Java 上の Scheme 処理系「ぶぶ」における単一のクラスローダを用いたオブジェクトシステムの実装

窪田貴志[†] 湯淺太一[†] 倉林則之[††]
八杉昌宏[†] 小宮常康[†]

「ぶぶ」は，Java 言語とのシームレスなインタフェースを備えたオブジェクト指向型の Scheme 処理系である．Web ブラウザ上での実行や，遠隔実行処理への応用が試行され，Scheme の持つ対話性を生かしつつ，Java の機能を最大限に引き出すことが望まれている．ぶぶの従来のオブジェクトシステムでは，Scheme のクラスの実装に，Java のクラスそのものを用いていた．このために，クラス再定義やメソッドの追加・再定義の際にクラスローダを再構築することが必要となりオーバヘッドが大きく，また，アプレット上で Scheme のクラスを生成することができないといった問題点があった．そこで，これらの問題を，Java オブジェクトに Scheme のクラス情報を保持させることにより解決した．Scheme のクラスを定義する際や，Scheme のクラスに対するメソッドの追加・再定義は，クラス情報を格納するための Java オブジェクトの生成，メソッド検索テーブルへの登録という，比較的軽い処理で実現した．また，Java 側から Scheme のメソッドが呼び出せるように，両言語間のインタフェースをとる Java クラスをあらかじめ用意しておく方式を採用した．本研究の方法は，Java VM が提供するデフォルトのクラスローダ上で，従来と同等の機能を持つオブジェクトシステムを実現するものであり，アプレットのセキュリティ・モデルを満たすものである．

An Object System Implementation Using the Primordial Class-loader for the Java-based Scheme System "Bubu"

Takashi Kubota,[†] Taiichi Yuasa,[†] Noriyuki Kurabayashi,[††]
Masahiro Yasugi[†] and Tsuneyasu Komiya[†]

"Bubu" is an object-oriented Scheme system with seamless interface to Java. "Bubu" is intended to be used on Web browsers and for distributed applications, and we expect this system to draw out maximum functionality of Java in an interactive environment of Scheme. The previous implementation of the object system used a Java class to represent a user-defined Scheme class. That implementation therefore had to construct a new class-loader each time the programmer modifies a Scheme class, or adds or modifies a Scheme method. It was a time-consuming operation. In addition, it caused a problem that we could not create a Scheme class on Java applets. We solved these problems by representing the class information of each Scheme class as a Java object. In this method, the object system has only to create a Java object when a Scheme class is defined and to register a Scheme method to a method table when the programmer adds a new method or modifies an already existing method. Our new implementation assumes that Java classes are prepared in advance which work as the interface from Java to Scheme so that Scheme methods can be called from the Java side. This implementation realizes the same functionality as the previous implementation by making use of the primordial class-loader of Java VM, and satisfies the security model for Java applets.

1. はじめに

ぶぶ[1)]は Java 言語で記述された Scheme 処理系である．フルセットの IEEE Scheme[2)]に準拠し，拡張仕様として，オブジェクトシステム，例外処理機能，遠隔オブジェクト操作機能を持つ．オブジェクトシステムに関しては，Java 言語を用いて実装された他の Scheme 処理系[3)~6)]と異なり，Java との完全な言語透過性を実現している点が特徴である．Java のクラスライブラリや，プログラマが作成した Java のクラスをロードして利用できるほか，これらのクラスを拡張して Scheme 言語レベルのサブクラスを定義するこ

† 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
†† ATR 環境適応通信研究所
ATR Adaptive Communications Research Laboratories

とが可能である．Scheme のクラスやインタフェースは，Java クラスの変数やメソッドを継承できると同時に，Java クラスの変数のハイディング（hiding），メソッドのオーバライディング（overriding）も可能である．このようなオブジェクト指向拡張機能は，Java の持つネットワークや GUI の機能と Scheme のダイナミックな特性の両者をより良いバランスで利用することを可能にしており，Web ブラウザ上のアプリケーションや遠隔実行機構への応用[7]が試みられている．

ぶぶの従来のオブジェクトシステムは，セキュリティやパフォーマンス面で，いくつかの問題をかかえていた．本研究では，これらの問題を解決する新たな方式の設計と実装を行った．

本稿では，Java 上の Scheme 処理系ぶぶにおけるオブジェクトシステムの実現法について述べる．Scheme と Java という特徴の異なる言語間に透過的な相互呼び出し機構を導入することにより，Java レベルで記述可能な操作は Scheme レベルでも記述できるようにした．まず，2 章でぶぶの実行方式について説明し，3 章でオブジェクトシステムの仕様について述べる．4 章では，従来の実装方式の問題点とその解決法を述べる．5 章で新方式を用いた実装の詳細について説明する．6 章では，Java-Scheme 間の相互呼び出しを利用した簡単なアプリケーションを示す．

2. ぶぶの概要

図 1 に，ぶぶ処理系の実行方式を示す．ぶぶ処理系は，S 式をぶぶバイトコードと呼ばれる中間コードへ変換し，Java で記述されたぶぶバイトコード インタプリタ上で実行する．S 式からぶぶバイトコードへの変換を行うぶぶコンパイラは，それ自身 Scheme で記述されており，ぶぶバイトコード インタプリタ上で動作する．

ぶぶのファーストクラス・オブジェクトはすべて Java オブジェクトであり，各データ型に固有の操作は各データに対応するクラスのメソッドとして実装されている．組込み関数は，Java のクラスメソッドとして記述されている．また，ごみ集めには Java VM の GC を利用している．

現在，インターネット上で入手可能な Java 上の Scheme システムとしては，Kawa[3]や Silk[4]がよく知られている．Kawa は S 式から Java バイトコードへのコンパイラとして実装されている．Silk はインタプリタ型で，プログラムサイズがコンパクトであるため，Web ブラウザ上でアプレットとして利用するのに適している．ぶぶと Kawa をいくつかの点で比較した結果を**表 1** に示す．

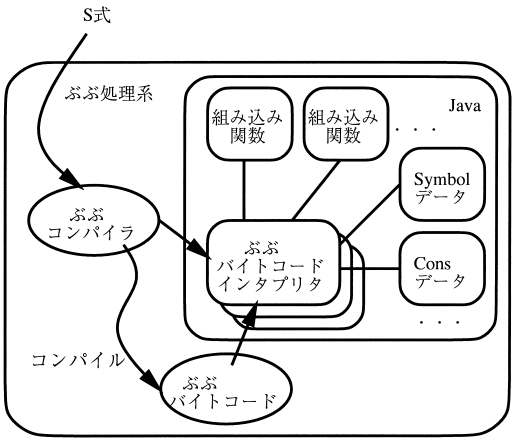

図 1 ぶぶ処理系の概要
Fig. 1 An overview of Bubu.

