Lispへのオブジェクト指向の自然な導入

山崎憲一[†,☆] 吉田雅治[††]
天海良治[†] 竹内郁雄[†††]

本論文では，Lispでオブジェクト指向プログラミングを行うための，Lispの拡張について述べる．本論文で提案する言語TAOは，ポリシーとメカニズムを分離するという考えに基づいて設計されており，たとえば，クラスや継承といったポリシーは提供しない．TAOは，オブジェクト指向計算のための最小限のメカニズムと，ポリシーを構築するためのメカニズムだけを備える．メカニズムは，Lispの環境とラムダ式を自然に拡張することにより導入される．ユーザは，これらを用いてTAO上にさまざまなポリシーを構築できる．本論文では，一例として，単純継承と委譲の構築例を示す．また，実装についても述べ，評価を行い，TAOの提供するメカニズムがいずれも十分な性能を達成していることを示す．

An Object-oriented Extension of Lisp

Kenichi Yamazaki,[†,☆] Masaharu Yoshida,[††] Yoshiji Amagai[†]
and Ikuo Takeuchi[†††]

This paper describes an extension of Lisp to incorporate object-oriented programming into Lisp. A symbolic processing language TAO, proposed in this paper, is designed based on policy/mechanism separation principle where mechanisms support essential primitives for object-oriented computation and policies, such as class definition and inheritance, determines how mechanisms are managed. The mechanisms are introduced naturally by extending the Lisp's concept of environment and lambda expression. By using these mechanisms, a TAO user can construct his/her own policy, such as single inheritance and simple delegation which are shown in this paper as an example. We describe the implementation of TAO and evaluate it using some benchmark programs, and as a result, we show that these primitives are efficient enough.

1. はじめに

記号処理あるいは知識処理が，より多くの分野で必要となりつつあり，Lispは今後とも重要な言語の1つであり続けると考えられる[1)]．一方で，オブジェクト指向技術は，単にプログラミング言語にとどまらず，UMLなどの設計法，CORBAなどの分散処理のモデル等々，多くの分野に取り込まれている．したがって，Lispにおいてもオブジェクト指向プログラミングを行いたいという欲求は自然であり，本研究の目的もこの点にある．

同様の目的ですでに多くの言語が提案されてきた．それらの多くはある特定のオブジェクト指向計算を目指すものであった．たとえば，CLOS[2)]は総称関数に基づく言語であり，メッセージ送信型の計算はサポートしない．Object Lisp[3)]は，オブジェクトのスロットを動的に追加できるなどの点できわめて動的な言語であり，逆に静的なクラスベースの言語と同程度の効率を達成することは難しいと思われる．本研究では，このようにオブジェクト指向のある概念をLispに導入するのではなく，オブジェクト指向計算の最も重要な計算機構だけをLispに導入し，それらを組み合わせて実現できることはLispでプログラムするというアプローチをとる．新規に言語を設計するのではなく，ベース言語（本研究ではLisp）の拡張による導入という立場においては，拡張量を小さくすることと，ベース言語を有効に生かすことがポイントであると考えるからである．本論文では，OS研究におけるメカニズムとポリシーの分離の概念を借り，Lispを拡張して取り込む，オブジェクト指向計算の最も根幹となる機能を**メカニズム**，それら機能の組合せにより提供される

† 日本電信電話株式会社
Nippon Telegraph and Telephone Corporation
†† エヌ・ティ・ティ アイティ株式会社
NTT-IT Corporation
††† 電気通信大学
The University of Electro-Communications
☆ 現在，株式会社NTTドコモ
Presently with NTT DoCoMo, Inc.

機能（たとえばクラスの継承）を**ポリシー**と呼ぶ．

その他の既存の言語として，Scheme ベースの研究[4]のように，Lisp の拡張を（ほとんど）せず，つまり，メカニズムをほとんど導入せず，できる限りポリシーとして Lisp で記述するという，いわば「オブジェクト指向 on Lisp」によるアプローチがある．この場合，オブジェクトやメソッドをファーストクラスのデータとして扱うことができない，高い実行効率が期待できないといった問題が生ずる．このことは，単にメカニズムが少なければよいわけでないことを意味する．本研究では，できる限り少ないメカニズムでできる限り多くのポリシーを構築可能とすることを目指し，Lisp をベースとしたプログラミング言語 TAO を提案する．なお，TAO は，Lisp をベースとはしているものの，いくつかの点で独自の機能を持つ．一般の Lisp への本研究成果の適用可能性については，6 章において考察する．

本論文は，次のように構成される．まず，2 章において設計方針を述べる．3 章において TAO 独自の機能について説明する．4 章で提案するオブジェクト指向プログラミングの詳細を述べ，5 章では実装について述べる．6 章ではプログラム例を用いた言語機能の評価および性能評価を行う．7 章で他の関連研究との比較を行い，最後にまとめを行う．

準備：本論文では，通常の関数の意味で引数を評価しない組み込みの式（Common Lisp の特殊形式に相当する式）を構文式と呼ぶ．たとえば，`let` を car とする式は構文式である．これを `let` 構文と呼ぶこともある．構文式において，評価しない引数はブラケット（“<”，“>”）で囲んで表す．... は繰返しを，〜は任意の式の 0 個以上の並びを表す．`#f` は，偽を表すデータである．

2. 言語の設計

オブジェクト指向技術の研究は数多くなされており，多くの提案がなされているが，代表的な概念には以下のようなものがあろう[5),6)]．

- 計算原理に関する概念
  メッセージ送信，総称関数，並行オブジェクト
- オブジェクトの表現に関する概念
  スロット☆とメソッドの分離，あるいは統合，オブジェクトの動的構造変化
- 情報共有のための概念
  クラスとインスタンス，メタクラス，プロトタイプとクローニング
- 継承に関する概念
  （単純/多重）継承，メソッド結合，委譲

