簡潔な並行言語 L0: 論理型と関数型プログラミングを越えて

平田 圭二　原田 康徳　山崎 憲一
NTT 基礎研究所
{hirata@nefertiti,hara@square,yamazaki@nuesun}.brl.ntt.jp

GHC, Oz などに代表される並行言語が持つ様々な機能を検討し，より簡潔な要素から成る並行言語 L0 を設計した．L0 の特徴は，モード付き論理変数，λ 項による述語の抽象，任意の変数スコーピング，制御のための宣言的な結合子などである．本論文では L0 の構文と実行規則を与え，幾つかのプログラム例を示す．L0 の記述力や構成要素などに関して議論を行う．

A Simple Concurrent Language L0: Beyond Logic and Functional Programming

Keiji Hirata　Yasunori Harada　Kenichi Yamazaki
NTT Basic Research Laboratories

By examining several notions possessed by conventional concurrent languages, such as GHC and Oz, we have designed a simple yet powerful concurrent language L0. The features of L0 are moded logic variable, predicate abstraction by λ term, arbitrary variable scoping, declarative connectives for control and so forth. This paper presents the syntax and operational semantics for L0, exhibits sample programs and discusses some issues regarding descriptive power and a set of primitives.

1 はじめに

近年，より簡潔で柔軟なプログラミング言語を設計するために，様々な計算モデル間の統合，再構築，拡張などが試みられている [5] [6] [9] [7] [1]．本研究の目的は，より簡潔で柔軟な並行言語を設計するために，単純で直交した言語の構成要素を選択することである．

これまでの研究の多くは，少ない個数の機能や概念から成る記述力の高い言語の設計を目指していた．しかしここで問題なのは，基本機能や概念など構成要素の数が少ない言語が簡潔な言語なのだろうか，という点である．例えば Moded FGHC [10] のガード及びコミットオペレータは複雑で，共有変数の排他制御，逐次制御，条件判定，受動単一化による構造の分解，スレッドの絞り込みという機能を持っている．また，ガード部に出現する単一化の '=' とボディ部のそれでは機能が異なっている．つまり必然的に，1 つの要素や表現に多くの機能を集約せざるを得ないのである．1 つの構成要素や表現には単純な 1 つの機能を割り当てるべきである．

しかし一方で，構成要素を過度に分解し記述レベルが低くなり過ぎても問題が生じる．例えば π 計算は非常に単純で直交した構成要素から成っているので，実用的言語なら通常備えているようなデータ構造や制御構造を全て π 計算にエンコードして実現しなければならない．従って π 計算をベースに実際に言語を設計する際には，実行と記述の効率を考慮し，新たな構成要素 (例えば record, λ 項, if-then-else, 等価性判定の組込み述語など) を追加せざるを得ない [8]．ここで記述の効率とは，プログラマが扱う構成要素の粒度の問題と言い換えることもできる．

我々がここで提案する簡潔で柔軟な言語とは，単純で直交した必要最低限の要素をプログラマに提供する言語であり，本来的に実用に耐え得るだけの記述レベルを保持することを目指している．

本論文の構成は次の通り．第 2 章では L0 の基本的な概念や特徴を説明し，実行イメージを与える．第 3 章では L0 の実行規則を遷移ダイアグラムで与え，動作説明のためのサンプルプログラムを示し，第 4 章では L0 の実際のプログラミング例を 3 つ挙げる．第 5 章では非決定性を表現するために導入した排他制御の組込み演算子について述べ，第 6 章で本論文のまとめを行う．

2 L0 の基本的な仕組み

2.1 構成要素

まず，L0 を構成する主な特徴を概観する．

λ 項による述語の抽象 [2]: プログラムを関係として記述できるような言語は一般に宣言的と呼ばれ，論理型言語や関数型言語などが代表的である．L0 では述語を採用したがその理由は，並行性を自然に表現できることと，返り値のための引数を陽に書くことでより複雑なプロセス構造を記述できるからである．さらに L0 では高階プログラミングを行うために，述語をホーン節ではなく，λ 項による無名述語で動的に定義する (第 3.5 節)．