表 1 ぶぶと Kawa の比較
Table 1 Comparison between Bubu and Kawa.

	Bubu	Kawa
準拠	near-IEEE	near-R^{5}RS
実装方式	interpreter	compiler
string mutability	×	○
continuation	○	△
Applet execution	○	×
(fib 25) (sec)	5.051	7.714
(tak 21 14 7) (sec)	7.758	11.775

- 文字列
  Java の String クラスは immutable な文字列を生成するため，Scheme の mutable な文字列を生成するためには直接的に利用することができない．Kawa では，mutable な文字列を表現するために，FString という独自のクラスを用意しているのに対し，ぶぶでは Java の String クラスをそのまま使用している点で，IEEE 準拠となっていない．
- 継続
  ぶぶでは，無制限の寿命を持つ継続を扱うことができる．実装方法としては，ぶぶバイトコード インタプリタの実行時スタックをコピーし，現在の実行状態をつかまえておくという，一般的な方法を用いている．Java 言語レベルでは，Java の実行時スタック（Java VM のオペランド スタック）の状態をコピーすることができないため，Kawa では Java の例外を利用して継続を実装している．Kawa の継続は非局所的脱出にのみ使用することができ，コルーチンなどは実現することができない．
- アプレット実行

Java アプレットは，セキュリティ上の制約により，実行時に生成した Java クラスをロードすることができない．したがって，Kawa のように，S 式を直接 Java バイトコードに落とす方式の処理系は，アプレットとして利用することができない．一方，ぶぶでは，S 式を Java で記述された中間コード（ぶぶバイトコード）に落とすため，この点は問題とならない．

- 実行性能
  2 種類の関数 `fib` と `tak` によって，ぶぶと Kawa の実行性能を比較した．結果として，Java VM 上でぶぶバイトコードインタプリタを実行するという，二重のインタープリテーション方式をとっているにもかかわらず，ぶぶの方が若干高速であるという結果が得られた．Kawa では，文字列以外にも，数値やベクタなど，Scheme のデータ型を表現するために独自のクラスを導入しているのに対し，ぶぶでは，数値やベクタを Java のプリミティブ・クラスや配列を用いて表現している．これにより，Scheme の演算を Java の基本演算に置き換えているという点が大きな要因の 1 つと考えられる．

3. オブジェクトシステムの仕様

3.1 Java と Scheme の相互運用

Kawa や Silk で用いられている Java アクセスの方法は，

```
(define f
        (get-java-method target-class
                         method-name
                         return-type
                         (arg-type...)))
(f target arg...)
```

のように，あらかじめ完全なメソッドシグネチャを指定してメソッドオブジェクトを取得しておき，それらを Scheme の関数と同等に扱うというものである．一方，ぶぶでは，メッセージパッシング的に

```
(send target method-name arg...)
```

のように指定する．呼び出されるメソッドは，ターゲット，メソッド名，そして引数の型によって動的に決定される．

いずれの方式も，Scheme 環境から Java をアクセスするインタフェースによって，Java のクラスライブラリの一部を利用する機能を提供している．しかし，クラスの定義機能を持たない Java アクセスインタフェースだけでは，クラス継承によるフレームワークを提供するクラスライブラリを，十分に活用することができない．

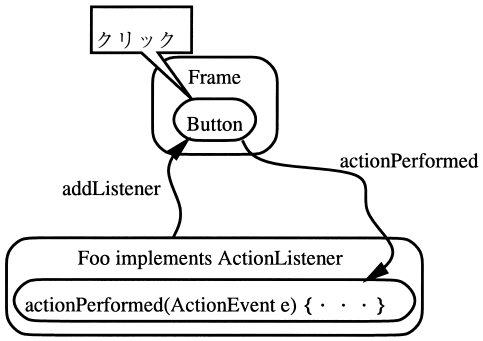

図 2 イベント委譲モデルの一例
Fig. 2 An example of event delegation model.

一例として，Java AWT や Swing で採用されている，GUI のイベント処理フレームワークである，イベント委譲モデル（event delegation model）をあげる．このモデルでは，**図 2** に示すように，Listener と呼ばれる特定のインタフェースを実装し，イベントの処理を行うメソッドを持つオブジェクトを各 UI 部品に関連づけることにより，UI 部品から発生するイベントの処理を行う．このモデルに基づくアプリケーションを記述するためには，Java クラスのサブクラスを Scheme 環境で定義する機能が必要である．

ぶぶオブジェクトシステムは，こうしたクラス定義機能を有しており，Java のオブジェクト指向機能に基づく Scheme のオブジェクト指向拡張機能を提供している．構文は CLOS[8] や ISLISP[9] を参考にしているが，多態性（polymorphism）は包括関数（generic function）を用いるのではなく，Java と同様に継承（inheritance）とメソッドオーバライディングによって表現する．Java におけるインタフェースの機能もサポートしており，表現能力は Java とほとんど☆等しい．Scheme で定義されるクラスのインスタンスは，Java サイドでは，通常の Java オブジェクトと同様に振る舞う．**図 3** にその例を示す．図 3 では，Java クラス A を引数の型とする Java メソッド N の引数として，クラス A を拡張して定義した Scheme クラス B のインスタンス b を渡している．また，クラス A のインスタンスメソッド M は Scheme クラス B においてオーバライドされており，Java 側から Scheme で記述されたメソッド M に制御を移している．

☆ いまのところ内部クラスと無名クラス，クラスパッケージ・システムはサポートしていない．

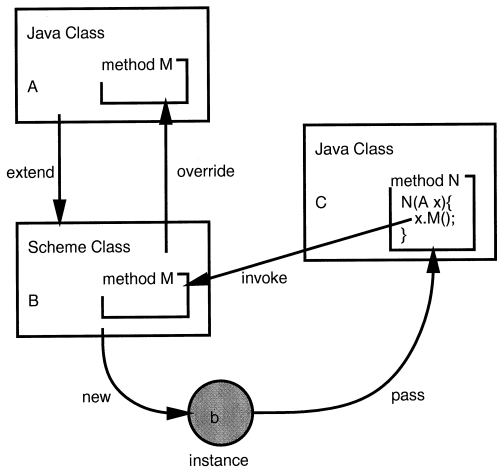

図 3 Scheme と Java の相互運用
Fig. 3 Inter-operation between Scheme and Java.