これらの中には，互いに同時にメカニズムとはしえないものがいくつかあるが，これらを取捨選択し，我々は次のような方針で言語を設計する．

- オブジェクトをファーストクラスデータとし，このために新しいデータタイプを導入する．ただし，すべてをオブジェクトとは考えない．たとえば，Smalltalk のようにあらゆるデータをオブジェクトとするためには，システム全体のポリシーを必然的に導入する必要がある．
- メッセージ送信を計算原理のメカニズムとする．総称関数は Lisp との親和性の点で優れるが，たとえば PCL（Portable Common Loops）により，Lisp の上にのせることができ，メカニズムとして導入する必要はないと考える．
- 静的なオブジェクトを重視する．スロットの動的な追加/削除を許し，必ずメッセージ送信によりスロットへアクセスするというプロトタイプ型言語の方法では，静的にアクセス可能な言語と同程度の性能を達成することは難しい．動的なメソッド追加が可能であれば，オブジェクトの動的変更はポリシーとして実現可能である．また，スロットを静的なものとしたことにより，スロットの共有に関しても何らかのメカニズムが必要となる．
- その他の機能，クラス，継承，委譲等々は，すべてポリシーと考え，ユーザがこれを構築することを支援するためのメカニズムを用意する．

3. TAO 独自の機能

本章では，TAO 独自の機能について述べる．

3.1 フォーム化

プログラムをデータとして操作できることは，Lisp の特徴であるが，TAO では，プログラム操作の意味を明確にするため，操作対象としてのデータと実行可能なプログラムとを区別する．あるデータを実行可能とするためには，`form` 構文による**フォーム化**が必要である．

```
(eval (form '(+ 1 2)))
```

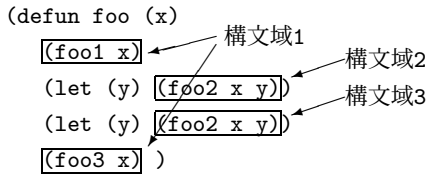
`form` は，ファーストクラスデータである**フォーム**を返す．フォームは関数 `eval` により実行可能である．

フォームに関する重要な概念に，**構文位置**と**構文域**がある．構文位置とは，直感的にはテキスト上の位置

☆ インスタンス変数やフィールドとも呼ばれるが，本論文では一貫してスロットと呼ぶ．

のことであるが，より正確にはフォームの中の位置である．同じテキストを複数回フォーム化すると，複数のフォームができるから，同じテキスト上の位置でも異なる構文位置となる．あるいは，フォーム化によりテキストのバージョン番号が上がると考えてもよい．

let などの構文式により確立されるレキシカルな束縛のスコープはネストするので，ある構文位置のレキシカルな状態を決める構文式は複数存在する．構文域とは，同じ構文式の集合によってレキシカルな状態が決められる構文位置の集合のことである．たとえば，



というプログラムには，3 つの構文域がある．

あるフォームは，それがフォーム化された構文域と同じ構文域においてのみ eval 可能である．

```
1:  (let ((expr '(cons x y)) f)
2:     (!f (form expr))
3:     (eval f)
4:     (let (x y) (eval f))
5:     (let (a b) (eval f)) )
```

ここで 2 行目は，変数 f への代入を行う代入式である．3 行目の eval は，form（2 行目）と同じ構文域であるから実行可能である（変数 x と y は大域変数となる）．4 行目の let のボディは，2 行目とは異なる構文域であるから，エラーとなる．また，5 行目のようにフォーム化した式に影響を与えない構文式であっても，構文域が異なるため eval できない．

実装の観点では，フォーム化とはバイトコードへのコンパイルである．フォームには，それをフォーム化した構文域に関する識別子が付与され，eval の際にこれをチェックする．なお，フォームから，関数 unform を使って元の式を取り出すことができる．

3.2 関　　数

TAO における計算を行う主体には，関数と述語とがあり，両者を合わせて作用素と呼ぶ．ただし，本節を除いては「作用素リジョン」という言葉でしか現れない．また，本論文では，述語については触れない．

次のような関数生成構文を実行することにより無名関数が生成される．

(op *<parameter-list>* *<body-form>*...)
(op* *<parameter-list>* *<body-form>*...)
(op@ *<parameter-list>* *<body-form>*...)

これらは，Common Lisp のラムダ式に相当する．op，op*，op@ の違いは，主にスコープルールである．

作用素のスコープに関する議論において，作用素のボディ（*<body-form>*... の部分）を作用素リジョンという．op によって生成される関数は，**単純関数**と呼ばれる．次の式を実行すると，単純関数が返される．

```
(op (x y) (/ (+ x y) 2))
```

単純関数の作用素リジョンにおいては，その外のレキシカルな束縛は，いっさい参照できない．

op* によって生成される関数は，**動的関数**と呼ばれる．その作用素リジョンにおいては，その外側のレキシカルな束縛が（その束縛の存続期間内である限り）参照できる．つまり，動的関数は，いわゆる下向きの funarg（関数引数）において有効な関数閉包である．

op@ によって生成される関数は，**オブジェクト関数**と呼ばれる．詳細は 4.3 節で述べる．

parameter-list には，次のような書き方も許される．

```
(op (x y . z) (foo . z))
```

第 3 引数以降の引数の並びを z に受け取り，それを foo にそのまま渡す．Common Lisp の &rest と apply に相当する機能であるが，セルメモリを消費しない．

次のように，関数に一部（またはすべて）の引数を与えて新たな関数を作るプリミティブが用意されている．

(curry *function* arg_1 arg_2 ...)

たとえば，

```
(let ((f (curry #'+ 1))) (funcall f 2))
```

を実行すると 3 が返される．これをカリー化☆と呼ぶ．

関数は，次の関数呼び出し式により実行される．

(*<function>* arg_1 arg_2 ...)