モード付き論理変数 [10]: 述語で表現されるプロセス間同期を制御する機構として，論理変数に対する読み書きに基づくデータフロー同期を採用する．その読み書きのモードをプログラム中で陽に指定することで，論理変数の出現場所によらず，同じ表現に同じ機能を持たせることができる (第 2.2 節)．モード付き基礎項等式理論 (MGE) は，モード付き論理変数を含む単一化の実行順序と，関数項を含む値の等価性を定義する (第 2.4 節)．

任意の変数スコーピング: 入れ子構造によるプログラムの構造化を行うためには, 変数のスコーピングをプログラマが指定できなければならない. そこで関数型言語にならい, L0 では変数束縛のスコープをプログラム中で陽に指定することにした. 例えば上述の λ 項を局所的な論理変数に束縛することで, 述語の隠蔽やモジュール機能が実現できる.

直交した並行制御プリミティブ: and, or, Sequence 式は, それぞれ and 実行, or 実行, 逐次実行を表現し, 引数の式の真理値を計算する単純な機能だけを持ち, 制御構造をより宣言的に記述することができる (第 3.3, 3.4 節). コミットオペレータの代わりに導入した or は, 変数の束縛環境をコピーしない.

非決定性: reactive なシステムを記述するには非決定性を表現する必要がある. 非決定性を表現する方法は大別して, 専用並行制御プリミティブの導入と排他制御による書換えの 2 つある. L0 では, 構成要素の直交性を考慮し, 第 5 章で導入する排他的データアクセスを行う組込み演算子と, 上述の並行制御プリミティブの組合せで非決定性を表現する.

2.2 モード付き論理変数

ある論理変数にアクセスする時, 値を与えたり値を読み出したりする場合をそれぞれ out-moded, in-moded と呼び区別する. L0 の構文上, `!X` は論理変数に値を与える出現 (out-moded), `X` は値を読み出す出現 (in-moded) を表す. 即ちプログラマが ! を付けると変数のモードは out に, ! を付けないと in になり, `(= !X t)` は変数と項の能動単一化, `(= X t)` は受動単一化を表す. プロセス p が `(= !X t)` を実行する前に, 別のプロセス q が `(= X t)` を実行しようとすると, q の受動単一化は suspend し, プロセス間のデータフロー同期が行われる.

2.3 構文と実行イメージ

L0 の構文を図 1 に示す. 予約語は `nu`, `and`, `or`, `lambda`, `->`, `=`, `#t`である. L0 のプログラムは 1 つの式 (*Exp*) であり, 式 は `#t`を返すか何も返さないかのいずれかである. `nu` は変数の宣言とスコーピングを行い, 全ての変数は必ず `nu` で宣言される. `and` は $Exp_1 \cdots Exp_m$ を並行に実行し, すべての Exp_i が `#t`を返すと, その `and` 式自身も `#t`を返す. `or` も $Exp_1 \cdots Exp_m$ を並行に実行するが, どれか 1 つの Exp_i が `#t`を返すと, その `or` 式自身が `#t`を返す. `->` (Sequence) 式 は, まず Exp_1 を実行し `#t`が返ってきたら, 次に Exp_2 を実行する. 以下では *Lam*を λ 項と呼び, *App*は λ 項に引数を適用したものである. また, 実引数にはモード付き論理変数 (*MVar*) のみ許される. (`V` $MVar_1 \cdots MVar_n$) の `V` には, λ 項が束縛されることが期待されている. 関数項 (*Func*) の引数は *MVar*のみ許される. 尚, L0 の項とは, 図 1 における定数 *c*, *MVar*, *Func*, *Lam*, *App* のいずれかである.

L0 では, 1 つの式 (*Exp*) が与えられると, その複数個の部分式を複数のプロセスが並行に書換えて行く. それら複数のプロセスが 1 つの Store を共有し, Store への新しい変数の導入, 変数への値の読み書きなどを並行に行う. Store は, `(= V t)` (`t` は項) の形をした能動単一化と (`= V` *Func*) の形をした受動単一化の conjunction である. Store 中のこれら単一化は制約ともみなすことができ, Oz の実行イメージに近い [6].