3.2 構文と仕様

Java クラスを扱うための関数として，以下のものが用意されている．まず，クラスオブジェクトの取得には，次式を用いる．

(`class` *class-name*)

クラス名を完全指定する代わりに，`import` 式

(`import` *path*)

によって，Java クラスの検索パスを指定しておけば，完全限定名の代わりに，単純形式のクラス名を使用することもできる．インスタンスの生成には，`new` 式

(`new` *class-name arg*...)

を使用する．この式はまず `class` 式によって *class-name* に対するクラスオブジェクトを取得し，引数に対応するコンストラクタを呼び出す．生成されるオブジェクトは Scheme のファーストクラス・オブジェクトとなる．オブジェクトのクラスを得るには，`class-of` 式

(`class-of` *object*)

を用いる．メンバ変数へのアクセスは

(`slot-ref` *target field*)

(`slot-set!` *target field value*)

を用いて行う．ここで，*field* はフィールド名，または，クラス名とフィールド名からなるリスト（*class-name field-name*）のいずれかである．Scheme オブジェクトの型はダイナミックに決まるため，Java 言語のように，アクセスすべき変数をコンパイル時に特定することができない．Scheme クラスのインスタンスが，同名のメンバ変数を 2 個以上持つ場合，クラス階層上最も近いクラスに属するメンバ変数がアクセスされるが，*class-name* を指定することにより，任意のメンバ変数をアクセスすることも可能である．*target* がクラスの場合，クラス変数のみへの参照となる．メソッド呼び出しには，次の `send` 式を用いる．

(`send` *target method-name arg*...)

target がクラスの場合，クラスメソッド呼び出しとなる．メソッド呼び出しの例として，以下のような Java クラスを考える．

```
public class A {
   M(String x) {...}
   M(int x) {...}
   M(Integer x) {...}
}
```

Java クラス A には 3 個のメソッドが定義されており，これらはすべて同じメソッド名 `M` を持つが，互いに異なる型の引数を受け取る．1 番目のメソッドは `String` を受け取り，2 番目のメソッドはプリミティブ型の `int` を受け取る．そして 3 番目のメソッドは参照型の `Integer` を受け取る．ここで，クラス A の検索パスをインポートし，A のインスタンス `p` を生成する．

```
(import "java.....A")
(define p (new A))
```

以下のように，文字列`"Hello."`を引数としてメソッド `M` を呼び出した場合，

```
(send p M "Hello.")
```

ぶぶオブジェクトシステムは，Java クラス A の 1 番目のメソッドを呼び出す．次に，引数の型が整数型である場合を考える．

```
(send p M 10)
```

Java におけるプリミティブ型データは，ぶぶでは参照型に自動変換されて保持される．そして，Java に渡される時点で，参照型からプリミティブ型に自動変換される．この例の場合，ぶぶオブジェクトシステムは，プリミティブ型の整数を受け取る 2 番目のメソッドを優先的に呼び出す．3 番目のメソッドを呼び出す手段として，次のように，メソッドのシグネチャを指定した呼び出し方法が提供されている．

```
(send p (M java.lang.Integer) 10)
```

ただし，プリミティブ型の `int` を受け取る 2 番目のメソッドが定義されていない場合は，このようにシグネチャを指定しなくても参照型の `Integer` を受け取るメソッド `M` が呼び出される．

次に，Scheme クラスを扱うための関数を説明する．Scheme クラス定義には，次式を用いる．

(`defclass` *class-name*
([*superclass-name*])
class-option...

field-def...
method-def...)

ここで

class-option ::= (:is *Java-class-modifier*)
| (:implements *interface-name*...)
field-def ::= (*field-name type field-option*...)
field-option ::= *init-form*
| (:reader *name*)
| (:writer *name*)
| (:accessor *name*)
| (:is *Java-field-modifier*)
method-def ::= (:method *method-name*
((*arg type*)...)
method-option...
body)

言語仕様上，すべての Scheme クラスはスーパクラスとして Scheme クラスまたは Java クラスを持つ．*superclass-name*が省略された場合は `java.lang.Object` が指定される．*interface-name* には Java インタフェース名または Scheme インタフェース名を指定可能である．*field-option* の *init-form*，`:reader`，`:writer`，`:accessor` は CLOS[8]や ISLISP[9]と類似の形式である．*method-def*節は，ここで定義しようとしているクラスのメソッドを定義するもので，次のメソッド定義式と同じ意味を持つ．

(defmethod *class-name*
method-name
((*arg type*)...)
method-option...
body)

ここで

method-option ::= (:throws *exception-class*)
| (:is *Java-method-modifier*...)

`defmethod` 式は Scheme クラスに対するメソッドの追加/置き換えを行う．Java では，返値を指定することにより，クラス名と同名のメソッドを定義することもできる．しかし，ぶぶではメソッドの返値を指定しないため，*method-name* と *class-name*が等しい場合には，コンストラクタ定義となる．コンストラクタ *body* 部の先頭の式では，Java のコンストラクタと同様

(this *arg*...)
(super *arg*...)

による他のコンストラクタ呼び出しが可能である．これらが省略された場合，無引数のコンストラクタでは

(super)

によって，スーパクラスの無引数のコンストラクタが，1 個以上の引数を持つコンストラクタでは

(this)

によって，自クラスの無引数のコンストラクタが呼び出される．

Scheme クラスのメソッドおよびコンストラクタは，次のメソッド削除式により，削除することが可能である．

(delete-method *class-name*
method-name
(*type*...))

特に，*method-name*と *class-name*が等しい場合には，コンストラクタの削除となる．引数の型指定（*type*...）を省略した場合，多重定義（overload）されているメソッドがすべて削除される．

インタフェース定義式は，次のように，クラス定義式と類似の形式を用いる．

(definterface *interface-name*
(*superinterface-name*...)
interface-option...
field-def...
method-def...)

