実行機構の類似性に着目した関数型言語と論理型言語の融合

山崎憲一[†] 吉田雅治[††]
天海良治[†] 竹内郁雄[†††]

本論文では，Lispをベースとして論理型プログラミングを融合したマルチパラダイム言語TAOについて述べる．これまで研究されたきた融合型言語においては，純粋な関数型言語と論理型言語を融合するものが多かったが，TAOではすべてのLispプリミティブと論理型プログラミングを利用できる．TAOでは，Lispと論理型言語の実行機構の類似性に着目した融合を行う．ここで，実行機構の類似性とは，状態変数と論理変数，関数呼び出しと述語呼び出し，大域的脱出とバックトラックなどを指す．たとえば，`catch/throw`のような脱出とバックトラックには類似性があり，バックトラックを大域脱出の一種と考えることができる．これにより，バックトラックが，Lispの大域脱出に関するさまざまなプリミティブを自然に利用できるようになる．

Amalgamation of Functional and Logic Programming Languages Based on Similarities in Execution Mechanisms

KENICHI YAMAZAKI,[†] MASAHARU YOSHIDA,[††] YOSHIJI AMAGAI[†]
and IKUO TAKEUCHI[†††]

This paper describes a Lisp-based multi-paradigm programming language TAO that incorporates a logic programming (LP) paradigm. Whereas previous multi-paradigm languages only supported purely functional and LP facilities, TAO supports all Lisp primitives as well as LP facilities. This amalgamation of Lisp and LP is done by making use of the similarities in their execution mechanisms, giving a natural semantics to the amalgamation. The similarities include state vs. logic variables, function vs. predicate calling, and non-local exit vs. backtracking. For example, the similarity between non-local exit (`catch/throw`) and backtracking enables backtracking to be regarded as a kind of non-local exit, so that backtracking can use all Lisp facilities which support non-local exit.

1. はじめに

実世界の問題を扱うためには，実時間の制約の中での知的な行動に関する記述ができなければならない．そのようなシステムには次の性質が要求される．

(1) 手続き的実行の柔軟な制御と実時間制約の記述が可能
(2) 記号処理についての豊かなプリミティブを持つ
(3) 多様な計算パラダイムを提供する

我々はLispをベースとし，実時間記述プリミティブ，実時間GCおよび実時間OSを組み合わせることで，要求(1)，(2)の性質を持ったシステムが構築できることを示した[1),2)]．

要求(3)は，複雑で多くの側面を持った問題を記述するために必須である．現在，代表的な計算パラダイムとして手続き型，関数型，論理型，オブジェクト指向があるが，いずれも実世界の問題記述に重要である．特にLispは，副作用を含む手続き型の記述と関数型の記述が可能という意味からも，本研究のベース言語に適していると考える．なお，並行計算の記述も重要な要求事項であるが☆，ここでは逐次計算のパラダイムのみを考える．

計算パラダイムを融合する研究，いわゆる融合型言語の研究も比較的古くから行われてきた．特に，論理型に関数型を融合する研究においては，論理型プログラム中で手続き的制御の記述を行いたいということが研究の動機になっており[3)]，その結果，いくつかの計算モデルが提案されてきた[4)]．本研究の動機もこれと

† NTT未来ねっと研究所
NTT Network Innovation Laboratories
†† NTTサイバースペース研究所
NTT Cyber Space Laboratories
††† 電気通信大学
The University of Electro-Communications

☆ 実際，本研究で提案する言語TAOは並行計算機能を持つ．

同じであるが，その目的は新たな計算モデルを構築することではない．むしろ，上記の要求(1)，(2)を満たすために，Lispをベースとし，そのフレームワークを崩さないことを大前提としている．すなわち，本研究の目的は，Lispで手続き的な動作を記述し，その中から論理型言語の機能を自然に使える言語を設計することである．

ここでのLispとは，一般的な関数型計算とともに，`catch/throw`などを含む大域脱出，`block/return-from`などを含む静的制御，さらに文献1)で導入された実時間制御などの豊富なプリミティブをあわせ持った言語の総称である．論理型言語とこのような意味でのLispとの融合は，従来ほとんど試みられていなかった．我々は，このような新しいタイプの融合のためのアプローチとして，実行機構のレベルに着目した．以降では，実装方式がほぼ確立しているPrologを論理型言語の代表と考える．LispとPrologの実行機構は類似する点が多々あるが，傾向としてLispよりもPrologの方が言語要素の抽象度が高い．たとえば，Prologの単一化はLispの代入や比較といった複数の実行機構からなり，後戻りは自動的な継続の生成と大域脱出などからなる．

そこで，Lispの実行機構に対応するレベルのPrologの実行機構を言語要素として独立させ，Lispと融合する．このアプローチには，以下のような利点がある．1) Lispの実行機構は保持され，Lisp部分については言語としても変更を受けない．Prologは言語としては変更を受けるが，同等の実行機構は保持されるので，Prologと等価なプログラミングが可能である．2) Prologよりも細かいレベルでのプログラミングが可能な新しい論理型言語を構成できる．3) LispとPrologのプリミティブ間の対応を利用した融合が行える．4) 対応する機能がない場合には，それを補うことで自然に言語が拡張できる．

なお，本論文では，計算の実行機構のレベルに注目してはいるが，実装に関しては触れない．これは，Lisp，Prologともに実装方式がほぼ確立されており，実装に触れなくとも実行機構のレベルにおいて，議論が可能だからである．本論文は以下のように構成される．2章で設計方針について述べ，3章で論理型計算部分の概要について述べる．4章では，本研究で提案する融合型言語独自の機能を用いた例題を示し，5章で他の研究との比較検討を行う．

2. 設　　計

我々の目標は，実世界の問題を発見的プログラミングにより解くための言語およびシステムを提供することである．このために，いわゆる動的な性質を重視した言語TAO[5]を設計・実装中である．具体的には，以下のような意味でTAOは動的な言語である．

- 静的に型付けされていない．
- インタープリタ方式の実行．すなわち，`eval`などのメタ操作が実行可能であり，対話的なプログラム構築が可能．ただし，インタープリタ方式で実装することを規定するものではない．
- 動的変数（CommonLispのdynamic variableに相当）や動的実行制御など動的な意味を持つプリミティブを豊富に提供．
- メッセージ送信メタファーによるオブジェクト指向計算を融合．すなわち，メッセージ送信の方が総称関数（generic function）によるオブジェクト指向計算よりも，動的なメソッド検索に適するという意味での動的特性．

以降では，これらの動的な枠組みを前提として，実行機能のレベルで論理型計算を融合することの問題を考える．

2.1 関数型と論理型言語の実行機構

我々は，LispとPrologとの間には実行機構レベルにおいて，次の対応があると考える．

Lisp		Prolog
（状態）変数	$\Longleftrightarrow$	論理変数
関数呼び出し	$\Longleftrightarrow$	述語呼び出し
変数束縛，比較，代入	$\Longleftrightarrow$	単一化
`cond`など	$\Longleftrightarrow$	節選択（浅い後戻り）
`catch/throw`	$\Longleftrightarrow$	（深い）後戻り

状態変数（ここでは，Lispの変数は代入により状態を変えるという意味でこのように呼ぶ）と論理変数とは，次のような相違がある．論理変数のスコープは静的に1つの節に閉じており，スコープがネストすることもない．論理変数には値が未定義という状態がある．しかし，値の束縛という基本機構は同じであり，融合型言語では統一的に扱うべきである．

Lispにおいては，関数の仮引数が呼び出し側の値に束縛されることと，破壊的な代入とは異なる機構である．一方，Prologでは述語ヘッドで初出の変数が値を受け取ることと，呼び出し側の変数に値が代入されることとは，単一化によって統一的に行われる．TAOでは，関数と述語の対応をとるため，述語の引数渡しの機構として，単一化でなくパターンマッチを用いる．パターンマッチでは，変数を束縛するだけで破壊的な代入は生じない．単一化は，述語の本体でのみ可能と

する．これにより，関数と述語の引数渡しが束縛の機構だけで構成されることになり（より正確には等値判定機構も含まれるが後述），結果としてデータの流れが明確になる．

浅い後戻りと深い後戻りは，実行機構が異なる．前者は実行すべき節を選択するもので，静的範囲の実行制御である．この意味で，Lispの`cond`や`if`などの制御構文に近い．一方，深い後戻りは動的に移動先が決定されるという意味で，Lispの動的大域脱出（`catch/throw`）に相当する．これら2つの後戻りは，実行機構の観点やLispのプログラミング技法の観点からは異なるものであり，融合型言語においても異なるものとして扱う．