2.4 モード付き基礎項等式理論

モード付き基礎項等式理論 (Moded Ground Equality Theory, MGE) は, L0 の項の同値性を与える演繹規則である (図 2). 基本的には Clark の等式理論と同様に, 同じ形の項は等しいということを定義している. しかし MGE は, 基礎項に対してのみ等しいと判定できること, 現われる変数がモード付きであるという点で異なっている. 後述の実行規則 (図 3) と組合せて, *MVar*を含む単一化の順序制御を規定している.

MGE に従うと, 例えば `(= X Y)` のような受動単一化は, 変数 `X`, `Y` の値が完全な基礎項でかつ Clark の等式理論の意味で等しい場合に限り真となる. たとえ `(= X X)` でも, `X` の値が完全に基礎項に具体化されないと, 真と決定できない. 規則 1 の代わりに ∀ `(= X X)` を用いると, 変数を含む項間の等価性が判定できるようになる. 変数 `X` に関して, 値を書き込む `!X` という出現は, 高々 1 回まで許される. 規則 7 より, もし同じ変数に対して out-moded の出現が 2 回以上ある場合は inconsistent になる. L0 では λ 項と *App* を受動単一化の引数として書けないので, これらには規則 5, 6 だけが関係する (第 3.5 節).

3 実行規則

本章では, Plotkin 流の遷移ダイアグラムに従って L0 の実行規則を与える (図 3). プログラムの実行状態は $\langle \mathcal{P}, \mathcal{S} \rangle : \mathcal{V}$ の三つ組 (configuration) で表現され, $\mathcal{P}$ は式 (*Exp*), $\mathcal{S}$ は先に述べた Store, $\mathcal{V}$ は $\mathcal{S}$ 中に含まれる変数の集合 (single set) である. $\mathcal{P}$, $\mathcal{S}$ 中に出現する変数はモード付きであり, $\mathcal{V}$ の変数にモードは付いていない. プログラム P の実行は $\langle P, \texttt{\#t} \rangle : \{\}$ から始まる遷移に相当し, 一般に

$$\frac{\langle \mathcal{P}, \mathcal{S} \rangle : \mathcal{V}}{\langle \mathcal{P}', \mathcal{S}' \rangle : \mathcal{V}'} \quad \textit{if 遷移が可能であるための条件}$$

のような形をした遷移規則に従ってトップレベルの *Exp* が書換えられていく. やがてプログラムは, $\langle \texttt{\#t}, \mathcal{S} \rangle : \mathcal{V}$ を含む irreducible な状態になって停止するか, 永久に止まらないのいずれかである. 論理的には, $\exists \mathcal{V}\ \mathcal{S}$ が解となる.

L0 では, `#t`が返ってこないという意味で, デッドロックと irreducible な受動単一化と無限ループは同一に扱われ, 真に対応するシステム定数 `#t`はあるが, 偽に対応するものは無い. $S \models_{MGE} E$ は, 式の conjunction S から MGE に従い式 E が演繹されるということを表す. L0 の実行規則で与えられているのは, Store が consistent な状態 (図 2 の 7, 8 が成立している) での振る舞いであり, Store が inconsistent な時の動作は無意味である.

3.1 モード付き論理変数の単一化

図 3 の Active Unification 規則に従うと, すべての能動単一化は無条件に Store に格納され, その式は `#t`を返

プログラム	::=	Exp
Exp	::=	(nu ($V_1 \cdots V_m$) Exp) | (and $Exp_1 \cdots Exp_m$) | (or $Exp_1 \cdots Exp_m$) | (-> Exp_1 Exp_2) | App | $AUnif$ | $PUnif$ | #t
App	::=	(Lam $MVar_1 \cdots MVar_n$) | (V $MVar_1 \cdots MVar_n$)
Lam	::=	(lambda ($MVar_1 \cdots MVar_n$) Exp)
$AUnif$	::=	(= !V W) | (= !V c) | (= !V $Func$) | (= !V Lam) | (= !V App)
$PUnif$	::=	(= V W) | (= V c) | (= V $Func$)
$Func$	::=	(f $MVar_1$ $\cdots$ $MVar_m$)
$MVar$	::=	V | !V

ここで, V, W は変数記号 (大文字で始まるシンボル), f は関数記号, c はストリングや整数を含む定数記号 (f, c は小文字で始まるシンボル), $m \geq 1$, $n \geq 0$.