関数呼び出し式の car は評価されないが，アンダースコアを付与することにより，評価することができる．たとえば，上の例の funcall は次のように書ける．

```
(_f 2)
```

4. オブジェクト指向プログラミング

この章では，まずオブジェクト指向計算のメカニズムについて述べ，それらを組み合わせてポリシーを構築するためのメカニズムを 4.4 節で述べる．

4.1 オブジェクト

オブジェクトは，データと操作を閉じ込める．関数

☆ これは部分適用と呼ぶべき操作だが，便宜的にカリー化と呼ぶ．

閉包を用いて，たとえば let で作られたレキシカルな変数束縛（つまり環境）を閉じ込めることはよく知られた技法である[7]．しかし，その場合，環境は副次的な結果として閉じ込められるにすぎない．TAO では，環境を作り出すプリミティブである let を拡張し，環境をファーストクラスデータとして独立させ，それをオブジェクトと見なす．これにより，オブジェクトの同一性などの概念が自然に導入できる．

オブジェクトは，obj-let 構文により生成される．

```
(obj-let ((<slot-name> initial-value)...)
    <body-form>...)
```

この式を評価すると，まず各スロット名を初期値に束縛し，それらを閉じ込める**オブジェクト**を生成する．そして，左から順に *body-form* を実行していき，最後の値を obj-let 構文の値として返す．obj-let のボディを，**オブジェクトリジョン**と呼ぶ．obj-let によって宣言された変数を，スロット変数またはスロットと呼ぶ．obj-let の生成したオブジェクトは，オブジェクトリジョンにおいて，読み込み専用変数 @ で参照できる．スロット変数は，レキシカルかつ無限存続である．一方，TAO の let で宣言された変数は，レキシカルかつ動的存続であり，Common Lisp の let の変数は，レキシカルかつ無限存続である．これは，Common Lisp の lambda が変数を無限存続に閉じ込めるからである．TAO では，let を含む静的変数を動的関数によって閉じ込めることができるが，これは下向きの funarg であるため動的存続となる．TAO では，無限存続の閉じ込めは，obj-let によってしか行えない．

オブジェクトは概念的には，それを生成した obj-let の構文位置の情報，スロット名から値へのマップ（スロット値集合と呼ぶ），シンボルからメソッドへのマップ（メソッド表と呼ぶ）の三つ組みで定義される．構文位置の情報は，そのオブジェクトがどの obj-let で生成されたかを特定する．これは，いわばオブジェクトのバージョン番号のようなものである．同じ構文位置を持つオブジェクトどうしは，スロット名の集合も同じである．メソッド表については後述する．

obj-let は，次のようにネストすることができる．

```
(obj-let ((a 0))
   (obj-let ((b 0)) (!a 1))
   (obj-let ((c 0)) (print a)) )
```

3 番目の obj-let で作られたオブジェクトは，スロット a，c を持つ．上のプログラムを実行すると 1 が印字される．これは，let の動作から類推される自然な動作である．ただし，オブジェクトを主体として考えれば，上の動作はスロット a が共有されていると見なすこともできる（5 章で述べるように，実装上もポインタにより共有されている）．そこでこのように複数のオブジェクトで共有されたスロットを，**共有スロット**と呼ぶ．

4.2 情報の共有

TAO では，オブジェクトどうしが情報の共有を行うための基本的なメカニズムを用意する．オブジェクトは，前述のように三つ組により規定される．この三つ組みの共有方法に関して次の 3 種類がある．

最初の方法は，構文位置とメソッド表を共有し，スロット値集合だけが異なるものである．このような共有をするオブジェクトどうしは，強相似であるといわれる．強相似のオブジェクトは，次の関数によって生成される．

```
(copy-obj  object)
```

この式の返した直後のオブジェクトは，元の *object* とスロット値は同じであるが，その後は各スロット値を独立に変更できる．強相似は，たとえば同じクラスに属するインスタンスどうしの関係を表す．

2 番目の方法は，構文位置の情報だけを共有するものである．このような共有をするオブジェクトどうしを，弱相似であるという．そのようなオブジェクトの典型例は，同じ obj-let を複数回実行することによって生成される．たとえば，

```
(defun make-obj (x) (obj-let ((a x)) @))
```

により定義された関数 make-obj が返すオブジェクトどうしは，弱相似である．同じ構文位置にある obj-let から生成されたからである．また，次の関数

```
(new-obj  object)
```

を用いてコピーすることにより，弱相似のオブジェクトを生成することもできる．弱相似のオブジェクトは，異なるメソッド表を持つことができる．強相似ならば弱相似である．以降，相似といった場合は，弱相似を指す．

3 番目の方法は，前述の obj-let のネストによる，スロット値の共有である．後述の例（図 1）では，これを用いてクラス変数を表現する．

なお，copy-obj や new-obj によるコピーでは，（もしあれば）共有スロットは新しく生成されたオブジェクトにおいても共有される（実装上は，共有ポインタをそのままコピーするだけである）．

4.3 メソッドとメッセージ送信

Lisp においては，操作はラムダ式によって抽象化される．op@ 構文は，オブジェクトに閉じ込められた操作を生成するためのラムダ式である☆．op@ 構文は，オブジェクトリジョン内でのみ実行可能である．op@ 構文を含む最も内側の obj-let が生成したオブジェクトを,「オブジェクト関数のオブジェクト」と呼ぶ．

オブジェクト関数の仮引数の第 1 要素には自動的に @ が付与される．つまり，次のようなオブジェクト関数

```
(op@ (x y) ～)
```

は，実は 3 引数であり，その仮引数は @，x，y である．オブジェクト関数を実行する場合，その第 1 引数は，オブジェクト関数のオブジェクトと相似なオブジェクトでなければならない．

オブジェクト関数の作用素リジョンにおいては，その外側のレキシカルな束縛のうち，すべてのスロット変数の束縛だけが参照可能である．たとえば，let で束縛された変数は参照できない．オブジェクト関数の作用素リジョン内からスロット変数にアクセスすると，オブジェクト関数の第 1 引数に渡されたオブジェクトの保持するスロットがアクセスされる．

オブジェクト関数と動的関数の違いについて述べる．動的関数は，従来の Lisp の（下向き）関数閉包に相当するもので，それを生成したときの環境をそのまま閉じ込める．一方，オブジェクト関数では，第 1 引数にオブジェクトを与え，そのオブジェクトのスロットを参照する．つまり，オブジェクト関数にとって，オブジェクトが環境である．例を示す．

```
1:  (obj-let ((x 0))
2:    (let ((ofn (op@ (n) (!x (+ x n)) x)))
3:      (print (_ofn (new-obj @) 2))
4:      (print (_ofn @ 3)) ))
```

このように，new-obj の生成した新たなオブジェクト（=環境）に対してオブジェクト関数を適用（3 行目）してもエラーとならず，実行される点が重要である．その結果，2 が印字される．一方，4 行目の結果，3 が印字される．このように，オブジェクト関数には，環境を個別に与えることができる．その意味では，実はオブジェクト関数は動的関数のような閉じ込めは行っていない．