2.2 Lispへの論理型言語の融合の問題点

実行機構レベルでの，このような融合を行ううえで，次の問題点がある．

- 論理変数と状態変数をどのように統一するか．また，Prologに特有のデータである未定義値や，未定義値どうしのリンクをどのようにLispで扱うか．
- Lispでは関数はファーストクラスデータであるが，述語をどのように扱うか．
- Lispでは，`catch/throw`，`block/return-from`，`with-open-file`などのように，静的あるいは動的に囲まれた有効範囲がきわめて重要な意味を持つ．一方，Prologにはこれに対応するものはない．これをどのように両立させるか．
- Lispは副作用を前提とした言語であり，Prologは基本的に副作用のない言語である．この違いをどのように融合するか．

3. TAOの論理型計算の概要

本章では，論理型計算の概要について述べ，上記の問題をTAOでどのように解決したかを説明する．

3.1 文法と意味

論理型計算に関係する部分の文法を**図1**に示す．

通常の関数呼び出しとは異なる評価方法をとる式（Common Lispのspecial form）は，一般には「構文式」，個別には「～構文」と呼ぶ．構文式の意味を指示するために，その式中にキーワード（コロンで始まるシンボル）を第1要素とするリストを書ける．このリストをトレインと呼ぶ．トレインは，必須のものと任意のものがあるが，必須トレインではトレインのキーワードを省略してよい．図1においては，たとえばヘッドは必須トレインであり，`:aux`トレインは任意トレインである．本論文でも適宜省略する．

論理型計算の主な機能についての操作的な意味を付

```
プログラム ::= 式+
式 ::= 定数 | 変数 | 作用素呼び出し式 | メッセージ式
     | 構文式 | 代入式 | 単一化式
定数 ::= シンボル | 文字列 | 数字 | #t | #f
変数 ::= シンボル
構文式 ::= 作用素生成構文 | 作用素定義構文
作用素定義構文 ::= (define シンボル 作用素生成構文)
作用素生成構文 ::= 関数生成構文 | 述語生成構文
関数生成構文 ::= (op 仮引数定義 式+)
述語生成構文 ::= {op 節+}
節 ::= ([:clause] ヘッド [(:aux 局所変数宣言+)] ガード
       [(:choice [名前])] [(:before-fail 式)]
       ボディ*)
ヘッド ::= ([:head] Hパターン*)
ガード ::= ([:guard] 式)
ボディ ::= 式 | (:on-fail 式)
Hパターン ::= 定数 | _変数 | (Hパターン . Hパターン)
Bパターン ::= 定数 | _式 | (Bパターン . Bパターン)
作用素呼び出し式 ::= 関数呼び出し式 | 述語呼び出し式
関数呼び出し式 ::= (関数名 式*)
述語呼び出し式 ::= {述語名 Bパターン*}
メッセージ式 ::= 関数メッセージ式 | 述語メッセージ式
関数メッセージ式 ::= [式 (メッセージ名 式*)]
述語メッセージ式 ::= [式 {メッセージ名 式*}]
代入式 ::= (!式 式)
単一化式 ::= {! Bパターン Bパターン}
```

図1 TAOの主な文法
Fig. 1 Syntax of TAO.

録に記述した．以下の節においては，主に特徴的な点について述べる．

3.2 データ

TAOは多くのデータタイプを持つが，本論文では以下のデータのみを考える．

- 即値データ：シンボル，数値，*undef*，ブール値（`#t`, `#f`）
- 構造を持ったデータ：コンス，オブジェクト，配列

ここで，*undef*は未定義値を表すデータである．変数および配列の要素の既定初期値は*undef*である．記号_の評価値は*undef*であるが，記号_はいくつかの意味がオーバーロードされている．このため，以降では，未定義値を表すデータとして*undef*を用いる．一方，ブール値は，表記（`#t`, `#f`）とデータとを同一視する．

3.3 代入と単一化

状態変数と論理変数の機構上の最大の相違点は，論理変数ではリンクと呼ばれる操作が可能なことである．すなわち，未定義値の論理変数どうしを単一化すると，一方からもう一方にリンクが張られる．これにより，一方の変数に値が代入されると，もう一方の値も自動的に定まる．これは，Lispにはない機構であるが，TAOには通常のLispの代入の記述力を向上させるために導入された，場所と呼ばれる機構がある．こ

れを用いて状態変数と論理変数とを融合する．

TAOでは，変数は場所に束縛され，場所にはデータが格納される．また，構造を持つデータでは，その要素ごとに独立した場所がある．場所はメモリ中の番地に相当する概念であるが，ファーストクラスのデータではない．場所は，変数の評価および構造データ要素をアクセスする組み込み関数（car，cdrなど）の実行をした結果，本来の評価値に付随して返される．返された場所を扱えるのは代入と単一化だけであり，それ以外の場合は，場所が付随していても値の部分しか参照できない．

代入式：

(!式$_1$ 式$_2$)

を実行すると式$_1$と式$_2$が順に評価され，式$_2$の返した値が式$_1$の返した値に付随する場所に代入される（場所が付随していなければエラー．また代入式の値は式$_2$）．関数呼び出しがネストした場合でも，場所は付随したまま返される．たとえば，次のような代入が可能である．

```
(defun foo (x) (car x))
(let ((x (cons 'a 'b)))
   (!(foo x) 'c)
   x ) → (c . b)
```

場所は，変数束縛の場合はその変数束縛が確立されている期間でのみ有効な動的存続期間（dynamic extent）を持ち，構造データの場合は無限存続期間（indefinite extent）を持つ．たとえば，

```
(!(let ((x 1)) x) 1)
```

このプログラムでは，xを束縛した場所は，let式の終了とともに存続しなくなっておりエラーとなる．

単一化式：

{! Bパターン$_1$ Bパターン$_2$}

を実行すると，Bパターン$_1$とBパターン$_2$が単一化される．その際，これらの中に_で始まる式（これを強制評価式と呼ぶ）があると，_を除いた部分の式が評価され，その値が新たに単一化の対象となる．例を示す．

```
(let (x y z)    ; x, y, z の初期値は undef
   {! _x 1}             ; x に 1 を代入
   {! (a _y) (a 1)}     ; y に 1 を代入
   {! _z _(1+ x)} )     ; z に 2 を代入
```

一般の論理型言語と同様に，単一化は2つのパターンが同じになるように代入を行う．そのような代入が存在しない場合，単一化は失敗するが，これについては後述する．代入は，*undef*とデータを単一化しようとした場合に起こる．*undef*をパターンとして直接書くことはできないため，*undef*は，強制評価式の値としてしか現れない（単独で_と書くと，Prologの無名変数と同じ意味となる）．

強制評価式が*undef*を返した場合には，付随して返された場所に代入が起こる．ただし，この代入は後で取り消されうるので，これを特に可塑代入と呼ぶ．*undef*と*undef*を単一化した場合には，一方の場所に，もう一方の場所が格納されリンクが張られる．リンク元へのすべてのアクセスにおいて，自動的にリンクがたどられる（いわゆるdereference処理）ため，ユーザに場所が陽に見えることはない．

代入と可塑代入とでは，場所に関する動作がわずかに異なる．可塑代入では，*undef*が存在する場所の値を書き換えるが，代入では必ず指定された場所を書き換える．たとえば，

```
(let (u v x y)
   {! _u _v}          ; (1)
   {! _u 1}           ; (2)
   {! _x _y}          ; (3)
   (!x 1) )           ; (4)
```

(2)の結果，vの値は1となる．一方，(4)の結果，yの値は*undef*のままか，1になるかは分からない．x，yに束縛された場所をそれぞれ場所$_x$，場所$_y$とする．(4)では，必ず場所$_x$に代入されるため，(4)の結果は，(3)によって場所$_x$から場所$_y$へリンクが張られたか，その逆かに依存する．ユーザは，これらのことに依存したコードを書くべきではない．

（状態）変数と論理変数の統合により，TAOの提供する静的変数，動的変数，大域変数，コンテキスト変数（プロセス間共有のための変数）といった多種類の変数が，論理型プログラミングからも扱えるようになった．また，場所の機構により，たとえば

```
{! _(aref array n) 3}
```