図 1: L0 の構文

1	$\forall$ (= X Y) $\leftarrow$ (= !X (f μ_{u_1}U$_1$ $\cdots$ μ_{u_m}U$_m$)) $\wedge$ (= !Y (f μ_{v_1}V$_1$ $\cdots$ μ_{v_m}V$_m$)) $\wedge$ (= U$_1$ V$_1$) $\wedge \cdots \wedge$ (= U$_m$ V$_m$)	
2	$\forall$ (= X Y) $\leftarrow$ (= !X c) $\wedge$ (= !Y c)	
3	$\forall$ (= μ_1X$_1$ $\overline{\mu}_1$Y$_1$) $\wedge \cdots \wedge$ (= μ_mX$_m$ $\overline{\mu}_m$Y$_m$) $\leftarrow$ (= U (f μ_1X$_1$ $\cdots$ μ_mX$_m$)) $\wedge$ (= !U (f $\overline{\mu}_1$Y$_1$ $\cdots$ $\overline{\mu}_m$Y$_m$))	
4	$\forall$ (= U c) $\leftarrow$ (= !U c)	
5	$\forall$ (= !X t) $\leftarrow$ (= !X Y) $\wedge$ (= !Y t)	
6	$\forall$ (= s t) $\leftarrow$ (= t s)	
7	$\forall \neg$ ((= !X t) $\wedge$ (= !X s))	(t と s が同じ場合も含む)
8	$\forall \neg$ (= !X t)	(項 t は変数 X の出現を含む)

ここで X, Y, U, V は論理変数, c は定数, f は関数記号, s, t は項, μX は X (in-moded) あるいは !X (out-moded). $\overline{\mu}$X は μX と逆のモードの出現を表す. $m \geq 1$.

図 2: モード付き基礎項等式理論 MGE

す (#tに書き換えられる). 受動単一化については, Passive Unification (C) 規則にあるように, Store 中の能動単一化の conjunction S からその受動単一化の式が演繹できる時に限り #tを返す.

Sample Programs 3.1

```
1 (nu (X) (= !X 17))
2 (nu (X Y) (and (= !X a) (= !Y X)))
3 (nu (X) (and (= X 23) (= !X 23)))
4 (nu (X Y) (and
      (= !X d) (= Y c)
      (nu (X) (and (= !X c) (= !Y X)
                   (= X c)))))
```

それぞれの式は #tを返し Store には次のような能動単一化が格納される: 1 (= !X 17), 2 (= !X a)$\wedge$(= !Y X) (X も Y も a), 3 (= !X 23), 4 内側の X を X' で表すと, (= !X d)$\wedge$(= Y X')$\wedge$(= X' c).

次のプログラムはいずれも #tを返さない:

Sample Programs 3.2

```
1 (nu (X) (= X 5))
2 (nu (X) (and (= X 23) (= !X 31)))
```

1 は X の具体化が不十分なために受動単一化が suspend する. 2 の (= X 23) は irreducible である.

3.2 関数項

関数項の引数には変数のみ出現する. MGE の 8 番目の規則 (図 2) より無限項は許されない. Passive Unification (F) 規則では関数項の引数個数が 1 の場合だけ示したが, 2 以上の場合も同様. 次の式はいずれも #tを返す.