ここで

interface-option ::= (:is *Java-class-modifier*)

言語仕様上，メソッド呼び出しやインスタンス生成において，Java クラスと Scheme クラスに違いはない．Java クラスの再定義はできない．Scheme クラスを再定義した場合は，すでに生成したインスタンスのクラスは変化しない．次の 5 つの式は Scheme クラスについても Java クラスとまったく同様に使用することができる．

`class` `new` `slot-ref`
`slot-set!` `send`

ぶぶオブジェクトシステムは，Scheme クラスに対するパッケージ・システムをサポートしていないため，`import` 式で指定される検索パスは Java のクラス検索に関してのみ使用される．

スーパクラスのメソッドを呼び出す機能として，*method-def* 節およびメソッド定義式の *body* 部でのみ，`super` 式

(super *method-name arg*...)

の使用が認められている．また，メソッド内でスロットをアクセスする場合，`let` 式や `lambda` 式によって変数名 *field-name* が局所的に束縛されない限り，以下の 2 式は同義となる．

(slot-ref this *field-name*)
field-name

Java 言語と同様に，変数 this は，オブジェクト自身を指す変数として，インスタンスメソッドとコンストラクタの *body* 部で使用することができる．

4. オブジェクトシステムの設計

4.1 従来方式の問題点

ぶぶの従来のオブジェクトシステムでは，以下のような実装方式を用いている．Scheme のクラスは，Java-Scheme 間のインタフェースを提供する Java のクラスと，メソッドおよびアクセサの実体となる Scheme のクロージャを組み合わせたものとして表現する．これらのクロージャは Scheme の大域変数に格納しておき，Java クラスの各メソッドが，大域変数からクロージャを取得して実行する．

たとえば，以下のようなクラス定義式を考える．

```
(defclass B (A)
  (field_1 ...)
    :
  (field_n ...)
  (:method method_1 (arg ...) ...)
    :
  (:method method_m (arg ...) ...)
)
```

ぶぶコンパイラは，上記の Scheme コードから，以下のような，同名の Java クラスと，各メソッドに対応する Scheme のクロージャを生成する．そして，Scheme クラス B は，これらを組み合わせたものとして実現する．つまり，Scheme クラス定義式に存在するメンバ変数やメソッドは，すべて Java クラス B の定義中に埋め込まれる．

- Java クラス

```
public class B extends A {
  field_1;
  field_1_reader(){...}
  field_1_writer(Object obj){...}
   :
  field_n;
  field_n_reader(){...}
  field_n_writer(Object obj){...}
  method_1(arg ...) {
    method_1 に対応する Scheme のクロージャ実行
  }
   :
  method_m(arg ...) {
    method_m に対応する Scheme のクロージャ実行
  }
}
```

- Scheme のクロージャ

```
method_1 に対応する Scheme のクロージャ定義
  :
method_m に対応する Scheme のクロージャ定義
```

この実装方法には，2 つの問題点が存在する．

第 1 の問題点は，アプレットのセキュリティに関する制約から生じる問題である．Web ブラウザ上にネットワーク経由でダウンロードされるアプレットは，アプレット・セキュリティマネージャの制約に従わなければならない．実行前にバイトコード検証系によってコードの安全性がチェックされ，ファイル I/O やクライアントのライブラリのロード，そしてユーザ定義のクラスローダの使用が禁止される．これらの制約により，アプレットは，実行時に生成される Java クラスをロードすることができない構造になっている．自前のクラスローダを生成しようとすると，セキュリティ例外が発生してしまう．

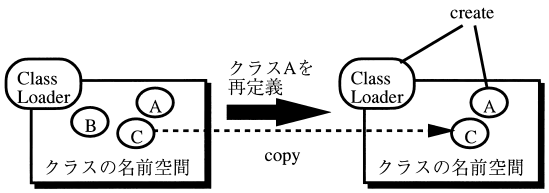

図 4 従来の実装方式
Fig. 4 Previous implementation.

第 2 の問題点として，クラスやメソッド定義に対するパフォーマンスが悪いという点があげられる．ぶぶをファイルシステムから起動する場合には，アプレットの制約はなくなるため，自前のクラスローダを生成し，実行時に生成した Java クラスをロードすることが可能になる．しかし，Java のクラスローダは，単一のクラスローダを用いて同名のクラスを再ロードし，差し換えることができない．このため，同名の Scheme クラスを再定義したり，Scheme クラスにメソッドを追加/再定義する度に，**図 4** のように，新たなクラスローダを構築し，必要なクラスは古いクラスローダからコピーしてくる必要がある．

4.2 新方式の検討

上記の問題に対する解決策の候補として，以下のような方法が考えられる．

(1) クラスファイルと同等のバイト配列をアプレット上で生成してサーバ側に転送し，サーバはそれを適切な URL でアクセス可能となるようにする．アプレットがそのクラスを必要としたとき，通常の方法でそのクラスをダウンロードする．

(2) Scheme のクラス名と Java のクラス名を分離して，それらの対応関係をオブジェクトシステムで管理する．Scheme 上で同名のクラスが定義されても，それぞれ異なる名前の Java クラスにマッピングを行う．

(3) Scheme クラスを Java クラスを用いて表現す

るのをやめて，別のオブジェクトを用いて表現する．Scheme クラスのインスタンスの実装に用いるための Java クラスは，あらかじめ既存のライブラリから作成しておく．

方法 (1) では，クラスをリロードすることができない．リロードするためには，アプレット自身をリロードせざるを得ない．方法 (2) は，単一のクラスローダを用いて実現できるが，依然として動的にクラスを生成しなければならないという問題が残っている．方法 (3) は，根本的に Java のクラスを生成する必要がない．クラス再定義時における名前の衝突に関しても，Scheme のクラス名は，クラス情報を格納するオブジェクトの一部として保持すればよい．

以上のことから，方法 (1) と方法 (2) を組み合わせる方法，もしくは方法 (3) が候補としてあげられる．本研究では，比較的コンパクトに実現可能であるという理由により，後者を採用することにする．

ここで，Java クラス A のサブクラスとして Scheme クラス S を定義するとする．このとき，S のインスタンスの実装に用いる Java クラス A′ の仕様を考える．S のインスタンスは，クラス A を引数の型に持つ Java メソッドに渡すことができなければならない．このため，A′ は必然的に A のサブクラスである．

Scheme のクラスは，Java のクラスを拡張することによって定義されるため，Java のプログラム上に Scheme クラスに関する記述は現れない．この点に注目すると，次のことがいえる．