オブジェクト関数と，その環境であるオブジェクトを，セットで扱って閉じ込めを表現したい場合がある．そのためには，カリー化を行って第 1 引数のオブジェクトを与えておけばよい（もちろん，オブジェクト関数のオブジェクトと相似でなければならない）．そのようにして生成された関数を，特に**オブジェクト閉包**と呼ぶ．

関数にメッセージ送信のインタフェースをもたせるには，次の関数により，オブジェクトに関数を登録する．

(attach-method *object selector to-be-method*)

selector はシンボルでなければならず，*to-be-method* は次の 3 つのいずれかでなければならない．

(1) オブジェクト関数（ただし，このオブジェクト関数のオブジェクトと *object* とは，相似でなければならない）
(2) 任意のオブジェクト閉包
(3) 任意の単純関数

登録に用いたシンボルを**セレクタ**と呼ぶ．登録された関数を**メソッド**と呼ぶが，特に，(1) のメソッドをホームメソッド，(2) を駐在（alien）メソッド，(3) を単純メソッドと呼ぶ．すでに同じセレクタで登録されたメソッドがあれば，上書きされる．

メソッドは，次のメッセージ送信式によって，起動される．

[*object* (<*selector*> arg_1 arg_2 ...)]

メッセージ送信式は，次のように実行される．*selector* を用いて *object* のメソッド表が検索される．メソッドが見つからなかったときについては，後述する．メソッドが見つかったときは，その種類に応じ次のようになる．

- ホームメソッドのときは，*object*, arg_1, arg_2 ... を引数として，ホームメソッドを呼び出す．
- 駐在メソッドのときは，arg_1, arg_2 ... を引数としてオブジェクト閉包を呼び出す．すでに第 1 引数は，カリー化によって与えられており，*object* は結果として捨てられる．
- 単純メソッドのときは，*object*, arg_1, arg_2 ... を引数として，単純関数を呼び出す．

new-obj においては，メソッド表はコピーされるが，それぞれのメソッドはコピーされない．上の手順から，ホームメソッドは送信先オブジェクトが必ず第 1 引数となるが，駐在メソッドはすでにオブジェクトをカリー化によって保持しているため，コピーされた後も，必ずそのオブジェクトが第 1 引数となることが分かる．

例として，きわめて簡単なクラスを考えてみよう（**図 1**）．中間の obj-let は，クラス SpaceShip を生成し，最も内側の obj-let は，インスタンスのテンプレートを生成する．

☆ 論文 8) における op@ の定義とは異なるものである．

```
(!SpaceShip
  (obj-let ((number-of-spaceships 0))
    (obj-let ((template
               (obj-let ((x 0) (y 0))
                 (attach-method @ 'warp ~)
                 (attach-method @ 'move ~)
                 @ ))) ; end of template
      (attach-method @ 'make
        (op@ ()
          (!number-of-spaceships
            (1+ number-of-spaceships))
          (copy-obj template) ))
      @ )))
```

図 1 簡単なクラスの定義
Fig. 1 Definition of a simple class.

```
(!aSpaceShip [SpaceShip (make)])
```

を実行すると，テンプレートの強相似オブジェクトが aSpaceShip に代入される．このオブジェクトは，メソッド warp や move を持つ．一番外側の obj-let は，クラス変数 number-of-spaceships を共有するためのものである．

4.4 ポリシー構築のためのフック

TAO のオブジェクト指向計算の基本的なメカニズムは，以上に述べてきたものですべてである．図 1 で述べたような簡単なクラスの構築，特に実行前にすべての定義が決定可能な場合には，これで十分であるが，より動的な性質を持つ言語を構築するには，ユーザがメカニズムの意味を変更できる機能が必要となる．

このために，TAO はメッセージ送信にフックをかけるメカニズムを提供する．メッセージ送信は，送信先のオブジェクトからメソッド表を得て，その中を探し，見つかった関数を起動するという一連の動作である．このそれぞれのステップを，独立したメカニズムとして用意し，ユーザがそれらを組み合わせて，メッセージ送信の意味を自分で構築するという方法もありえよう．しかし，これは過度の単純化である．そのような方法で，メッセージ送信に十分な性能を与えることは難しいと考えられる．なお，TAO は上記の各ステップを行うプリミティブも用意するが，それらはフックがかかった後のメタ操作に用いることを目的としている（付録 A.1）．

メッセージ送信の意味を変更するため，メソッド探索に関して，2 つのフックを設定できる．1 つは，メソッドが見つからなかったときに呼ばれる関数で，**不明メソッドハンドラ**と呼ばれ，次の関数により登録する．

(attach-missing-method-handler *obj mmh*)

不明メソッドハンドラ *mmh* は，次のようなインタフェースの関数でなければならない．

(op (*obj mfh selector . args*) ~)
(op@ (*mfh selector . args*) ~)

obj にメッセージが送られ，メソッド表にメソッドが登録されていなかった場合，不明メソッドハンドラが呼び出される（このハンドラが登録されていなければエラー）．不明メソッドハンドラの仮引数 *obj*（op@ の場合は @），*selector*，*args* は，それぞれ元のメッセージ送信のオブジェクト，セレクタ，引数並びである．*mfh* については後述するが，通常は #f である．この関数が返す値が，元のメッセージ送信式の値となる．なお，不明メソッドハンドラはメソッド表に属すもので，コピーに対する意味などはメソッドと同じである．

もう 1 つのフックは，**メソッド発見フック**である．このフックは，メソッドが見つかった直後に呼び出される．メソッド発見フックは，不明メソッドハンドラのようにオブジェクトに対して登録するのでなく，次のようにメッセージ送信のたびに陽に指定する．

(method-found-hook <*message-form*> *mfh*)

message-form は，メッセージ送信式でなければならない．また，メソッド発見フック *mfh* は，

(op (*obj found-method selector . args*) ~)
(op* (*obj found-method selector . args*) ~)

というインタフェースの関数でなければならない．

method-found-hook 中の *message-form* が評価され，メソッドが見つかると，メソッド発見フックが呼び出される．その仮引数 *obj*，*selector*，*args* は，それぞれ元のメッセージ送信式のオブジェクト，セレクタ，引数並びである．*found-method* は発見されたメソッドである．メソッド発見フックが返した値が，method-found-hook 構文の値となる．見つかったメソッドは実行されないため，必要ならば，メソッド発見フックの中で陽に *found-method* を呼び出す．