あるいは，先の例に定義されたfooを用いて，

```
{! _(foo x) c}
```

などが可能となるように，単一化が拡張される．

3.4 関数と述語

TAOでは関数と述語はともにファーストクラスデータであり，両者をあわせ作用素と呼ぶが，データ型としては関数と述語は異なる．それぞれは独自の呼び出し形式を持つ．述語と関数を同じデータと見なし，同じ構文で呼び出すような仕様としなかったのは，次の2つの理由による．1つは，後述するように両者の引数に対する意味，つまり処理方法が異なる点である．同一の構文としてしまうと，呼び出す関数が実行時に決まるapplyなどのプリミティブにおいては，静的に

引数処理の方法が決められないため，実行時に重い処理が必要となる．もう1つは，プログラムスタイルの問題である．TAO_{86}[6]は，同じ構文で呼び出すタイプの言語であるが，著者らが実際にプログラムしてみると，関数型プログラムと論理型プログラムのスタイルはまったく異なるものであり，局所的に関数呼び出しか述語呼び出しかが判別できないと可読性が著しく低下することが分かった☆．

関数は，関数生成構文によって生成する．これはCommon Lispの`lambda`に，構文的（つまり`op`を`lambda`に置き換える）にも意味的にもほぼ対応する☆☆．一方，述語は述語生成構文で生成する．たとえばリストの連結は，次のように記述する（ただし，この例では，第1引数はリストでなければならず，いわゆる入出力が逆方向の計算はできない）．

```
(define append
  {op ((() _x _y) (:guard #t)
       {! _x _y} )
      (((_a . _x1) _y _z) (:aux z1)
       (:guard #t)
       {! _z (_a . _z1)}
       {append _x1 _y _z1} )})
```

関数の呼び出しでは，引数の各式が順に評価され，それらの値が関数に渡される．一方，述語の呼び出しでは，引数に対し構造コピーという操作がなされる．構造コピーは，Bパターンという構文上の表現からデータ構造を生成する操作である．Bパターンがリスト（の構文）であった場合は，その構文上の要素が再帰的にコピーされ，その結果をリストにして返す．Bパターンが定数の場合にはその値がコピー結果となり，強制評価式であった場合には，その式が評価され，その値がコピー結果となる．構造コピーは，Prologでも行われている処理であるが，実装技法の1つであり，言語の意味論に陽に現れることはなかった．一方，TAOでは評価のタイミングと評価の環境を明確にするため，これを陽に扱う．すなわち，コピーはテキスト上の出現順に左から右に行われ，評価の環境は呼び出し側の環境である．これにより，副作用を持った式の実行環境も正確に定義される．

構造コピーが終わると述語が呼ばれ，引数と最初の節のヘッドとをパターンマッチしながら，変数束縛を確立してゆく．データの型あるいは値の不一致があるとパターンマッチは失敗する．呼び出し側が*undef*であっても，それへの代入は起きず，ヘッドが初出の変数の場合にのみ（例外ケースについては後述）成功し，その変数の場所と*undef*のある場所とがリンクされる．

パターンマッチが成功すると，`:aux`宣言に従って変数の束縛を行い☆☆☆ガードを実行する．ガードが`#f`以外の値を返すとボディが実行される．パターンマッチに失敗したり，ガードが`#f`を返したりすると，その節の選択は失敗したといい，次の節に対して上記の操作を繰り返す．節選択がすべて失敗するとエラーとなる．Prologとは異なり（深い）後戻りは自動的に起動されない．ボディが最後まで実行されると，ボディ中の最後の式が返した値をそのまま返す．ボディのない節は`#t`を返す．

なお，ヘッドに同じ変数が複数回現れてもよい．パターンマッチのとき，2回目以降の出現においては，最初の出現で確立された束縛値を用いて比較を行う．この比較は，真に同一のデータかを判定する（Lispの`eq`に相当）のみであり，構造データを分解して比較することはしない．上に述べた，呼び出し側が*undef*でヘッドが初出の変数以外でも成功する例外ケースとは，同じ場所を比較した場合である．つまり，

```
(define eq {op ((_x _x) (:guard #t))})
(let (a) {eq _a _a}) → #t
(let (a b) {eq _a _b}) → エラー
```

となる．2番目の`let`では，異なる場所を比較したことになり，パターンマッチは失敗する．なお，

```
(let (a b) (eq a b)) → #t
```

である．`eq`は，場所の情報を知ることはできず，値のみの比較を行い，*undef*どうしを同値と判定する．

ヘッドでの変数の複数回の出現が許されること，`:aux`宣言によって補助変数が宣言できること，ガードがあること，といった設計は，ボディの実行に入る前に引数を調べる能力を上げるためのものである．つまり，後で後戻りをして節を探すという可能性をより低くするためである．また，節選択が失敗したときの状態を戻す処理が軽く（具体的には局所変数束縛の解除だけに）なるように設計されている．

3.5 大域脱出としての後戻り

後戻りは実行時に「チョイス」と呼ばれる一種の継続を生成し，後でそのチョイスに戻ってくるという機構である（なお，`catch`が作る継続はキャッチと呼ぶ）．チョイスが，キャッチと異なるのは，戻った後で計算

☆ 結局，TAO_{86}では，名前の付け方で関数と述語とを区別するようにした．

☆☆ 正確には，`op`は`lambda`のように外部の静的環境を参照することはできない．`op`のほかに，`op*`，`op@`というスコープルールの異なる3種類の生成構文がある．

☆☆☆ Common Lispの`&aux`にほぼ同じ．

を再実行するために，チョイス生成以降になされた計算を取り消す点である．このようなチョイスの実行機構は，Lispの「囲み」の考え方に適合しにくい．ここで囲みとは，静的有効範囲（lexical scope）あるいは動的存続期間（dynamic extent）によって表現される，束縛の有効領域の総称である．囲みには，`block`などの静的な囲み，`catch`などの動的な囲み，さらにTAOで独自の囲みとして，並行計算におけるクリティカルセクション，実時間計算におけるタイムアウト機構などがある．

囲みの中でチョイスが生成され，囲みをいったん抜けたとする．囲みを抜けること自身が重要な意味を持つため，その後でこのチョイスに戻るために，抜けたことを取り消すことは難しい．たとえば，TAOのタイムアウト構文は，その本体がタイムアウト期間内に実行を終了するとタイマを停止する．もし，その本体中でチョイスが生成されタイマ停止後にチョイスに戻ったとすると，タイマ計測をどのように再開すべきかは難しい問題である．これを解決するには，いくつかの方法が考えられる．たとえば，囲みの種類ごとにチョイスで戻った場合の正確な意味を与える，あるいは，このようなチョイスの使い方をユーザの責任としてしまうなどである．しかし，前者では，言語の意味が複雑になるし，すべての囲みに無矛盾で使いやすい意味を与えられるか疑問である．一方，後者ではユーザの負担が増加する．そこで，TAOでは，チョイスの生成と存続期間を以下のように制限することにより，この問題が生じないようにしている．

チョイスは，`:choice`トレインが指定された節において，節の選択が成功したときにのみ生成され，その次の節を継続として持つ．トレインの要素の評価値がチョイスの名前となる．

チョイスの呼び出しは，

```
(fail タグ)
```

という式を実行することによってなされる．タグは評価され，それが *undef* の場合は最も新しいチョイスが，それ以外の場合はそれを名前とするチョイスが探される．そのチョイスが生成されたときまで溯って可塑代入が元に戻され，そのチョイスの持つ継続に制御が移る．

チョイスの存続期間について正確に規定するために，評価のモードを導入する．評価には，$\mathcal{L}$ 評価と $\mathcal{P}$ 評価がある．節のボディの式のみが $\mathcal{P}$ 評価され，それ以外のすべての式は $\mathcal{L}$ 評価される．$\mathcal{L}$ 評価と $\mathcal{P}$ 評価は述語呼び出しの意味のみが異なる．$\mathcal{P}$ 評価は，Prologの述語の実行と同じであり，述語を $\mathcal{P}$ 評価すると，その実行が終了してもチョイスが残される．

一方，述語呼び出し式 E を $\mathcal{L}$ 評価すると，次のように実行される．1) *undef* という名前のチョイス（これは暗黙のチョイスと呼ばれる）が生成される．その継続は，`#f` を E の値として返すものである．2) E を $\mathcal{P}$ 評価する．3) 暗黙のチョイスを含め，それより新しいチョイスをすべて削除し，`#t` を主値，ステップ2の評価値を副値とする多値を返す．例で示す：

```
(define foo
  {op ((a) (:guard #t) 'end)
      ((b) (:guard #t) (fail _☆)) })

(let () {foo a}) → #t,end ; 多値
(let () {foo b}) → #f
(let () {foo a} (fail _)) → エラー
```