(1) Java のメンバ変数は，静的な型情報を基にアクセスされる．Scheme クラスが Java クラスから継承したメンバ変数は Java から参照されるが，Scheme クラスのその他のメンバ変数は Java から参照されることはない．

(2) Java から呼び出される Scheme のメソッドは，Java のメソッドをオーバライドしたものに限定される．つまり，Java から呼び出すことのできる Scheme のメソッドは，Java クラスで定義されたメソッドと同じシグネチャを持つものだけである．

(1) より，Scheme クラス S で定義されたメンバ変数は，Scheme サイドからのみ参照できればよい．よって，S のメンバ変数は，必ずしも A′ のクラス定義中に埋め込む必要はない．Java からのアクセス対象となりうるメンバ変数は，A′ が A から継承しているため，特に考慮する必要はない．また，(2) より，Java クラス A を調べれば，Java から呼び出される可能性のある Scheme のメソッドシグネチャを判定することができる．A′ では，これらのメソッドについてのみ，Scheme のメソッドを呼び出すインタフェースを提供すればよい．必ずしも S のすべてのメソッド定義を A′ のクラス定義中に埋め込む必要はない．

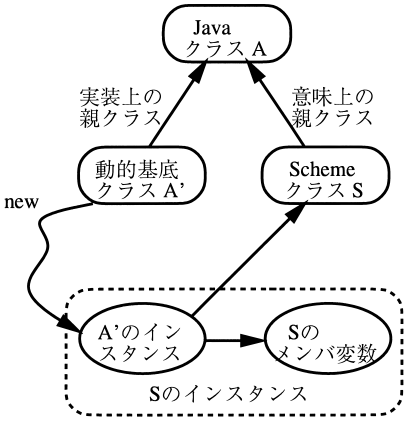

図 5 動的基底クラス
Fig. 5 Dynamic base-class.

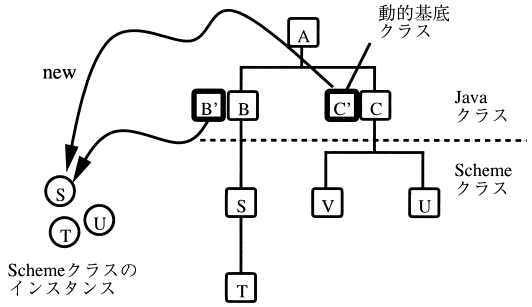

図 6 動的基底クラスの決定
Fig. 6 Decision of dynamic base-class.

4.3 動的基底クラス

Java クラス A′ を動的基底クラス（dynamic base-class）と名付ける．動的基底クラスの概念図を**図 5** に示す．Scheme クラス S のメンバ変数は動的基底クラス A′ 上には定義されていないため，A′ のインスタンスとは別に保持する．つまり，S のインスタンスは，A′ のインスタンスと Scheme のメンバ変数をまとめたものと見なすことができる．S のインスタンスを Java に渡す場合は，A′ のインスタンスを渡す．

次に，S のサブクラスを T とし，T の動的基底クラス A″ について考える．上記の性質 (1)，(2) より，A″ に求められる仕様は，A′ と同じである．つまり，T のインスタンスも，S と同一の動的基底クラスを用いて実装できることが分かる．**図 6** に示すように，Scheme クラスのインスタンスは，クラス階層上最も近い Java クラスの動的基底クラス（S と T に対

してはB′，UとVに対してはC′）を用いて実装できる．図6における動的基底クラスB′やC′は，Javaのクラス B，Cから前もって生成しておくことが可能である．このため，実行時の Java クラス生成は不要となり，従来の実装方式における問題は解決される．

Scheme のクラス定義は，メンバ変数情報やメソッド情報などを格納するための Java オブジェクトを生成することで実現できる．また，Scheme クラスに対するメソッドの追加/再定義が行われる場合も，上記の Java オブジェクトに対してメソッドを登録/更新するだけで実現できる．これは，Java クラス生成と比較して，はるかに軽い処理で済む．

5. オブジェクトシステムの実装

5.1 Scheme クラス/メソッド定義

Scheme のクラス/メソッド定義の流れを**図7**に示す．ぶぶコンパイラは，Scheme クラス定義式の評価の過程で，`SchemeClass` クラスのインスタンスを生成し，これにクラス情報を格納し，クラスデータベースへ登録する．メソッド定義式の *body* 部は，関数閉包（closure）にコンパイルし，アクセス修飾子などのメソッド情報とともに，`SchemeMethod` オブジェクトに格納する．`SchemeMethod` オブジェクトは，`SchemeClass` オブジェクトのメソッドテーブルに登録する．

5.2 動的基底クラスの実装

Scheme クラスのインスタンスの実装には動的基底クラスを用いる．動的基底クラスはあらかじめ生成しておくことが望ましいが，セキュリティ上問題がなければ実行時に生成することもできる．

動的基底クラスは，以下の2つのインスタンス変数を持つ．

- `SchemeClass $SchemeClass`

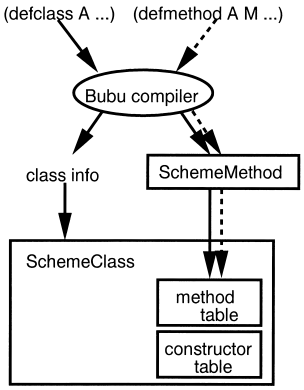

図7 クラス・メソッド定義の流れ
Fig. 7 Flow of class/method definitions.

インスタンスの意味上のクラスである `SchemeClass` オブジェクトへの参照を保持する．

- `List $slots`

Scheme のメンバ変数への参照を保持する．

`$slots` が保持する Scheme のメンバ変数は，Java からは通常の変数のようにアクセスできない．しかし，4.2 節で述べたように，Java がアクセスする変数は，Java クラスから Scheme クラスへと継承された変数に限定されるため，問題はない．

Java クラス A がインスタンスメソッド

```
int foo(int,Object)
```

を持つとき，A の動的基底クラス A′ では，`foo` を以下のようにオーバライドする．

```
public int foo(int p0, Object p1){
  if($SchemeClass == null) //(A)
    return super.foo(p0, p1); //(B)
  else { //(C)
    if(Invoker.find($SchemeClass, "foo")){
      Object[] args = {BCI.makeInt(p0),p1};
      Object ret = null;
      try {
        ret = Invoker.invoke(this,
           $SchemeClass, "foo", args);//(D)
        if(ret != null) //(E)
          return ((Number)ret).intValue();
      } catch (Throwable e) {
        SE.throwWhatever(e);
      }
    }
    return super.foo(p0, p1); //(F)
  }
}
```