一方，*message-form* を評価し，メソッドが発見されなかったとき，不明メソッドハンドラが（もし登録されていれば）呼び出される．このとき，仮引数 *mfh* にメソッド発見フックが渡される．不明メソッドハンドラの中で，*mfh* を呼び出せば，メソッドが見つかったようにシミュレートすることができる．

通常は，メソッド発見フックの仮引数 *obj*，*selector*，*args* は，元の *message-form* 中のものと一致するため，これらの仮引数は無意味なものに見えるが，上記のように不明メソッドハンドラが *mfh* を呼び出す場

合には，一致しないことがある．

メソッド発見フックは，メッセージ送信に対して指定するが，たとえばデバッグのため，あるオブジェクトへの全メッセージをトレースするには，メソッド発見フックをオブジェクトに対して指定できると都合が良い．その場合は，次のようにする．オブジェクト生成部を修正して，デバッグ用オブジェクトを代わりに返すようにする．このオブジェクトは，メソッドを持たず，不明メソッドハンドラでトレースをとった後で，本来のオブジェクトへメッセージを送り直す．このように，メソッド発見フックと不明メソッドハンドラには，相補的な性質がある．我々は，多くのテストプログラミングを行い，このような性質と，フックの指定方法を組み合わせることにより，さまざまなポリシーに対応できることを確認した．そのうちの 2 例を 6.1 節に示す．

5. 実　　装

本章では，メッセージ送信の実装について述べる．

TAO は，記号処理マシン SILENT[9] 上に実装される．SILENT はクロック 33 MHz，データキャッシュ 320 K バイト，マイクロプログラム可能なマシンで，バイトコードの自動ディスパッチ機能，簡易ハッシュ機能などが特徴である．

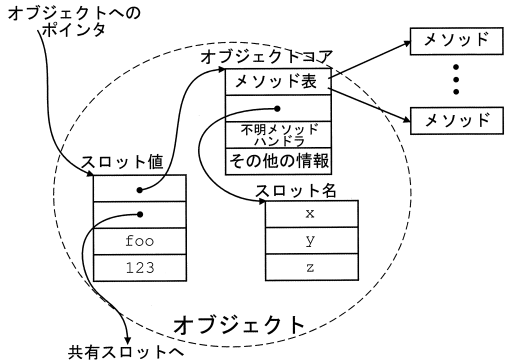
オブジェクトは，**図 2** のような内部表現を持つ．オブジェクトは，スロット値の集合と，オブジェクトコアから構成され，オブジェクトコアは，メソッド表，スロット名の表，不明メソッドハンドラの情報からなる．スロット名の表は，各 `obj-let` ごとに作られるもので，4.1 節で述べた構文位置の情報に相当する．オブジェクト間の弱相似性はこの表の同一性で，強相似性はオブジェクトコアの同一性で判定できる．このように，オブジェクトを規定する三つ組みが，ほぼそのままの形で実装されている．

メッセージ送信式

[*obj* (*selector* arg_1 arg_2 ...)]

は，次のようなバイトコードにコンパイルされ，マイクロコードにより記述されたバイトコードインタプリタがこれを実行する．

compile(*obj*)
`make-message-frame` *LABEL*, *selector*
compile(arg_1)
compile(arg_2)
...
`call-method` *offset*
LABEL:



`(obj-let (x) (obj-let ((y 'foo) (z 123)) @))` というプログラムにより生成されたオブジェクトの内部表現を示した．変数 `x` は，外側の `obj-let` が生成したオブジェクトのスロットへのポインタとなる．

図 2 オブジェクトの内部表現
Fig. 2 The internal representation of an object.

ここで，*compile* は，その引数をコンパイルしたバイトコード列を表す．*compile*(*obj*) のバイトコードを実行すると，*obj* がスタックトップにプッシュされる．`make-message-frame` は，*obj* のオブジェクトコアのメソッド表から，*selector* で登録されたメソッドを探し出して，これをスタックに積む．次に引数が順に積まれる．`call-method` は，スタックに積まれたメソッド（この位置は *offset* で示される）から，そのインタフェースの情報を読み，引数の個数のチェックなどの後，メソッドを実行する．

関数呼び出しではシンボルに格納された関数をスタックに積むが，これに対しメッセージ送信ではメソッド表を検索する部分がオーバヘッドとなる．メソッド表は，ハッシュと 2 分木探索を組み合わせたデータ構造である．また，ハッシュには，ハードウェアハッシュの機能を用いている．

6. 評　　価

TAO を評価する観点には，オブジェクト指向計算とポリシー構築のためのメカニズムの妥当性（機能面と性能面），および Lisp をベース言語としたことを有効に利用したかの点があろう．本章では，これらについて検討していく．

6.1 ポリシー構築機能

本節では，2 つのポリシーを構築することで，メカニズムの機能面を評価する．

まず，メソッドの単純継承を実現するプログラムの一部を**図 3** に示す．ここでは，メッセージが送られて見つからなかったら，そのときに上位クラスを探し，

```
 1: (obj-let
      〜
 2:    (attach-missing-method-handler @
 3:       (op@ (mfh selector . args)
 4:          (method-found-hook [super (_selector . args)]
 5:             (or mfh
 6:                 (op* (obj method selector . args)
 7:                    (let ((my-method [@ (eval-in (unform method))]))
 8:                       (attach-method @ selector my-method))
 9:                       (_my-method @ . args) ))))))
10:    (attach-method @ 'eval-in (op@ (e) (eval (form e))))
      〜 )
```

unform には，フォームの他に関数を与えることができ，元の関数生成構文が返される．メソッド eval-in により，任意のデータを obj-let の中の環境でフォーム化し eval することができる．

図 3 単純メソッド継承のための不明メソッドハンドラ
Fig. 3 Missing method handler for single method inheritance.

```
 1: (defun delegation (self mfh selector . args)
 2:   (method-found-hook [[self (prototype)] (_selector . args)]
 3:     (op* (method receiver . args)
 4:          (if (op-function? method)          ; 単純関数かの判定
 5:              (_method self . args)          ; 元々の送信先のメソッド呼び出し
 6:              (_method receiver . args) )))) ; 委譲先のスロットへのアクセス
 7: (!ship1 (obj-let ((x 0) (y 0))
              〜
 8:            (attach-method @ 'warp
 9:              (op (self px py) (![self (x)] px) (![self (y)] py)) )))
10: (!ship2 (obj-let ((x 0) (y 0) (prototype ship1))
11:            (attach-method @ 'x (op@ () x))
12:            (attach-method @ 'y (op@ () y))
13:            (attach-method @ 'prototype (op@ () prototype))
14:            (attach-missing-method-handler @ #'delegation) ))
```

図 4 簡単な委譲のための不明メソッドハンドラ
Fig. 4 Missing method handler for simple delegation.