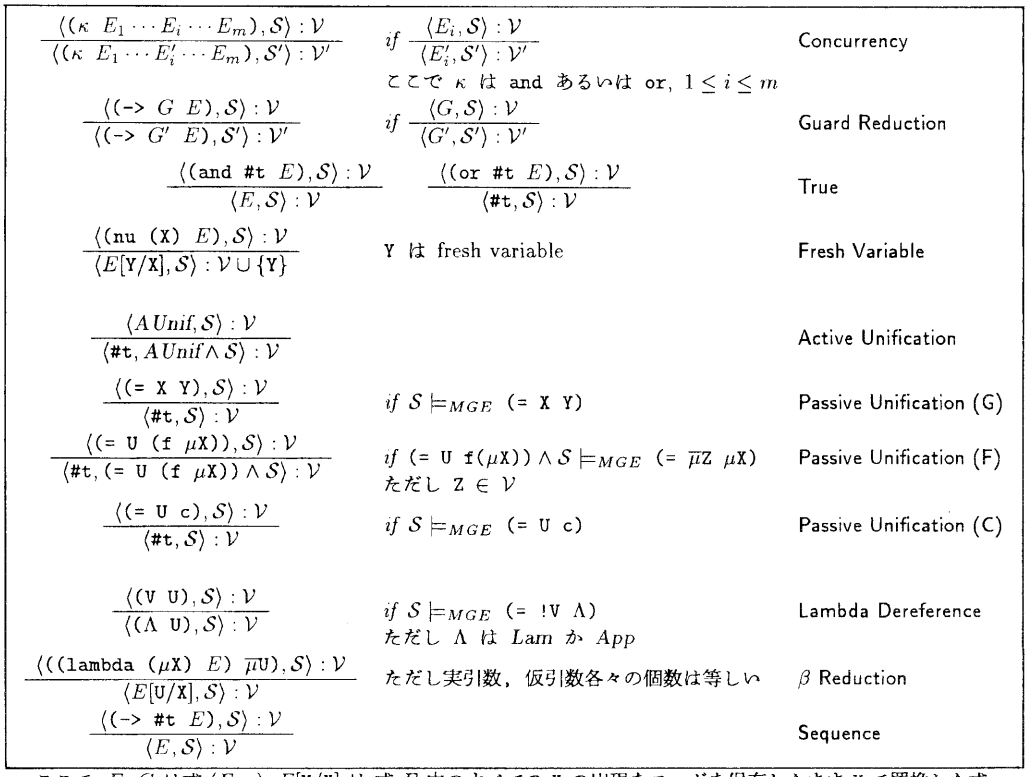
Sample Programs 3.3

```
1 (nu (X Y) (= !X (f Y)))
2 (nu (X A B) (and (= !A a) (= !X (f A))
                   (= X (f !B)) (= B a)))
3 (nu (X A B) (and (= !A 29) (= !X (g A !B))
                   (nu (C) (= X (g !C C)))))
```

2 の式を実行した後の Store は, Passive Unification (F) 規則を用いて (= !X (f A))$\wedge$(= !A a)$\wedge$(= X (f !B)) となる. この Store の状態を S とすると, $S \models_{MGE}$ (= !B a). つまり B は a であることが演繹される. 3 のように関数項の引数に同じ変数が複数回出現してもプログラマが正しくモードを付けていれば inconsistent にはならない. もし (= X (g !C C)) が (= !X (g A !B)) より先に実行された場合は suspend する.

Sample Programs 3.4

```
(nu (X Y Z) (and (= !X (f Y)) (= X (g !Z))))
```



ここで, E, G は式 (*Exp*), E[Y/X] は 式 E 中のすべての X の出現をモードを保存したまま Y で置換した式.

図 3: L0 の操作的意味を定義する遷移規則

この場合は，どのような順序で実行しても Store には (= !X (f Y)) が格納されるが, (= X (g !Z)) が irreducible なので, Z は未定義のまま, 式全体は #t を返さない.

3.3 and と or

and も or も各引数の式全部を fair に実行する. or の引数である式のいずれか 1 つが #t になった時以降なら, True 規則はいつでも実行できる. もしその実行を任意時間遅らせた場合, #t になった以外の引数に関しては, 実行を中断しても良いしそのまま続行しても良い (実装依存). つまり, or 式自体が #t を返すこと以外は特に規定していない. ただし or が現れても, 変数の束縛環境をコピーしない. この and や or と第 5 章で述べる exch/3 という組込み演算子と合わせて, 非決定的な動作を記述できる. 非決定性を表現するために, コミットの代わりに or を導入したことのメリットは Or-fairness と宣言性であるが, その代償としてプロセス制御のための AND-OR 木の管理コストが deep な Committed Choice 言語程度には重くなることが予想される.