このメソッドは，Java から Scheme のメソッドを呼び出すためのインタフェースを提供する．A′ のインスタンスが Java に渡され，Java がメソッド `foo(int, Object)` を呼び出すと，Java の動的メソッド検索により，A′ でオーバライドした上記のメソッドが呼び出される．メソッド `foo` ではまず，インスタンス変数 `$SchemeClass` が `null` でないかを調べる（A）．`$SchemeClass` には，A′ のインスタンス生成時に，`SchemeClass` オブジェクトが代入されるため，通常は必ず `else` 節に制御が移る（C）．しかし，A′ のコンストラクタから，上記のメソッド `foo` が呼び出される場合，変数 `$SchemeClass` には `SchemeClass` オブジェクトが代入されておらず，`null` となっている．この場合，`super` 呼び出しによって，親の Java クラ

スに制御を渡す（B）．else 節では，まず，メソッド Invoker.find によって，同名の Scheme メソッド foo が定義されているかどうかを調べる．定義されていなければ，false が返されるので，super.foo(p0,p1) により，Java のメソッドを呼び出す（F）．逆に，同名の Scheme メソッド foo が定義されている場合は，Scheme のメソッドを検索するために，メソッド foo への引数を配列 args に格納し，Invoker.invoke メソッドに制御を渡す（D）．同名のメソッドが存在しても，引数が一致しない場合には，やはり親の Java クラスのメソッドを呼び出す必要があるため（F），ここではシグネチャが完全に一致するか否かを調べる．一致した場合にのみ，Scheme メソッドを呼び出す．検索の成功/失敗の判断は，Invoker.invoke の結果が null かどうかを判定して行う（E）．検索が成功している場合，Invoker.invoke の返値として，Scheme のメソッドの実行結果，つまり，null 以外のオブジェクトが返ってくる．ただし，メソッド foo の返値の型が参照型で，かつ，Scheme メソッドの実行結果が null となる場合が考えられる．しかし，Scheme レベルでは，null を表現するデータとして，Symbol.nullObject オブジェクトを用意しているため，実際に返値が null となることはない．たとえば，foo の返値の型が java.lang.String の場合，（E）において，次のような変換を行う．

```
if(ret != null)
  return (java.lang.String)
   (ret == Symbol.nullObject ? null : ret);
```

5.3 Scheme からのメソッド呼び出し

send 式によるメソッド呼び出しは，ターゲットのクラス，メソッド名，引数の型に基づいて動的に行う．呼び出すメソッドは，以下の 2 種類に分類できる．

- Java メソッド
  Java で定義されたメソッド．本体は Java の Method オブジェクトであり，Java VM のオペランドスタック上で実行される．呼び出しには，Java Core Reflection API[10] を用いる．
- Scheme メソッド
  Scheme 環境で定義されたメソッド．本体は Scheme のクロージャ，すなわちぶぶバイトコードであり，ぶぶバイトコードインタプリタ上で実行される．呼び出しには，SchemeClass オブジェクトのメソッドテーブルに対する動的検索を用いる．

ターゲットのクラスが Java クラスの場合，Java メソッドを呼び出す．ターゲットのクラスが Scheme クラスの場合，動的検索を用いて Scheme メソッドを呼び出す．Scheme メソッドが見つからなければ，Java メソッドを呼び出す．以下では，ターゲットのクラスが Scheme クラスの場合を考える．

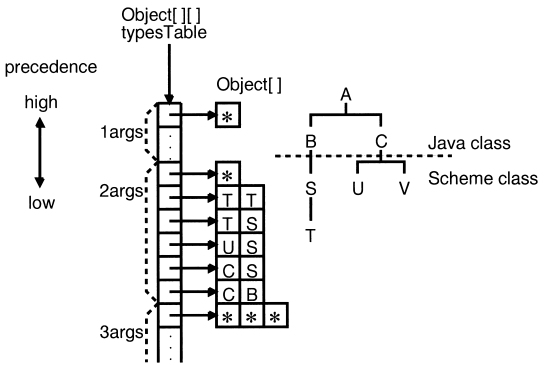

図 8 メソッドオーバーローディング機構
Fig. 8 Data structure for method overloading.

動的基底クラス A′ では，5.2 節で述べたメソッド foo とは別に，下記のメソッド $CSMfoo を定義する（CSM：Call Super Method）．

```
public int $CSMfoo(int p0, Object p1) {
  return super.foo(p0, p1);
}
```

A の Java メソッド foo は，A′ 上でオーバライドされている．このため，A′ のインスタンスから，直接的に A のメソッド foo を呼び出すことはできない．メソッド $CSMfoo は，Scheme から A のメソッド foo を呼び出すためのインタフェースを提供している．

send 式がメソッド foo を呼び出す場合，A′ の変数 $SchemeClass から，SchemeClass オブジェクトのメソッドテーブルを参照する．Scheme レベルでメソッド foo がオーバライドされていない場合，A′ のメソッド $CSMfoo を経由して，間接的に A のメソッド foo を呼び出す．

次に，引数の型を用いて最も特定化されたメソッド（the most specific method）を決定する機構について説明する．**図 8** に示すように，引数の型に基づいてメソッドを 1 列に順序付けることにより，バックトラックのないメソッド検索を実現している．ぶぶオブジェクトシステムは，Java と同様に単一継承のメカニズムを採用しているため，クラス階層は完全な木構造となる．メソッド定義時に，このような直列化を行っておくことにより，配列 typesTable を先頭から順に走査していき，最初に適合したメソッドを最も特定化されたメソッドと見なすことができる．

5.4 コンストラクタ

Java クラスと Scheme クラスとの間には，コンストラクタの振舞いに関して，以下のような相異点が存

在する．

- Java のコンストラクタは返値を持たないが，Scheme のコンストラクタは生成したインスタンスを返す．
- Java では，あるクラスで定義（この場合はオーバライド）したメソッドを，そのスーパクラスのコンストラクタから呼び出すことが可能である．一方，Java クラスのメソッドを Scheme クラスでオーバライドしても，Java クラスのコンストラクタから呼び出すことはできない．