見つかったメソッドを元のクラスに定義するという方法で実現している．この例のような不明メソッドハンドラが，すべてのクラスに定義されていたとしよう．また，すべてのクラスは，スロット super を持つとする．これは，上位クラスのテンプレートとなるオブジェクトであり，これにメッセージを送ることにより，上位クラスのメソッド定義を参照できる．あるクラスにおいてメソッドが見つからなかったとき，不明メソッドハンドラが呼び出され，method-found-hook（4 行目）によって，上位クラスが探索される．このとき mfh は #f であるから，op*（6 行目）によりメソッド発見フックが生成される．1 つ上位のクラスでもメソッドが見つからなかったときには，mfh に渡されたフックがメソッド発見フックとなる．ある上位クラスでメソッドが見つかると，メソッド発見フックが実行される．これは op* のスコープに関する性質により，op* が生成された環境を参照することになる．つまり，@ は元々のオブジェクトであり，これに対してメソッドが動的に登録される（eval-in については，図中の説明を参照）．最後に，そのメソッドを呼び出す．

これはメソッド継承の例であるが，スロットの継承はまったく異なる手法をとる．ここでは詳細は述べないが，継承すべきスロットをメタ操作（付録 A.1 の slots など）により集め，obj-let 構文を作り出し，それを適切な環境中で eval する．

次に，簡単な委譲の例を**図 4** に示す．これは基本的な流れを示すためのプログラムであり，関数 delegation は，委譲したメッセージがさらに委譲される場合を想定していない（つまり mfh を無視している）．ここでは，オブジェクトのスロットにアクセスするためのメ

ソッドはホームメソッドにより，それ以外の一般のメソッドは単純メソッドにより記述されることを前提としている．

このプログラムに対し，

```
[ship2 (warp 10 10)]
```

を実行すると，ship2 には warp が定義されていないため，関数 delegation が実行され，ship2 のスロット prototype の値である ship1 に対して，method-found-hook の中でメッセージが送られる．メソッド warp が見つかり，メソッド発見フックが実行される．登録されているメソッドの種類により，送信先を変えているところが 1 つのポイントである．通常のメソッドの場合には元々の送信先のオブジェクト（self）を引数として単純メソッドが呼び出され，スロットアクセスの場合には委譲先のオブジェクト（receiver）を引数としてホームメソッドが実行される．

以上，メソッド発見フックと不明メソッドハンドラという 2 つのフックにより，簡単な継承や委譲が TAO 上にポリシーとして構築可能であることを示した．これにより，メカニズムとポリシーを分け，少ないメカニズムで多くのポリシーを構築可能とするという，当初の設計目標が達成されたことをある程度示せたと考える．

一方，Lisp を有効に利用した拡張となっているかについては，この例で示したように TAO の次のような機能がポリシー構築のために有効であった．

(1) 引数を並びとして受け取る機能
(2) op* によりレキシカルな環境にアクセスする機能
(3) プログラム評価の機能

これらの点が一般の Lisp にも適用可能であるかについて考察する．(1) は，Common Lisp の &rest と同等である．(2) および TAO の関数全般にわたる議論として，TAO の関数は，実行効率向上のため，一般の Lisp の lambda による関数のスコープをより限定する方向で種類分けしたものにすぎない．したがって，op* は lambda で代用できる．また，オブジェクト関数の導入も，lambda の実装と同程度の手間で可能である．

一方，(3) は，TAO に強く依存した機能である．たとえば，Common Lisp の eval では外部のレキシカルな環境をいっさい参照できない．TAO の eval が必要になった理由は，主にスロットを静的なものとしたことによる．つまり，(図 3 の eval-in のように) スロットが見える環境中で評価を行うことによってしか，スロットにアクセスできないためである．図 3 のように継承を動的に行うのではなく，あらかじめ継承をすべて展開してしまうようにすれば eval は不要となる．また，図 4 の例のように，スロットアクセスをメソッド経由で行う場合にも不要である．このようにポリシーを限定すれば，本研究成果を通常の Lisp に適用することは可能であるが，どの程度までのポリシーを実現できるかについては今後の課題である．

表 1 各プリミティブの実行時間
Table 1 Execution time of the primitives.

let で宣言された変数の読み出し	0.0768
スロットの読み出し	0.142
関数呼び出し（式を実行して終了するまでの時間．以下も同じ．）	0.872
ホームメソッドの呼び出し（10 種類のセレクタで登録し，それらを順に呼び出した平均）	1.32
駐在メソッドの呼び出し（同上）	1.51
単純メソッドの呼び出し（同上）	1.46
不明メソッドハンドラ（未登録のメッセージを送信し，値が返るまでの時間．登録メソッド数は 10）	2.05
メソッド発見フック（メッセージを送信し値が返るまでの時間．ホームメソッド呼び出しと同じ条件）	1.89
attach-method（10 種類のセレクタを登録する平均値）	5.91
obj-let（スロット 8 個を宣言し，ただちに @ を返す）	9.26
copy-obj（スロット 8 個のオブジェクト）	6.44
new-obj（10 個のメソッドが登録され，スロット 8 個を持つオブジェクト）	24.3

（単位は μ 秒）

6.2 性能評価

SILENT は，実行マシンサイクル数を測定する機能を備えており，これを用いて実行性能の測定を行った．

各メカニズムの実行時間を測定した結果を**表 1** に示す．変数読み出しは，静的な変数の 2 倍程度かかっているが，これは let による変数はハードウェアスタックに置かれ，主記憶上に置かれたスロットとは，ハードレベルのアクセス速度が異なるためである．