Sample Programs 3.5

```
(nu (X Y) (and
    (= !X (f Y)) (= !Y 31)
    (or (nu (A) (and (= X (f A)) (= A 31)))
        (nu (B) (and (= X (f B)) (= B 37))))))
```

上のプログラムの実行後, Store には (= !X (f Y))∧(= !Y 31)∧(= X (f A)) は必ず格納されるが, (= X (f B)) が格納されるかどうかは分からない. しかし, 式全体としては #t を返す.

3.4 Sequence 式

(-> Exp_1 Exp_2) という式が現れたら, Exp_1 から #t が返ってくるのを待って Exp_2 の実行を開始する. 式全体の返り値は Exp_2 の返り値に等しい. Sequence 式は逐次性を保証する. L0 の場合, Sequence 式を用いても or を記述できず, 従って Sequence 式と or は直交した構成要素である.

Sample Programs 3.6

```
1 (nu (X Y) (and (-> (and (= X c) (= !Y d))
                     (= Y d))
                 (= !X c)))
2 (nu (X Y) (or (= !X 3)
                (-> (= X 3) (= !Y 5))))
```

2 のプログラム実行後の Y の値は実装依存である.

3.5 λ 項

機能としては通常の λ 式と同様に，仮引数と実引数の束縛 (置換) を行う．β 簡約に相当する操作として，β Reduction 規則が与えられている．λ 項の仮引数はお互いに異なる．λ 項自体は変数と能動単一化できる (変数に代入できる) が，受動単一化はできない．つまり，λ 項どうしの等価性 (β 簡約に基づく等価性や α 等価性) の検査はできない．

Sample Programs 3.7

```
1 (nu (Y) (and ((lambda (X) (= !X i)) !Y)
               (= Y i)))
2 (nu (L Y)
      (and (L Y) (= !Y j)
           (= !L (lambda (!X)
                   (or (= X j) (= X k))))))
3 (nu (K L Mh Mhk H) (and
      (= !L (lambda(X Y)
              (and (= !X h) (= Y k))))
      (= !Mh (L !H)) (= !Mhk (Mh K)) (= !K k)
      (Mhk)))
```

上の 2 のように，L0 では関数や述語を定義する代わりに λ 項を変数に代入して，任意の場所で引数に適用できる．この意味において L0 では高階プログラミングが可能であると言える．3 の 4 行目，能動単一化の引数の位置に *App* の構文が現われているが，ここは式が実行される場所ではない．L0 はレキシカル・スコービングであり，この時は *App* の現われる文脈での変数環境を実引数が参照するだけである．最終的に 5 行目の (構文上) 0 引数の *App* の個所で β Reduction 規則が実行される．つまり (*Lam* A1 A2 ··· An) ≡ (···((*Lam*A1) A2)···An) である．この適用の時点で実引数の個数と仮引数の個数は等しくなければならない．なお，図 3 の Lambda Dereference 規則，β Reduction 規則において，適用の引数個数が 0 および 2 以上の場合は省略したが明らかであろう．

4 プログラミング例

本章では 3 つの L0 プログラム例を示す: Append プログラム (図 4), Member 判定述語の flat 版と deep 版 (図 5), リストの要素の総和を求めるプログラム (図 6). 以下，各プログラムに説明を加える．

```
(nu (App L0 L1 Ans) (and
    (= !App (lambda (!X !Y Z)
              (or (nu (H Xs Zs)
                    (-> (= X [!H|!Xs])
                        (and (= !Z [H|Zs])
                             (App Xs Y !Zs))))
                  (-> (= X []) (= !Z Y)))))
    (= !L0 [1,2,3]) (= !L1 [4,5])
    (App L0 L1 !Ans)))
```

