. Programming in *id* — a parallel programming language. Technical Report Draft, Massachusetts Institute of Technology, June 1992.

[16] Rishiyur S. Nikhil. Id language reference manual version 90.1. Technical Report memo 284-2, Computation Structures Group, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.

[17] Paul Roe. Some ideas on parallel functional programming. In Kei Davis and John Hughes, editors, *Functional Programming, Glasgow 1989*, pp. 338–352. Springer-Verlag, August 1989.

[18] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, Vol. 21, No. 3, pp. 413–512, Sep 1989.

[19] Masato Takeichi. Evaluation partial order and synchronization mechanisms in parallel functional programs. Technical Report METR 89-07, Department of Mathematical Engineering and Insturmentation Physics, Faculty of Engineering, University of Tokyo, Bunkyo-ku, Tokyo, Japan, 1989.

[20] Simon Thompson. Interactive functional programming. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley Publishing Company, 1990.

[21] D. Turner. An approach to functional operating systems. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley Publishing Company, 1990.

[22] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pp. 344–358, 1988.

[23] Philip Wadler. Monads for functional programming. Lecture notes for Marktoberdorf Summer School on Program Design Calculi, Springer-Verlag, Aug 92.

[24] 石井裕一郎, 武市正人. 関数型言語における参照透明性を保持する参照型. 日本ソフトウェア科学会第 12 回大会論文集, pp. 257–260, Oct 1995.

[25] 石井裕一郎, 武市正人. 参照型による擬データ関数プログラミング, Jun 1996. コンピュータ ソフトウェア投稿中.

[26] 淵一博, 古川康一, 溝口文雄 (編). 並列論理型言語 GHC とその応用. 共立出版株式会社, 1987.