ホームメソッドの呼び出しは，関数呼び出しの 50% 程度，約 0.45 μ 秒の速度低下となっている．さらに分析したところ，この内訳は，メソッド表の検索が約 0.29 μ 秒，その他はメソッドの種類の分類などであった．これらがメッセージ送信のオーバヘッドとなる．ここで測定に用いたプログラムは，メソッドの中で何の計算も行わないので，50%は最悪値に近い．また，ハッシュの性能に関しては，より詳細な分析を行う必要はあるが，セレクタ数を 10 から 100 に増やしても，1.33 μ 秒とほぼ変わらない値を得ている．

2 つのフックの呼び出しも，通常のメソッド呼び出しと大きな差はない．図 3 のように，ある条件のとき

表 2 CLOS 処理系との比較
Table 2 Comparison with CLOS systems.

	o-tak	tak4	比	tak
TAO	21.9	15.6	1.40	11.2
CMU CL	2.97	1.47	2.02	1.46
Allegro CL	2.13	0.764	2.79	0.628

いずれも，(tak 12 6 0) 相当の実行時間で単位は秒．比とは o-tak/tak4．測定条件は，CMU Common Lisp: バージョン 18b PCL 版，Allegro Common Lisp: バージョン 5.0.1，測定マシン：Pentium III 600 MHz，FSB100 MHz，L2 キャッシュ512 KB，OS: FreeBSD 3.3.

（メソッドを上位クラスから下位クラスに移す必要があるときなど）にのみポリシーが実行されるような場合は，このフックの性能は十分なものである．一方，図 4 の委譲の例のように，通常の実行においても頻繁にポリシーが実行される場合には，この性能では不十分な場合もあろう．我々は，そのような場合でも，多くはプログラム上の技巧で解決できると考えている．たとえば，図 4 では委譲のたびにメッセージをプロトタイプへと送っていたが，オンデマンドで元の送り先オブジェクトのメソッド表にメソッドを再定義すれば，本来のメッセージ送信と同じ性能を達成できる．

次に，TAO のオブジェクト指向計算と関数型計算の性能バランスについて考察するため，CLOS 処理系と比較した（**表 2**）．プログラム o-tak は付録 A.2 に示す．o-tak の測定に際しては，TAO では，あるオブジェクトに 10 種類のセレクタで同じメソッドを定義し，各メソッドの実行時間を平均した．CLOS では，10 種類のクラスとそれぞれを第 1 引数として特定化する同名のメソッドを 10 個定義し，各メソッドの実行時間を平均した．また，tak4 は，引数の評価の回数が o-tak と同じになるように，ダミーの第 1 引数を加えた tak である．tak における TAO と Allegro CL の比は約 18 であり，クロック比も 18（600/33）である．アーキテクチャがまったく異なるため，厳密な議論はできないが，関数型プログラムの性能はほぼ同等と考えられる．

o-tak と tak4 の比は，TAO が最も小さい．CMU CL と Allegro CL の tak では，同一ハード上で 2 倍以上の差が生じていることから，Allegro CL は最適化によって性能向上を狙うタイプの処理系と考えられる．オブジェクト指向プログラムでは，最適化がより難しいため，Allegro CL の比が大きくなったものと思われる．一方，TAO のメッセージ送信のメカニズムは単純であり，関数型で記述してもオブジェクト指向で記述してもバランスの良い性能を達成できる．

7. 関連研究

TAO と同じスタンスの研究はあまり多くはなく，直接の比較は難しい．ここでは，さまざまな観点から，他研究との比較を行う．

Lisp の関数閉包を用いてオブジェクト指向プログラミングを行うことは，本論文の動機となった手法である．Scheme ベースの研究[4] においては，委譲の機能などが，関数閉包を用いて実現可能であることが示されている．オブジェクトも関数閉包によって表現されているが，一般の関数閉包と区別できるようにオブジェクトのために新しいデータ型を追加していることが唯一の拡張である．できるだけ少ない Lisp の拡張でオブジェクト指向を導入するという点は本研究の立場に近い．しかし，関数閉包でオブジェクトを表現しているため，オブジェクトどうしの類似性の判定やオブジェクトのコピーといったオブジェクトをデータとして扱う操作を定義しにくい．TAO の方が拡張量は多いが，これらの操作を厳密に定義できる．

Object Lisp[3] は，プロトタイプ型言語であり，メッセージ送信ではなく，オブジェクトという環境の中で式を評価することを計算モデルとする．たとえば，プリミティブ `ask` を用いて，

```
(ask obj (print x))
```

により `obj` の環境の中で式 `(print x)` を評価できる．`obj` が `x` というスロットを持てば，それが参照される．オブジェクトを環境ととらえるというコンセプトは TAO に共通するものであるが，Object Lisp では環境は動的であり，いわば Lisp の大域変数をオブジェクトに閉じ込めたものに近い．一方，TAO では環境は静的であり，`let` の拡張となっている．このため，6.2 節で示したように TAO では効率良くスロットにアクセスできる．

Lisp の拡張ではない，一般のオブジェクト指向言語と比較した場合，TAO はプロトタイプ型言語に最も近い．多くのプロトタイプ型言語[6] との相違は，オブジェクトの動的構造変化，つまりスロットの動的な追加が可能かどうかにある．TAO では，これはポリシーとして構築する．つまり，スロットは使わず，メソッドとして用意する．

ポリシー構築のメカニズムについては，フックを TAO にとっての MOP[10] であると考えることもできる．MOP は言語要素に対応する Metaobject を決め，それに default の意味を定義し，それらをオブジェクト指向プログラミングを用いて修正する．一方，TAO

のフックは，オブジェクト指向計算の意味を完全に再構築することを狙いとする．たとえば，不明メソッドハンドラを委譲を実現するための機構に用いることができ，それに耐えるような実装がなされている．つまり，TAO のフックは，あくまでもメカニズムの一部である．むろん，ユーザがプロトコルを設計し，フックを用いて MOP と類似のシステムを実現することも可能である．