Java 言語の `new` 式が行う処理は，Java VM レベルでは，次の 2 ステップに分けて実行される．

(1) オブジェクト生成

(2) コンストラクタ呼び出し

インスタンスの実体となるオブジェクトは，コンストラクタが呼び出される前に生成され，メンバ変数やメソッドテーブルの初期化が行われる．このため，コンストラクタ内部では，変数 `this` を通して，オブジェクト自身を参照することが可能となる．

一方，Scheme クラスのインスタンス生成では，引数の型を基に，`SchemeClass` オブジェクトの持つコンストラクタテーブルを検索し，実行すべきコンストラクタを動的に決定する．オーバローディングの機構は，Scheme メソッド呼び出しの場合と同様である．ただし，コンストラクタの *body* 部の先頭で，`super` 呼び出しによって，Java クラスのコンストラクタが呼び出される場合に注意する必要がある．例として，下記の Java クラス A を考える．

```
public class A {
  int x;
  double y;
  public A(int x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

ここで，A のサブクラス B を定義する．

```
(defclass B (A))
(defmethod B M () ...)
(defmethod B B ((x int) (y double))
           (super x y)
           (send this M))
```

B のコンストラクタ内部の変数 `this` には，B のインスタンス，すなわち，A の動的基底クラス A′ のインスタンスがバインドされなければならない．しかし，実際にインスタンスが生成されるのは，`(super x)` によって A のコンストラクタが呼び出された後である．このため，コンストラクタ B を呼び出す時点では，変数 `this` が参照すべきオブジェクトを生成することができない．そこで，ぶぶコンパイラは，コンストラクタ B の *body* 部の定義を，以下のように置き換えてコンパイルを行う．

```
(lambda (*dbc* *sc* *class* *slots* x y)
   (let ((this
          (invoke-constructor *dbc*
                              *sc*
                              *class*
                              *slots*
                              x y)))
     (send this M)
     this))
```

ここで，`lambda` 式の各引数は次のとおりである．

`*dbc*` インスタンス生成に用いる動的基底クラス A′．

`*sc*` コンストラクタの検索を行う `SchemeClass` オブジェクト．

`*class*` A′ のインスタンス変数 `$SchemeClass` に格納するための，`SchemeClass` オブジェクト．

`*slots*` A′ のインスタンス変数 `$slots` に格納するための，メンバ変数リスト．

4.3 節で述べたように，Scheme のメンバ変数は，A′ のインスタンとは別のオブジェクトに格納する．このため，コンストラクタを呼び出す前に生成し，引数 `*slots*`を通してコンストラクタ本体に渡すことが可能である．

関数 `invoke-constructor` は，`*sc*`に渡される `SchemeClass` オブジェクトのコンストラクタテーブルを検索し，最も特定化されたコンストラクタを呼び出す．そのコンストラクタは，再び `invoke-constructor` を呼び出すため，コンストラクタの検索と呼び出しが，スーパクラスへ遡って繰り返される．`*sc*`が参照する `SchemeClass` オブジェクトの親クラスが Java クラスとなった時点で，`*dbc*`が指す動的基底クラス A′ を用いて，Java クラス A のコンストラクタを呼び出す．Java クラスのコンストラクタ呼び出しには，Reflection API を使用する．この段階で，Scheme クラス B のインスタンスの実体が生成される．次に，引数`*class*`および`*slots*`を通して得られる，B の `SchemeClass` オブジェクトとメンバ変数を，A′ のインスタンス変数 `$SchemeClass` および `$slots` に，それぞれ代入する．最後に，Scheme クラスのコンストラクタの残りの部分が，スーパクラスから順次実行される．

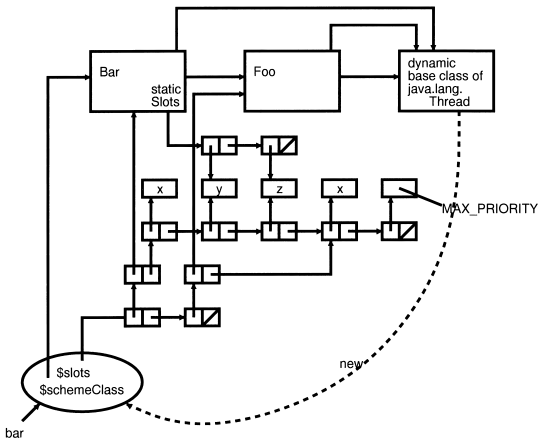

図 9 リストを用いたスロットの構成
Fig. 9 Slot representation by using list structure.

5.5 変数参照

Scheme クラスのメンバ変数は，リスト構造を用いて保持する．slot-ref 式，slot-set!式による変数参照では，最初に $slots 上の Scheme のメンバ変数を走査し，存在しなければ，Java のメンバ変数を参照するという手順を用いる．これにより，Scheme レベルでの変数の隠蔽を実現している．

以下の例では，java.lang.Thread クラスを拡張して Scheme クラス Foo を定義し，さらに Foo を拡張してそのサブクラス Bar を定義している．

```
>(defclass Foo (java.lang.Thread)
   (x int 1)
   (MAX_PRIORITY int 15))
Foo
>(defclass Bar (Foo)
   (x double 2.0)
   (y String (:is static) "hello")
   (z SchemeClass (:is static)
                  (class Foo)))
Bar
>(define bar (new Bar))
bar
```

クラス Bar のインスタンス bar のスロット構造を**図 9** に示す．Scheme クラス Foo のインスタンス変数 x は，Scheme クラス Bar において，同名の変数が定義されて隠蔽されている．クラス Foo に属する x の値を得るためには，以下のように，クラス名 Foo を指定してアクセスする．

```
>(slot-ref bar x)
2.0
>(slot-ref bar (Foo x))
1
```

同様に，Thread クラスのクラス変数 MAX_PRIORITY は，Scheme クラス Foo における同名の変数定義により隠蔽されている．Thread クラスの MAX_PRIORITY の値を得るためには，以下のように，クラス名 Thread を指定してアクセスする．