ステップ3でのチョイスの削除は，述語の $\mathcal{L}$ 評価が決定的に成功したと見なされることを意味する．囲みと後戻りの問題は，囲みの中では $\mathcal{L}$ 評価しかしないように，言語を設計することで解決される．つまり，囲みの中でチョイスが作られても $\mathcal{L}$ 評価終了時に削除されてしまい，囲みを抜けた後で戻ってくることはできない．

また，ガードが $\mathcal{L}$ 評価されることに注意されたい．ガードは，引数の値の検査だけをするために設けられており，検査の成功/失敗を一意に決め，チョイスを残すべきではない．すなわち関数で十分なはずであるから，これは $\mathcal{L}$ 評価される．

3.6 Lispの大域脱出との関係

論理型プログラムからLispを呼び出すことの利点の1つは，Lispにおいて副作用をともなった処理を行える点にある．ただし，論理型プログラムが後戻りをする際には，その副作用を適切に戻す必要がある．可塑代入の取り消しは，システムが自動的に行うが，それ以外については陽に行わなければならない．後戻りを大域脱出と見なすことで，これをLispの後始末処理（Common Lispのclean-up formに相当）の機能を用いて行うことが可能となる．しかし，Lispの大域脱出機構を利用するには，次のような問題がある．

1つは，`unwind-hook`（Common Lispにおける`unwind-protect`に相当）の囲みから脱出するときに，後始末処理をするというLispの機構が，前節と同様の理由から後戻りと適合しにくい点である．そこで，TAOでは，後戻りのときに後始末関数を実行する機構を，`:on-fail`トレインの形で提供する．`:on-fail`

☆ _の評価値は *undef* であったことに注意．

トレインが実行されると，その要素が評価される．その結果は，無引数の関数でなければならず，これがシステムに登録される．この関数は，**:on-fail** トレインよりも前のチョイスに後戻りするときに自動的に実行される．通常の Lisp では，後始末をフォームによって行うが，TAO では関数で行う．これは，後戻りの途中という通常の Lisp にはない状態で後始末が呼ばれるためである．つまり，ある述語のボディの実行中に後始末関数が登録されたとすると，その述語が終了しても後始末関数が実行される可能性があるのである．もし，後始末をフォームで行うとすると，このような事態に対処するため，フォームを実行するための環境をすべて保持しなければならない．なぜなら，フォームはすべての環境にアクセスしうるからである．一方，関数は，それが閉じ込める環境が制限されているため（たとえば **op** は静的環境をいっさい閉じ込めない），実現が容易である．

もう 1 つの問題は，これまで一種類（**throw**）だけだった大域脱出が多種類になったことである．これをプログラム上で区別するため，**throw** で大域脱出するときだけ関数を呼び出す **throw-hook** 構文も用意されており，これらをまとめると**表 1** のようになる．○は呼び出すことを，×は呼び出さないことを示す．また，可塑代入については，**:on-fail** を用いて代入を元に戻すと考えると，この分類が適用できる．

TAO では，エラーも大域脱出と考える．エラーが生ずると特定のタグを持つ **throw** または **fail** が起動される．すべての節選択が失敗したときのエラーでは，

```
(fail :undefined-predicate)
```

という式が実行される．また，単一化の失敗というエラーでは，**(fail _)** という式が実行される．原則として論理型プログラムに関するエラーでは **fail** が，それ以外では **throw** が実行されるが，チョイスやキャッチの継続に制御を移す前に関数を実行する機構（チョイスの **:before-fail** トレイン，キャッチの **:before-throw** トレイン）を用いて意味を変えることは可能である．

3.7 オブジェクト指向計算と単一化の拡張

TAO のオブジェクト指向計算は，いわゆるプロトタイプに基づくものである[7]．これに関する詳細な議論は本論文の範囲外であるので，ここでは簡単に説明する．

関数と述語を，メソッドとしてオブジェクトに登録することができる．それぞれのメソッドには，独自の呼び出し形式がある（図 1 の関数メッセージ式と，述語メッセージ式）．述語メッセージ式は，述語呼び出し式と同様の意味を $\mathcal{P}$ 評価に関して持つ．委託（delegation）などを準備するための機構も，関数メソッドと述語メソッドに対称的な形で準備されており，さまざまな拡張を行うことができる．

オブジェクトに関する単一化を，ユーザが定義することができる．オブジェクトが（*undef* 以外の）任意のデータと単一化された場合，次のような述語メッセージ式が $\mathcal{P}$ 評価される．

```
[オブジェクト {unify: データ}]
```

ユーザ定義の 1 引数の述語メソッドを，**unify:**☆の名で登録することにより意味を変更できる．

この式が $\mathcal{P}$ 評価される理由は，この述語メソッドの実行中に可塑代入が起きる可能性があり，これを後戻りで取り消すことができるようにするためである．これによる副次的効果として，チョイスを残すことができるため，非決定的な単一化を構築することも可能である．

4. 例

ここで，人名の読みから別の文を作り出すというクイズを考える．例：

```
(quiz ("や" "ま" "ざ" "き")) → ("や" "き"
 "ざ" "ま")
```

このクイズを，メモリを無駄に消費せずに解くプログラムを考える（**図 2**）☆☆．

ここで **curry** は，関数に（全部あるいは一部の）引数を与えて，より少ない引数をとる関数を作り出す関数である．ビットテーブル操作関数の意味は自明であろう．なお，ビットテーブル操作は，構造データの破壊的書き換えであるから，それを共有してさえいれば，どこからでも参照可能である．

関数 **quiz** に，人名をリストで渡すと，**permutation** による入れ換えリストが **answer** に返され，それを形態素解析によって採点する．ある点数以下であれば，**(fail _)** によって後戻りし，別の候補を探す．関数 **permutation/select** は，よく知られたリストの順

表 1 大域脱出と後始末関数の関係
Table 1 Difference between non-local exit and cleanup function.

	unwind-hook	:on-fail	throw-hook
throw	○	×	○
fail	○	○	×

☆ ：で終わるシンボルは，標準的なメッセージ名を表すキーワードの一種である．

☆☆ むろん，ここで無駄がないとは，最適という意味ではない．素直に書いたプログラムを段階的にリファインしてメモリ効率を上げるといった過程を想定している．

```
(define quiz
  {op ((_list )
       (:aux (len (length list)) answer)
       (:guard #t)
       {permutation
          _list _len
          _(make-bit-table len) _answer}
       (if (< (morphological-analyze
                answer)
              0.7)
           (fail _) )          ; 後戻りの起動
       answer )})

(define permutation
  {op ((_ 0 _ _answer)
       (:guard #t)
       {! _answer ()} )
      ((_list _n _bitTbl _answer)
       (:aux x rest)
       (:guard #t)
       {select _list 0 _bitTbl _x}
       {! _answer (_x . _rest)}
       {permutation _list _(1- n)
                    _bitTbl _rest} )})

(define select
  {op ((() _pos _bitTbl _)
       (fail _) )      ; 後戻りの起動
      (((_x . _) _pos _bitTbl _answer)
       (:guard (bit-off?  pos bitTbl))
       (:choice)       ; チョイスの生成
       (bit-set pos bitTbl)
       (:on-fail
          (curry
             (op (pos bitTbl)
                 (bit-reset pos bitTbl) )
             pos bitTbl))
       {! _answer _x} )
      (((_ . _list) _pos _bitTbl _answer)
       (:guard #t)
       {select _list _(1+ pos)
               _bitTbl _answer} )})
```

図 2 人名クイズを解くプログラム
Fig. 2 A quiz solver.