8. ま と め

本論文では，オブジェクト指向を Lisp に導入するための拡張について述べた．拡張量を少なくし，Lisp を有効に利用するため，メカニズムとポリシーの分離の考えに立ち，プログラム言語 TAO を提案した．TAO は，`let` を拡張した `obj-let` によりオブジェクトを生成し，情報共有のメカニズムとして弱相似と強相似によるコピーを提供し，ラムダ式（`op`）を拡張した `op@` とメッセージ送信により，オブジェクト指向計算を実現する．また，これらのメカニズムを組み合わせてポリシーを構築するためのフックが用意されており，さまざまなオブジェクト指向の概念をユーザが構築できる．本論文では，いくつかのポリシー構築の例を示すことにより，メカニズムの妥当性を示した．また，メカニズム部分の実行性能を測定し，フックなどのポリシー構築機能も含めて十分な性能となっていること，Lisp と比較した場合のオーバヘッドが大きくないことを確認した．

参 考 文 献

1) 日本 Lisp ユーザ会：*Lisp User Group Meeting Japan* (2000).
2) Steele Jr., G.: *Common Lisp the language*, 2nd edition, Digital Press (1990).
3) Byers, G., et al.: *Allegro Common Lisp Manual*, Coral Software Corp. and Franz, Inc. (1987).
4) Adams, N. and Rees, J.: Object-Oriented Programming in Scheme, *ACM Symp. on Lisp and Functional Programming*, pp.277–288 (1988).
5) Wegner, P.: Dimensions of Object-Based Language Design, *OOPSLA'87*, pp.168–182 (1987).
6) Noble, J., Taivalsaari, A. and Moore, I.: *Prototype-Based Programming: Concepts, Languages and Applications*, Springer (1999).
7) Abelson, H., Sussman, G. and Sussman, J.: *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press (1996). 和田英一（訳）：計算機プログラムの構造と解釈 第二版，ピアソン (2000).
8) Yamazaki, K., Amagai, Y., Yoshida, M. and Takeuchi, I.: TAO: An object orientation kernel, *Intl. Symp. on Object Technologies for Advanced Software*, Lecture Notes in Computer Science, Vol.742, pp.61–76, Springer-Verlag (1993).
9) 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).
10) Kiczales, G., Rivières, J. and Bobrow, D.: *The Art of the Metaobject Protocol*, MIT Press (1991).

付 録

A.1 オブジェクト指向に関するプリミティブ

TAO のオブジェクト指向に関するプリミティブを示す．`curry` など関数型に関するプリミティブは除く．本文中で触れなかったプリミティブを ∗ で示す．

- `[`*obj* `(`<*selector*> *arg*...`)]`
  関数メッセージ送信式
- ∗ `[`*obj* `{`<*selector*> <*arg*>...`}]`
  述語メッセージ送信式
- `(obj-let ((`<*slot-name*> *slot-value*`)...)`
  `(`<*body-form*>...`)`
  オブジェクトを生成し，*body-forms* を順に評価
- `@`
  生成されたオブジェクトを指す読み込み専用変数
- `(op@` <*param*> <*body-form*> 〜`)`
  オブジェクト関数の生成
- `{op@` <*clause*> 〜`}`
  オブジェクト述語の生成
- `(copy-obj` *obj*`)`
  強相似オブジェクトの生成
- `(new-obj` *obj*`)`
  弱相似オブジェクトの生成
- ∗ `(strong-akin?` obj_1 obj_2`)`
  オブジェクトが強相似かどうかの判定
- ∗ `(weak-akin?` obj_1 obj_2`)`
  オブジェクトが弱相似かどうかの判定
- `(attach-method` *obj selector to-be-method*`)`
  selector という名前で *obj* にメソッドを登録
- `(method-found-hook` <*message-form*> *mfh*`)`
  メソッド発見フック *mfh* の下で *message-form* を評価
- `(attach-missing-method-handler` *obj mmh*`)`
  obj に不明メソッドハンドラ *mmh* を登録
- ∗ `(slots` *obj*`)`
  スロットのリストを返す
- ∗ `(shared-slots` *obj*`)`
  共有スロットのリストを返す
- ∗ `(missing-method-handler` *obj*`)`
  obj に登録された不明メソッドハンドラを返す

* (selectors *obj*)
  obj に登録されたセレクタのリストを返す
* (method *obj selector*)
  obj に *selector* の名で登録されたメソッドを返す
* (slot-value <*slot-name*>)
  オブジェクトの外から代入可能とするため，スロットの左辺値と値を返す
* (attach-obj *x obj*)
  プロセスやベクタなどシステム定義データを疑似オブジェクトとして扱うためのオブジェクト（随伴オブジェクト）を *x* のデータ型に登録する
* (adjoint-obj *x*)
  x のデータ型の随伴オブジェクトを返す
* (obj-put *obj symbol value*)
  obj のオブジェクトコアに *symbol* という属性名で *value* を登録する
* (obj-get *obj symbol*)
  obj のオブジェクトコアから属性名 *symbol* の値を得て返す
* (obj-remove *obj symbol*)
  obj のオブジェクトコアから属性名 *symbol* を取り除く

attach-*XXX* という関数の最後の引数に _（未定義を表す特殊な値）を与えると，登録対象を削除するという意味になる．

A.2 ベンチマークプログラム

[TAO による o-tak の定義]

```
(attach-method 'o-tak @
  (op@ (x y z)
       (if (<= x y) y)
           [@ (o-tak
                [@ (o-tak (1- x) y z)]
                [@ (o-tak (1- y) z x)]
                [@ (o-tak (1- z) x y)])])))
```

[CLOS による o-tak の定義]

```
(defmethod o-tak ((obj class0) x y z)
  (if (<= x y) y
    (o-tak obj
      (o-tak obj (1- x) y z)
      (o-tak obj (1- y) z x)
      (o-tak obj (1- z) x y))))
```

(平成 12 年 6 月 21 日受付)
(平成 13 年 10 月 16 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．2000 年 NTT ドコモネットワーク研究所へ転籍．現在，同研究所ユビキタスサービスネットワーキング研究室長．記号処理プログラミング言語，モバイルネットワークの研究に従事．博士（工学）．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，東日本電信電話（株）企画部所属，エヌ・ティ・ティ アイティ（株）ファイル・映像ソリューション事業部に出向中．第 2 技術部担当部長．主として画像系のシステム開発に従事．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 未来ねっと研究所ネットワークインテリジェンス研究部主任研究員．1994 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 年同大学院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．1990 年情報処理学会論文賞．ACM，日本ソフトウェア科学会会員．