図 4: Append プログラム

図 4 での Prolog 流リスト記法 [|] は (cons X Y) のマクロであり，同様に [1,2,3] もマクロである．図 5 中，

```
(nu (MemberFlat MemberDeep X Y0 Y1 Dum) (and
    (= !X 4)
    (= !Y0 [1,23,4,56,7,8])
    (= !Y1 [Dum,1,23,4,56,7,8])
    (= !MemberFlat
       (lambda (!X !L) (or
         (nu (E M)
             (-> (= L [!E|!M])
                 (or (-> (int_eq X E) #t)
                     (-> (int_neq X E)
                         (MemberFlat X M)))))
         (-> (= L []) (nu (A) (A))) )))
    (= !MemberDeep
       (lambda (!X !L)
         (nu (E M) (and
           (= L [!E|!M])
           (or (int_eq X E)
               (MemberDeep X M))))))
    (MemberFlat X Y0)
    (MemberDeep X Y1)))
```

図 5: Flat な Member と Deep な Member

(nu (A) (A)) の部分は，値を返さない式 (例えば (nu (A B) (= A B))) なら何でも OK である．int_eq/2, int_neq/2 は，整数が等しいか，等しくないかを判定する組込み演算子である．deep 版は flat 版に比べて宣言的な記述になっている．図 6 のプログラムの特徴は，大域変数

```
(nu (Def G List Res Zero) (and
    (= !Zero 0)
    (= !Def
       (lambda (G R)
         (nu (F) (and
           (= !F (lambda (!I !L) (or
             (nu (X T In)
                 (-> (= L [!X|!T])
                     (and (add I X !In)
                          (F In T))))
             (-> (= L []) (= I !R)))))
           (= !G (F Zero))))))
    (= !List [10,20,30])
    (Def !G !Res)
    (G List)))
```

図 6: λ 項と大域変数によるモジュールと述語隠蔽

の使用と λ 項を局所変数に代入することで，モジュール機能，述語隠蔽を実現している点である．add/3 は整数加算のための組込み演算子である．

5 排他制御のための組込み演算子

reactive なシステムを実現するためには，非決定的な動作をした時の副作用をデータとして残す必要がある．L0 ではそのために，プロセス間で共有されているデータに対して排他制御をする組込み演算子exch/3 を導入する．構文的には (exch E !Old New) は式 (*Exp*) であり，その操作的な意味を 図 7 に示し，Store 中のデータを書換える

$$\frac{\langle (\texttt{exch E !Old New}), (\texttt{= !P t}) \wedge \mathcal{S} \rangle : \mathcal{V}}{\langle \texttt{\#t}, (\texttt{= !P New}) \wedge (\texttt{= !Old t}) \wedge \mathcal{S} \rangle : \mathcal{V}} \quad \textit{if} \quad (\texttt{= !P t}) \wedge \mathcal{S} \models_{MGE} (\texttt{= !E t}) \qquad \text{Exchange}$$

t は定数, *Func*, *Lam*, *App* のいずれか.

図 7: exch/3 の操作的意味

様子を図 8 に示す. (= P t) (t は変数以外の項) が成立

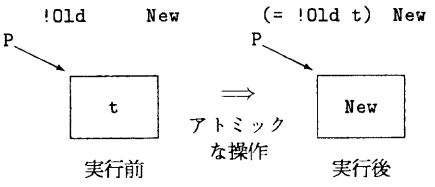


図 8: (exch P !Old New) の動作

している時に, exch/3 を実行すると, P の値を読み出して Old と単一化し ((= !Old t)), t を New で置き換える. この読み出して書き込むという一連の操作はアトミックである. 実行前に P が具体的化されていない場合 exch/3 は suspend する. これは Oz の cell の機能に似ている [6]. この exch/3 を用いて FGHC で書けるようなストリームマージャを実現することができる. ただし, FGHC のストリームマージャでは Or-fairness が成り立たないのに対し, exch/3 を用いたストリームマージャでは Or-fairness が成り立つ. 同様に, 配列の各要素に対して exch/3 を実行する excha/4 も定義できる ((excha Arr Idx !Old New) のように呼び出す). excha/4 を使うと, 例えば配列上の要素を直接書き換えるクイックソートを記述することができるが, 紙面スペースの制約上, L0 によるマージャやクイックソートのプログラムは割愛する.