列/選択のプログラムと機能的には同じであるが，内部で無駄なセルを消費しないようにするため，その文字が使用済みかどうかを，ビットテーブルを使って表現している．また，チョイスを作るのは，`select` の第 2 節だけであること，後戻りをするのは，採点後の `fail` と選択すべき文字がない場合の `fail`（`select` の第 1 節）だけであることが明示されている．

5. 関連研究

副作用まで含む関数型言語と，論理型言語を融合した試みは，あまり多くないが，主なものに Leda[8]，Poplog[9]，TAO_{86}[6] がある．

Leda は，オブジェクト指向，論理型の記述ができる Pascal 風の手続き型言語で，単一化，後戻り，さらには取り消し可能な代入もユーザ定義ができる．Leda は，論理型計算のメカニズムを提供するというより，むしろ論理型言語の実装にも使えるプリミティブを提供する言語というべきものである．Poplog も，これに類似した言語で，Pop-11 をベースとしていくつかの言語を融合している．Prolog と Pop-11 間は互いに呼び合うことができるが，未定義変数を作ったり，dereference をしたりするのは，Pop-11 プログラムで陽に行う．プログラマは Prolog のデータがどのように表現されているかを知らなければならない．一方，TAO では，データも変数も統合されており，Prolog のデータが Lisp 上に表現されているわけではない．プログラマは内部表現を知る必要はない．また，単一化，後戻りはあらかじめ定義されており，その一部の意味をユーザが変更する機構が用意されている．

TAO は，TAO_{86} を元に設計された言語であり，類似点も多いが，TAO_{86} では次の問題があった．1) 論理変数と状態変数が分離していた．2) 場所の概念はあったが，代入に対してのみ意味を持つものであり単一化では使えなかった．3) 前述のように，述語と関数を同じ構文形式で扱っていた．4) 論理型計算の制御は Prolog と同様のものが組み込まれており，変更することはできず，また Lisp の実行制御とは独立していた．

文献 10) では，Scheme 上に継続を利用して Prolog の制御を実装している（これは Poplog でも同様）．継続をプログラムで扱えるようにしてしまうと，継続の生成と後戻りが無制限に可能となり，前述の「囲みと後戻りの問題」が発生する．TAO では，チョイスの生成と存続期間を制限することにより，この問題を回避している．また，本論文では触れなかったが，TAO の論理型計算機構は WAM の一部を修正するだけで実現可能であり，同程度の効率を達成できると予想される．一方，継続を特定の場合に効率良く実現することはできるが[11]，無制限に可能とした場合には，効率は低下すると予想される．

論理型言語として比較した場合，B-Prolog[12] の matching clause は，単一化の方向の分離，決定性の陽な記述，ガードの導入という意味で，TAO と類似している．ガードは，並行論理型言語で主に用いられてきたが，最近では B-Prolog のほか，いくつかの逐次型言語にも取り入れられている[13),14)]．B-Prolog では，ガードの実行でチョイスを作らないようにプログラムするのはユーザの責任であるが，TAO ではガードを $\mathcal{L}$ 評価することにより言語側で保証している．

6. まとめ

実行機構のレベルに着目した関数型言語と論理型言語の融合について述べた．Lispの実行機構に対応する論理型言語の実行機構を考え，それを言語要素として独立させて，Lispとの融合を図った．TAOは，場所の概念を用いた論理変数と状態変数の統合，関数と同様の述語のファーストクラスデータ化，「囲みと後戻り」の問題を回避するためのチョイスの生成方法と存続期間の制限，`:on-fail`トレインによる副作用の取り消し，といった特徴を持つ．

我々の提案する融合方法によって，それぞれのパラダイムが得た利点をまとめてみよう．論理型プログラムにとっては，1) 変数の統合により多様なスコープの変数が使えるようになり，さらに場所の概念により単一化が拡張された．2) 関数との対応付けにより述語がファーストクラスデータとなり，また無名述語も表現可能となった．3) バックトラックを大域脱出と考えることにより，Lispの大域脱出の諸機能を利用できるようになった．4) ガードやボディからLisp関数を呼び出すことにより，決定的な処理を明確化できるようになった．一方，Lispにとっては，探索型プログラムを記述することが容易になった点が最も大きいであろう．またパターンマッチや単一化の機能によって，パターン処理が容易となった．

むろん，最大の利点はこれらの機能を任意に組み合わせ1つのプログラムを構築できる点にある．これについては一例を示した．なお，現在TAOを専用マシンSILENT[15]（クロック33 MHz）上に実装中であるが，本論文執筆時点では500KLIPS以上の性能を得ている．

謝辞 本論文の草稿の段階から丁寧なコメントをいただきましたNTTの平田圭二氏，ご討論いただいた原田康徳氏，日頃からTAO/SILENTの研究をご支援いただく尾内理紀夫氏に深く感謝します．貴重なご助言をいただきました査読者の方に感謝します．

参考文献

1) 天海良治，山崎憲一，中村昌志，吉田雅治，竹内郁雄：TAOのコンカレンシ・コントロール，情報処理学会記号処理研究会，69-3 (1993).
2) 竹内郁雄，吉田雅治，山崎憲一，天海良治：実時間記号処理システムTAO/SILENTにおける軽量プロセスの実現，情報処理学会論文誌，Vol.38, No.3, pp.595–605 (1997).
3) Bellia, M. and Levi, G.: The Relation between Logic and Functional Languages: A Survey, *J. Logic programming*, Vol.3, No.3, pp.217–236 (1986).
4) Hanus, M.: The Integration of Functions into Logic Programming: from Theory to Practice, *J. Logic Programming*, Vol.19&20, pp.583–628 (1994).
5) 竹内郁雄，天海良治，山崎憲一：新しいTAOの設計，情報処理学会記号処理研究会，56-2 (1990).
6) 山崎憲一，奥乃 博，竹内郁雄：TAOにおける論理型プログラミングとその処理方式，情報処理学会論文誌，Vol.32, No.9, pp.1090–1101 (1991).
7) Yamazaki, K., Amagai, Y., Yoshida, M. and Takeuchi, I.: TAO: An object orientation kernel, *ISOTAS '93*, LNCS, Vol.742, pp.61–76, Springer-Verlag (1993).
8) Budd, T.A.: *Multiparadigm Programming in Leda*, Addison-Wesley (1995).
9) Mellish, C. and Hardy, S.: Integrating Prolog in the POPLOG environment, *Implementations of PROLOG*, Campbell, J. (Ed.), Ellis Horwood (1984). (最新版は http://www.cogs.susx.ac.uk/users/adrianh/poplog.html).
10) Srivastava, A., Oxley, D. and Srivastava, A.: An (other) Integration of Logic and Functional Programming, *Symposium on Logic Programming*, pp.254–260, IEEE (1985).
11) Bruggeman, C., Waddell, O. and Dybvig, R.: Representing Control in the Presence of One-Shot Continuations, *SIGPLAN '96: Conference on Programming Language Design and Implementation*, pp.99–107, ACM (1996).
12) Zhou, N.: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Trans. Programming Languages and Systems*, Vol.18, No.6, pp.752–779 (1996).
13) Hill, P. and Lloyd, J.: *The Gödel Programming Language*, MIT Press (1994).
14) 中島秀之：有機的プログラミングの料理法，情報処理学会夏のプログラミングシンポジウム，pp.1–8 (1997).
15) 吉田雅治，竹内郁雄，天海良治，山崎憲一：新しい記号処理カーネルSILENTの設計，情報処理学会計算機アーキテクチャ研究会，84-1 (1990).
16) Plotkin, G.: A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Aarhus Univ. (1981).
17) Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, MIT Press (1990).

付録

A.1 TAOの論理型プログラム部分の操作的意味論

この意味論は論理型計算の主要部分を記述すること

を目的としており，紙面の制限から，**:on-fail** トレイン，**:before-fail** トレイン，Lispの中での大域脱出については記述しない．また可塑代入の取り消しを明示的に記述していないため，Lispの副作用との関係に関する操作的意味が十分に明確にはなっていない．

基本的なスタイルは文献16)に従い，一部の定義（$+$，*Dom* など）は，文献17)による．特に注意すべき記法としては以下のとおり．

- $\langle \ldots \rangle$ は組を表す．
- 構文上の並びは * で表し，演算子 :: によって連結されるものとする．この記法は，意味データなどにも使う．空並びは，$\diamond$ で表す．

意味データ（Semantic Data）を示す．

$$
\begin{aligned}
c &\in \mathit{Literal} = \mathit{Atom} \cup \mathit{Boolean} \\
o &\in \mathit{Object} \\
l &\in \mathit{Location} \\
(l_1 \ . \ l_2) &\in \mathit{Cons} = \mathit{Location} \times \mathit{Location} \\
b &\in \mathit{BasicValue} = \mathit{Literal} \cup \mathit{Cons} \cup \mathit{Object} \\
u &\in \mathit{Undef} = \{\mathit{undef}\} \\
v &\in \mathit{Value} = \mathit{BasicValue} \cup \mathit{Undef} \\
a, d &\in \mathit{Data} = \mathit{Value} \cup \mathit{Location} \\
[d_1, d_2, \ldots] &\in \mathit{MultipleValue} = \mathit{Data} \times \ldots \times \mathit{Data} \\
\langle\!\langle v, l \rangle\!\rangle &\in \mathit{ValLoc} = \mathit{Value} \times \mathit{Location}
\end{aligned}
$$