```
>(slot-ref bar MAX_PRIORITY)
15
>(slot-ref bar (Thread MAX_PRIORITY))
10
```

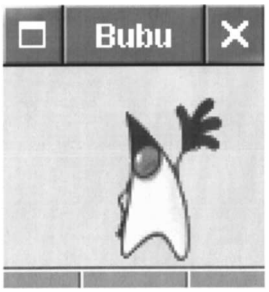

図 10 Duke アニメーション
Fig. 10 Duke animation.

6. オブジェクトシステムの実行例

実際にぶぶ上で作成したアプリケーションの一例として，Duke アニメーション実行の様子を**図 10** に示す．Scheme のプログラムは付録 A.1 に示す．この例では，java.awt.Frame クラスを拡張して，クラス Duke を定義し，その後，メソッド paint を追加定義している．Duke クラスのインスタンスメソッド repaint を呼び出すと，Java 側に制御が移り，フレームの再描画が行われる．その際，Java から，先ほどオーバライドを行った Scheme のメソッド paint が呼び出され，再び Scheme 側に制御が移る．

プログラム中のグローバル変数*interval*は，アニメーション描画の時間間隔を保持している．ユーザはこの変数の値を操作することによって，アニメーション実行中に描画間隔を任意に変更することができる．

7. おわりに

本稿では，Java 仮想マシンが提供するデフォルトのクラスローダを用いてぶぶオブジェクトシステムを実装する手法を述べた．今後の課題として，代理オブジェクトを用いたインタフェースの実現が考えられる．動的基底クラスそのものにインタフェースを実装する現在の方式では，Scheme クラスに実装されるインタフェースによって，対応づける動的基底クラスが変わっ

てしまう．Java のメソッドが引数の型としてインタフェース型を要求する場合，要求されるインタフェースを実装した代理オブジェクトを用いて Scheme クラスのインスタンスをラップしてやれば，動的基底クラス自体はインタフェースを実装せずに済む．

今後，Java 処理系の進化とともに，並列プログラミングとそのデバッギング環境，分散処理環境の改善を図り，実際的な応用を通してポータビリティに富んだ Scheme システムとして完成させていく計画である．ぶぶはまもなく公開ソフトウェアとなる予定である．

参 考 文 献

1) Yuasa, T.: An Object-oriented Scheme System Bubu with Seamless Interface to Java, *Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T.(Eds.), pp.101–121, World Scientific (2000).
2) IEEE: *IEEE Standard for the Scheme Programming Language* (*IEEE P1178*) (1991).
3) Bothner, P.: *Kawa: Compiling Scheme to Java*, Lisp Users Conference in Berkeley, CA (1998). http://www.gnu.org/software/kawa/.
4) Anderson, K., Hickey, T. and Norvig, P.: SILK. http://www.sc.brandeis.edu/silk/silkweb/index.html.
5) Travers, M.: Skij. http://www.alphaworks.ibm.com/tech/skij.
6) Queinnec, C.: Jaja: Scheme in Java. http://www-spi.lip6.fr/~queinnec/WWW/Jaja.html.
7) 倉林則之，湯淺太一，小宮常康：プログラムの部分移送に基づく遠隔実行機構とその知的インタフェースへの応用：Remote Execution by Partial Program Transfer and its Application to Intelligent Interfaces, Japan Lisp User Group Meeting (2000).
8) Steele, G.L.: *Common Lisp The Language*, Second Edition (1984).
9) ISO/IEC 13816:1997: *Information technology—Programming languages, their environments and system software interfaces—Programming language ISLISP* (1997).
10) Sun Microsystems: Java Core Reflection API and Specification, http://java.sun.com/products/jdk/1.1/docs/guide/reflection/index.html (1996).

付 録

A.1 Duke アニメーションプログラム

6 章で述べた Duke アニメーションのデモプログラムを示す．

```
(import "java.awt.*")
(define (create-image-circular-list)
  (define tk (send (class Toolkit)
                   getDefaultToolkit))
  (define (cicl-aux x idx)
    (if (> idx 0)
        (cicl-aux
         (cons (send tk getImage
                     (new java.net.URL
                          (string-append
 "file:/usr/docs/java/tutorial-ja/images/duke/T"
                      (number->string idx) ".gif")))
               x)
         (- idx 1))
        (let loop ((hd x) (p x))
          (if (null? (cdr p))
              (set-cdr! p hd)
              (loop hd (cdr p))))))
  (cicl-aux '() 9))
(define *images* (create-image-circular-list))
(define (get-image)
  (let ((img (car *images*)))
    (set! *images* (cdr *images*)) img))
(defclass Duke (Frame))
(defmethod Duke Duke ((s String)) (super s))
(defmethod Duke paint ((g Graphics))
  (send g drawImage (get-image) 30 30 this))
(define *interval* 100)
(define (main)
  (let ((d (new Duke "Bubu")))
    (send d setSize 100 100)
    (send d setBackground
          (slot-ref (class Color) yellow))
    (send d show)
    (dotimes (i 1000 #t)
      (send (class Thread) sleep *interval*)
      (send d repaint))))
```

(平成 12 年 10 月 25 日受付)
(平成 13 年 1 月 22 日採録)

窪田 貴志

1976 年生．1999 年京都大学工学部情報学科卒業．2001 年，同大学大学院情報学研究科修士課程修了．同年株式会社 PFU 入社．

湯淺 太一（正会員）

1952 年神戸生．1977 年京都大学理学部卒業．1982 年同大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987 年豊橋技術科学大学講師．1988 年同大学助教授，1995 年同大学教授，1996 年京都大学大学院工学研究科情報工学専攻教授．1998 年同大学院情報学研究科通信情報システム専攻教授となり現在に至る．理学博士．記号処理，プログラミング言語処理系，超並列計算に興味を持っている．著書「Common Lisp 入門」（共著),「Scheme 入門」,「C 言語によるプログラミング入門」ほか．ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．

倉林 則之（正会員）

1968 年生．1990 年豊橋技術科学大学工学部情報工学課程卒業．1992 年同大学院工学研究科情報工学専攻修士課程修了．同年富士ゼロックス株式会社入社．1999 年より ATR 環境適応通信研究所出向中．ヒューマンインタフェース学会会員．

八杉 昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒業．1991 年同大学院電気工学専攻修士課程修了．1994 年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

小宮 常康（正会員）

1969 年生．1991 年豊橋技術科学大学工学部情報工学課程卒業．1993 年同大学院工学研究科情報工学専攻修士課程修了．1996 年同大学院工学研究科システム情報工学専攻博士課程修了．同年京都大学大学院工学研究科情報工学専攻助手．1998 年より同大学院情報学研究科通信情報システム専攻助手．博士（工学）．記号処理言語と並列プログラミング言語に興味を持つ．平成 8 年度情報処理学会論文賞受賞．