6 おわりに

新しい言語を提案する意義は, 今までとは異なる思考法をプログラマに提供する点にある[1]. 並行言語 L0 は, 簡潔で柔軟な並行計算モデルの 1 つをプログラマに与えることができた. L0 の設計において, 論理プログラミングと関数プログラミングの統合は意図していない. また, L0 を設計したそもそもの動機は, 並行言語を構成する必要最低限の単純で直交した要素とは何かを調べることであり, その過程を通じて L0 が設計された. この観点から既存の並行言語を見直す作業も非常に大切である.

次に, ある機能の実現に関して, L0 の構成要素の組合せと既存言語のそれを比較する. まず, 逐次制御に関して L0 では Sequence 式で記述するが, 他の言語では, ガード (GHC), then (Oz), "." (π 計算) などを用いる. 条件分岐に関して L0 では or と Sequence 式の組合せで実現する. 他の言語では, ガードとコミットオペレータ, ガードと then, and と ".", (parallel) cond などを用いる. 非決定的な副作用に関して L0 ではexchと and (あるいは or) を組合せて記述する. 他の言語では, ガードとコミットオペレータ, exch と and, リソースの概念などを用いる. さらに, L0 の構成要素を組み合わせることで, 他の言語の構成要素を実現することも可能である.

今後の展望について述べる. プログラマが論理変数のモードを陽に指定すること, and, or, Sequence 式という論理的結合子を用いること, λ 項と変数の任意のスコービングの採用により, L0 ではプログラム構造を一様に記述することが可能となった. これより, 例えば fold/unfold 変換などが容易に適用できるのではないかと考えている. 一方, L0 プログラムが GHC や π 計算など既存の並列言語にどのようにエンコードできるか (あるいはできないか) については未検討である. 現在 L0 インタプリタを KLIC [3] で作成し, 本論文に示したプログラムを含むテストコーディングを行い, 簡潔さ, 構成要素の直交性, 記述力に関して検討を加えている. 今後は, 言語の精練, 意味論の付与, 効率の良い実装についての研究を進めて行く予定である.

謝辞: 本研究の機会を与えて下さった NTT 基礎研究所 石井健一郎氏に感謝します.

[1] 文献 [1] の CONCLUSION (p.227) は一読に値する.

参考文献

[1] W. Chen, M. Kifer and D.S. Warren, HiLog: A Foundation for Higher-Order Logic Programming, *J. Logic Programming*, Vol.15, pp.187–230, 1993.

[2] W. Chen and D.S. Warren, Predicate Abstraction in Higher-Order Logic Programming, *New Generation Computing*, Vol.14, pp.195–236, OHMSHA, 1996.

[3] T. Chikayama, *KLIC-2.001 User's Manual*, ICOT, Available through anonymous FTP from `http://www.icot.or.jp`, 1995.

[4] D.P. Friedman, M. Wand and C.T. Haynes, *Essentials of Programming Languages*, The MIT Press, 1993.

[5] M. Hanus, The Integration of Functions into Logic Programming: from Theory to Practice, *J. of Logic Programming*, Vol.19, No.20, pp583–628 (1994).

[6] M. Henz, G. Smolka and J. Würtz. Oz – A Programming Language for Multi-Agent Systems, In *Proc. of IJCAI'93*, pp.404–409, 1993.

[7] D. Miller, *λProlog: An Introduction to the Language and its Logic*, unpublished draft, Dept. of Comp. and Info. Sci., Univ. of Pennsylvania, 1996.

[8] B.C. Pierce and D.N. Turner, Concurrent Objects in a Process Calculus, *LNCS*, Vol.907, pp.187–215, Springer-Verlag, 1994.

[9] E. Tick, The Deevolution of Concurrent Logic Programming Languages, *J. of Logic Programming*, Vol.20, pp.89–123, 1995.

[10] K. Ueda and M. Morita, Moded Flat GHC and Its Message-Oriented Implementation Technique, *New Generation Computing*, Vol.13, pp.3–43, OHMSHA, 1994.