MultipleValue は多値を表す．*ValLoc* は値とそれに付随した場所のペアを表す．本文でも述べたとおり，場所がともなった値は特定の構文以外では値として扱われる．また，この意味論では構文的に正しいプログラムのみを扱う．式全体のクラスを *Exp* と表し，各式は *exp*，*aux* などの3文字以上の変数で指す．Configurationは，次のように定義される．

$$\Gamma_{\mathcal{P}} = \{\langle e, \rho, \sigma, \kappa, \tau \rangle | e \in \mathit{Exp} \cup \mathit{Data},\ \rho \in \mathit{Env},\ \sigma \in \mathit{Store},\ \kappa \in \mathit{Cont}^*,\ \tau \in \mathit{Choice}^*\}$$

$$\Gamma_{\mathcal{L}} = \{\langle e, \rho, \sigma \rangle | e \in \mathit{Exp} \cup \mathit{Data},\ \rho \in \mathit{Env},\ \sigma \in \mathit{Store}\}$$

ここで，

$$
\begin{aligned}
\mathit{Env} &= \mathit{Atom} \to \mathit{Location} \\
\mathit{Store} &= \mathit{Location} \to \mathit{Data} \\
\langle \mathit{exp}, \rho \rangle &\in \mathit{Cont} = \mathit{Exp} \times \mathit{Env} \\
\langle d, \mathit{exp}, \rho, \sigma, \kappa \rangle &\in \mathit{Choice} = \mathit{Data} \times \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Cont}^*
\end{aligned}
$$

遷移規則は，一般的には次のようにモード μ をともなう（μ は $\mathcal{L}$ または $\mathcal{P}$）．

$$\frac{\vdash_{\mu'} \Gamma_{\mu'} \to \Gamma_{\mu'} \quad \ldots \quad \vdash_{\mu''} \Gamma_{\mu''} \to \Gamma_{\mu''}}{\vdash_{\mu} \Gamma_{\mu} \to \Gamma_{\mu}}$$

記述の簡略化のため，構文に関して以下の記法を用いる．

detclause : (:clause (:head *hpat**) (:aux *aux**) (:guard *guard*) *body**)
ndetclause : (:clause (:head *hpat**) (:aux *aux**) (:guard *guard*) (:choice *choice_tag*) *body**)
bothclause : (:clause (:head *hpat**) (:aux *aux**) (:guard *guard*) ...)
pexp : 述語呼び出し式と単一化式
lexp : *pexp* 以外の式

記述量を減らすため，*Literal* は，表現と意味データを同一視する．遷移規則を以下に示す．

[決定的成功]

$$\frac{\begin{array}{l}\vdash_{\mathcal{L}} \langle (\text{:head } \mathit{hpat}^*\ a^*), \rho_0, \sigma_0 \rangle \to \langle \#t, \rho_1, \sigma_1 \rangle \\ \vdash_{\mathcal{L}} \langle (\text{:aux } \mathit{aux}^*), \rho_1, \sigma_1 \rangle \to \langle d, \rho_2, \sigma_2 \rangle \\ \vdash_{\mathcal{L}} \langle \mathit{guard}, \rho_2, \sigma_2 \rangle \to \langle v, \rho_3, \sigma_3 \rangle \quad v \neq \#f \end{array}}{\vdash_{\mathcal{P}} \langle \{\mathit{detclause}:: \mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle \to \langle \mathit{body}^*, \rho_2, \sigma_3, \kappa, \tau \rangle} \quad (1)$$

[非決定的成功]

$$\frac{\begin{array}{l}\vdash_{\mathcal{L}} \langle (\text{:head } \mathit{hpat}^*\ a^*), \rho_0, \sigma_0 \rangle \to \langle \#t, \rho_1, \sigma_1 \rangle \\ \vdash_{\mathcal{L}} \langle (\text{:aux } \mathit{aux}^*), \rho_1, \sigma_1 \rangle \to \langle d, \rho_2, \sigma_2 \rangle \\ \vdash_{\mathcal{L}} \langle \mathit{guard}, \rho_2, \sigma_2 \rangle \to \langle v_1, \rho_3, \sigma_3 \rangle \quad v_1 \neq \#f \\ \vdash_{\mathcal{L}} \langle \mathit{choice\_tag}, \rho_2, \sigma_3 \rangle \to \langle v_2, \rho_4, \sigma_4 \rangle \end{array}}{\vdash_{\mathcal{P}} \langle \{\mathit{ndetclause}:: \mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle \to \langle \mathit{body}^*, \rho_2, \sigma_4, \kappa, \langle v_2, \{\mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa \rangle :: \tau \rangle} \quad (2)$$

[次節の選択]

$$\frac{\vdash_{\mathcal{L}} \langle (\text{:head } \mathit{hpat}^*\ a^*), \rho_0, \sigma_0 \rangle \to \langle \#f, \rho_1, \sigma_1 \rangle}{\vdash_{\mathcal{P}} \langle \{\mathit{bothclause}:: \mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle \to \langle \{\mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle} \quad (3.1)$$

$$\frac{\begin{array}{l}\vdash_{\mathcal{L}} \langle (\text{:head } \mathit{hpat}^*\ a^*), \rho_0, \sigma_0 \rangle \to \langle \#t, \rho_1, \sigma_1 \rangle \\ \vdash_{\mathcal{L}} \langle (\text{:aux } \mathit{aux}^*), \rho_1, \sigma_1 \rangle \to \langle d, \rho_2, \sigma_2 \rangle \\ \vdash_{\mathcal{L}} \langle \mathit{guard}, \rho_2, \sigma_2 \rangle \to \langle \#f, \rho_3, \sigma_3 \rangle \end{array}}{\vdash_{\mathcal{P}} \langle \{\mathit{bothclause}:: \mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle \to \langle \{\mathit{clause}^*\ a^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle} \quad (3.2)$$

[節選択の失敗]

$$\vdash_{\mathcal{P}} \langle \{\diamond\ a^*\}, \rho, \sigma, \kappa, \tau \rangle \to \langle (\text{:fail :undefined-predicate}), \rho, \sigma, \kappa, \tau \rangle \quad (4)$$

[ボディ中での述語呼び出し]

$$\frac{\begin{array}{l}\vdash_{\mathcal{L}} \langle (\text{:copy } \mathit{bpat}_0), \rho_0, \sigma_0 \rangle \to \langle a_0, \rho_1, \sigma_1 \rangle \\ \ldots \\ \vdash_{\mathcal{L}} \langle (\text{:copy } \mathit{bpat}_{n-1}), \rho_0, \sigma_{n-1} \rangle \to \langle a_{n-1}, \rho_n, \sigma_n \rangle \\ \mathit{def\_of}(\mathit{pred}, n) = \mathit{clause}^* \end{array}}{\vdash_{\mathcal{P}} \langle \{\mathit{pred}\ \mathit{bpat}^*\}, \rho_0, \sigma_0, \kappa, \tau \rangle \to \langle \{\mathit{clause}^*\ a^*\}, \phi, \sigma_n, \kappa, \tau \rangle} \quad (5.1)$$

$$\frac{\begin{array}{l}\vdash_{\mathcal{L}} \langle (\text{:copy } \mathit{bpat}_0), \rho_0, \sigma_0 \rangle \to \langle a_0, \rho_1, \sigma_1 \rangle \\ \vdash_{\mathcal{L}} \langle (\text{:copy } \mathit{bpat}_1), \rho_0, \sigma_1 \rangle \to \langle a_1, \rho_2, \sigma_2 \rangle \end{array}}{\vdash_{\mathcal{P}} \langle \{!\ \mathit{bpat}_0\ \mathit{bpat}_1\}, \rho_0, \sigma_0, \kappa, \tau \rangle \to \langle \{!\ a_0\ a_1\}, \phi, \sigma_2, \kappa, \tau \rangle} \quad (5.2)$$

[ボディの実行]

$$\vdash_{\mathcal{P}} \langle \mathit{exp} :: \mathit{rest}, \rho, \sigma, \kappa, \tau \rangle \to \langle \mathit{exp}, \rho, \sigma, \langle \mathit{rest}, \rho \rangle :: \kappa, \tau \rangle \quad (6)$$

[単位節]

$$\vdash_{\mathcal{P}} \langle \diamond, \rho_0, \sigma, \langle \mathit{exp}, \rho \rangle :: \kappa, \tau \rangle \to \langle \mathit{exp}, \rho, \sigma, \kappa, \tau \rangle \quad (7)$$

[最後の事実節]

$$\vdash_{\mathcal{P}} \langle \diamond, \rho, \sigma, \diamond, \tau\rangle \to \langle \#t, \rho, \sigma, \diamond, \tau\rangle \tag{8}$$

[関数の $\mathcal{P}$ 評価]

$$\frac{\vdash_{\mathcal{L}} \langle lexp, \rho_0, \sigma_0\rangle \to \langle d, \rho_1, \sigma_1\rangle}{\vdash_{\mathcal{P}} \langle lexp, \rho_0, \sigma_0, \langle exp, \rho\rangle :: \kappa, \tau\rangle \to \langle exp, \rho, \sigma_1, \kappa, \tau\rangle} \tag{9}$$

[最後の関数の $\mathcal{P}$ 評価]

$$\frac{\vdash_{\mathcal{L}} \langle lexp, \rho_0, \sigma_0\rangle \to \langle d, \rho_1, \sigma_1\rangle}{\vdash_{\mathcal{P}} \langle lexp, \rho_0, \sigma_0, \diamond, \tau\rangle \to \langle d, \rho_0, \sigma_1, \diamond, \tau\rangle} \tag{10}$$

[述語の $\mathcal{L}$ 評価]

$$\frac{\begin{array}{c}\vdash_{\mathcal{P}} \langle pexp, \rho_0, \sigma_0, \diamond, \langle u, \{\texttt{(:clause () \#t)}\}, \phi, \sigma_0, \diamond\rangle\rangle \to \\ \langle d, \rho_1, \sigma_1, \diamond, \tau_1\rangle \\ \tau_1 \neq \diamond\end{array}}{\vdash_{\mathcal{L}} \langle pexp, \rho_0, \sigma_0\rangle \to \langle [\#t, d], \rho_0, \sigma_1\rangle} \tag{11.1}$$

$$\frac{\vdash_{\mathcal{P}} \langle pexp, \rho_0, \sigma_0, \diamond, \langle u, \{\texttt{(:clause () \#t)}\}, \phi, \sigma_0, \diamond\rangle\rangle \to \langle \#t, \rho_1, \sigma_1, \diamond, \diamond\rangle}{\vdash_{\mathcal{L}} \langle pexp, \rho_0, \sigma_0\rangle \to \langle \#f, \rho_0, \sigma_1\rangle} \tag{11.2}$$

[失敗]

$$\frac{\begin{array}{c}\vdash_{\mathcal{L}} \langle tag, \rho_0, \sigma_0\rangle \to \langle d, \rho_1, \sigma_1\rangle \\ search(d, \tau_0) = \langle d, exp, \rho, \sigma, \kappa\rangle :: \tau_1\end{array}}{\vdash_{\mathcal{P}} \langle \texttt{(fail } tag\texttt{)}, \rho_0, \sigma_0, \kappa_0, \tau_0\rangle \to \langle exp, \rho, \sigma, \kappa, \tau_1\rangle} \tag{12.1}$$

$$\frac{\begin{array}{c}\vdash_{\mathcal{L}} \langle tag, \rho_0, \sigma_0\rangle \to \langle d, \rho_1, \sigma_1\rangle \\ search(d, \tau_0) = \diamond\end{array}}{\begin{array}{l}\vdash_{\mathcal{P}} \langle \texttt{(fail } tag\texttt{)}, \rho_0, \sigma_0, \kappa_0, \tau_0\rangle \to \\ \quad \langle \texttt{(fail :choice-not-found)}, \rho_0, \sigma_0, \kappa_0, \tau_0\rangle\end{array}} \tag{12.2}$$

[パターンマッチ]

$$\vdash_{\mathcal{L}} \langle \texttt{(:head } \diamond\ \diamond\texttt{)}, \rho, \sigma\rangle \to \langle \#t, \rho, \sigma\rangle \tag{13.1}$$

$$\vdash_{\mathcal{L}} \langle \texttt{(:head } c :: hpat^*\ c :: a^*\texttt{)}, \rho, \sigma\rangle \to \langle \texttt{(:head } hpat^*\ a^*\texttt{)}, \rho, \sigma\rangle \tag{13.2}$$

$$\frac{c \neq b}{\vdash_{\mathcal{L}} \langle \texttt{(:head } c :: hpat^*\ b :: a^*\texttt{)}, \rho, \sigma\rangle \to \langle \#f, \rho, \sigma\rangle} \tag{13.3}$$

$$\frac{\sigma(l) \in \text{Undef}}{\vdash_{\mathcal{L}} \langle \texttt{(:head } hpat^*\ l :: a^*\texttt{)}, \rho, \sigma\rangle \to \langle \#f, \rho, \sigma\rangle} \tag{13.4}$$

$$\frac{\sigma(l) \notin \text{Undef}}{\begin{array}{l}\vdash_{\mathcal{L}} \langle \texttt{(:head } hpat^*\ l :: a^*\texttt{)}, \rho, \sigma\rangle \to \\ \quad \langle \texttt{(:head } hpat^*\ \sigma(l) :: a^*\texttt{)}, \rho, \sigma\rangle\end{array}} \tag{13.5}$$

$$\frac{var \notin Dom(\rho) \qquad l \notin Dom(\sigma)}{\begin{array}{l}\vdash_{\mathcal{L}} \langle \texttt{(:head } \_var :: hpat^*\ d :: a^*\texttt{)}, \rho, \sigma\rangle \to \\ \quad \langle \texttt{(:head } hpat^*\ a^*\texttt{)}, \rho + \{var \mapsto l\}, \sigma + \{l \mapsto d\}\rangle\end{array}} \tag{13.6}$$

$$\frac{var \in Dom(\rho_0) \quad \vdash_{\mathcal{L}} \langle \texttt{(:eq } \rho_0(var)\ d\texttt{)}, \rho_0, \sigma_0\rangle \to \langle \#t, \rho_1, \sigma_1\rangle}{\begin{array}{l}\vdash_{\mathcal{L}} \langle \texttt{(:head } \_var :: hpat^*\ d :: a^*\texttt{)}, \rho_0, \sigma_0\rangle \to \\ \quad \langle \texttt{(:head } hpat^*\ a^*\texttt{)}, \rho_0, \sigma_0\rangle\end{array}} \tag{13.7}$$

$$\frac{var \in Dom(\rho_0) \quad \vdash_{\mathcal{L}} \langle \texttt{(:eq } \rho_0(var)\ d\texttt{)}, \rho_0, \sigma_0\rangle \to \langle \#f, \rho_1, \sigma_1\rangle}{\vdash_{\mathcal{L}} \langle \texttt{(:head } \_var :: hpat^*\ d :: a^*\texttt{)}, \rho_0, \sigma_0\rangle \to \langle \#f, \rho_0, \sigma_0\rangle} \tag{13.8}$$

$$\frac{\vdash_{\mathcal{L}} \langle \texttt{(:head } car\ l_1\texttt{)}, \rho_0, \sigma_0\rangle \to \langle \#t, \rho_1, \sigma_1\rangle}{\begin{array}{l}\vdash_{\mathcal{L}} \langle \texttt{(:head (}car\ .\ cdr\texttt{)} :: hpat^*\ (l_1\ .\ l_2) :: a^*\texttt{)}, \rho_0, \sigma_0\rangle \to \\ \quad \langle \texttt{(:head } cdr :: hpat^*\ l_2 :: a^*\texttt{)}, \rho_1, \sigma_1\rangle\end{array}} \tag{13.9}$$

$$\frac{b \in Literal \cup Object}{\vdash_{\mathcal{L}} \langle \texttt{(:head (}car\ .\ cdr\texttt{)} :: hpat^*\ b :: a^*\texttt{)}, \rho, \sigma\rangle \to \langle \#f, \rho, \sigma\rangle} \tag{13.10}$$

[構造コピー]

$$\vdash_{\mathcal{L}} \langle \texttt{(:copy } c\texttt{)}, \rho, \sigma\rangle \to \langle c, \rho, \sigma\rangle \tag{14.1}$$

$$\frac{\vdash_{\mathcal{L}} \langle exp, \rho_0, \sigma_0\rangle \to \langle b, \rho_1, \sigma_1\rangle}{\vdash_{\mathcal{L}} \langle \texttt{(:copy } \_exp\texttt{)}, \rho_0, \sigma_0\rangle \to \langle b, \rho_0, \sigma_1\rangle} \tag{14.2}$$

$$\frac{\vdash_{\mathcal{L}} \langle exp, \rho_0, \sigma_0\rangle \to \langle\!\langle u, l\rangle\!\rangle, \rho_1, \sigma_1\rangle}{\vdash_{\mathcal{L}} \langle \texttt{(:copy } \_exp\texttt{)}, \rho_0, \sigma_0\rangle \to \langle l, \rho_0, \sigma_1\rangle} \tag{14.3}$$

$$\frac{\begin{array}{c}\vdash_{\mathcal{L}} \langle \texttt{(:copy } car\texttt{)}, \rho_0, \sigma_0\rangle \to \langle d_1, \rho_1, \sigma_1\rangle \\ \vdash_{\mathcal{L}} \langle \texttt{(:copy } cdr\texttt{)}, \rho_0, \sigma_1\rangle \to \langle d_2, \rho_2, \sigma_2\rangle \\ l_1, l_2 \notin Dom(\sigma_2)\end{array}}{\begin{array}{l}\vdash_{\mathcal{L}} \langle \texttt{(:copy (}car\ .\ cdr\texttt{))}, \rho_0, \sigma_0\rangle \to \\ \quad \langle (l_1\ .\ l_2), \rho_0, \sigma_2 + \{l_1 \mapsto d_1, l_2 \mapsto d_2\}\rangle\end{array}} \tag{14.4}$$

[単一化]

$$\vdash_{\mathcal{P}} \langle \{!\ c\ c\}, \rho, \sigma, \kappa, \tau\rangle \to \langle \diamond, \rho, \sigma, \kappa, \tau\rangle \tag{15.1}$$

$$\frac{c_1 \neq c_2}{\vdash_{\mathcal{P}} \langle \{!\ c_1\ c_2\}, \rho, \sigma, \kappa, \tau\rangle \to \langle \texttt{(fail \_)}, \rho, \sigma, \kappa, \tau\rangle} \tag{15.2}$$

$$\frac{\sigma(l) \notin \text{Undef}}{\vdash_{\mathcal{P}} \langle \{!\ l\ d\}, \rho, \sigma, \kappa, \tau\rangle \to \langle \{!\ \sigma(l)\ d\}, \rho, \sigma, \kappa, \tau\rangle} \tag{15.3}$$

$$\frac{\sigma(l) \in \text{Undef}}{\vdash_{\mathcal{P}} \langle \{!\ l\ d\}, \rho, \sigma, \kappa, \tau\rangle \to \langle \diamond, \rho, \sigma + \{l \mapsto d\}, \kappa, \tau\rangle} \tag{15.4}$$

$$\vdash_{\mathcal{P}} \langle \{!\ (l_{11}\ .\ l_{12})\ (l_{21}\ .\ l_{22})\}, \rho, \sigma, \kappa, \tau\rangle \to \langle \{!\ l_{11}\ l_{21}\}, \rho, \sigma, \langle \{!\ l_{12}\ l_{22}\}, \rho\rangle :: \kappa, \tau\rangle \tag{15.5}$$

$$\vdash_{\mathcal{P}} \langle \{!\ o\ d\}, \rho, \sigma, \kappa, \tau\rangle \to \langle \texttt{[}o\ \texttt{\{unify: } d\texttt{\}]}, \rho, \sigma, \kappa, \tau\rangle \tag{15.6}$$

[参考：Lisp に関するいくつかの遷移規則]

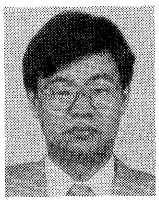
$$\frac{var \in Dom(\rho)}{\vdash_{\mathcal{L}} \langle var, \rho, \sigma\rangle \to \langle\!\langle \sigma(\rho(var)), \rho(var)\rangle\!\rangle, \rho, \sigma\rangle} \tag{16}$$

$$\vdash_{\mathcal{L}} \langle\!\langle l_1, l_2\rangle\!\rangle, \rho, \sigma\rangle \to \langle\!\langle \sigma(l_1), l_1\rangle\!\rangle, \rho, \sigma\rangle \tag{17}$$

[注意すべき点] (1) で，補助変数束縛に関する規則は省略．*guard* の実行では環境は変化しないため，ρ_3 は実際は ρ_2 である（同様のケースはこの式以外にもいくつかある）．(3.1) はパターンマッチが失敗した後の節選択，(3.2) はガードが失敗した後の節選択．(5.1)

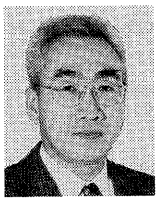
では *Env* を空（ϕ）にし新しい環境を始めている．ここで *def_of* は，述語名とアリティから述語定義を得る関数であり，構文{op 節$^+$}の節並びの部分または $\diamond$ を返す（定義は略）．:copy は，意味記述のために導入した仮想的な構文．a^* の各要素を a_i のように指す．(5.2) は単一化の実行．(9) はボディの途中に現れた関数の $\mathcal{P}$ 評価．返された値を無視する．(10) は継続がない場合の関数の評価．評価値がそのまま返され，これが (11.1) に渡る．(11.1) は述語の成功，(11.2) は失敗．暗黙のチョイスを κ に積んでいる．(11.2) の $\#t$ は暗黙のチョイスが返したもの．(12) で，$search(d, \tau)$ は，d をタグとして持つ最初のチョイスを τ から探し，それ以降のチョイス並びを返す（定義は略）．(12.2) は，fail で指定したタグが見つからなかったときの動作．(13.5) は dereference 処理．(13.6) はヘッドに初出の変数のための束縛をして環境を更新．(13.7), (13.8) はヘッドに 2 回目に現れた変数．:eq は，*undef* どうしの場合はその場所を比較することを除いて Lisp の eq と同じ（定義は略）．(13.9) はコンスセルの分解．呼び出された側のヘッド（:head の第 1 要素）がゴチックつまり構文上の表記であるのに対し，呼び出した側の引数（同第 2 要素）は，すでに構造コピーされているため意味データとなっていることに注意．(14.4) は，コンスの構文から意味データ *Cons* の生成．(15.2) は単一化の失敗．このほかの，*Literal* と *Cons* といった自明な失敗については省略．(15.3), (15.4), (15.6) は，引数を交換した形の規則もあるが，ここでは省略．(15.5) は，(13.9) と異なり，両方が意味データ．これは (5.2) によって，構造コピー済みのデータが単一化に渡されるため．(16) は変数の $\mathcal{L}$ 評価．値とともに場所を返している．(17) は $\mathcal{L}$ 評価における dereference 処理．

(平成 9 年 11 月 10 日受付)
(平成 11 年 3 月 5 日採録)



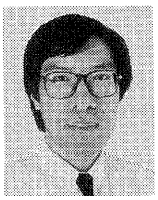
山崎 憲一（正会員）

1961 年生．1984 年東北大学工学部通信工学科卒業．1986 年同大学院情報工学科修士課程修了．同年，日本電信電話（株）入社．現在，NTT 未来ねっと研究所分散ネットワークシステム研究部主任研究員．記号処理専用計算機，記号処理プログラミング言語，日本語文書処理の研究に従事．ACM 会員．



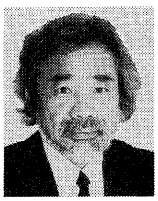
吉田 雅治（正会員）

1953 年生．1976 年千葉大学工学部電気工学科卒業．1978 年同大学院工学研究科修士課程修了．同年，日本電信電話公社入社．現在，日本電信電話（株）NTT サイバースペース研究所サイバー入出力プロジェクト主任研究員．並列処理・画像生成・記号処理等の専用計算機，知能ロボット用センサ等のハードウェアの研究に従事．ユーログラフィックス，電子情報通信学会会員．



天海 良治（正会員）

1959 年生．1983 年電気通信大学電気通信学部計算機科学科卒業．1985 年同大学院修士課程修了．同年日本電信電話（株）入社．以来，プログラミングパラダイム，計算機アーキテクチャ，計算機ネットワークの研究に従事．現在 NTT 未来ねっと研究所分散ネットワークシステム研究部主任研究員．平成 6 年度山下記念研究賞．日本ソフトウェア科学会会員．



竹内 郁雄（正会員）

1946 年生．1969 年東京大学理学部数学科卒業，1971 同大学理学系研究科修士課程修了．同年，日本電信電話公社入社．日本電信電話（株）基礎研究所，ソフトウェア研究所を経て，1997 年から電気通信大学情報工学科教授．主として記号処理言語やシステムの研究に従事．工学博士．平成 2 年情報処理学会論文賞．ACM，日本ソフトウェア科学会各会員．