プログラミング言語処理系の部品化

一杉　裕志

■ 研究のねらい

本研究は、プログラミング言語処理系を部品化することにより、いままで進歩の遅かったプログラミング言語の進歩速度を、飛躍的に向上させることを目的とする。部品化により、部品単位での言語機能の研究開発を容易にすると同時に、自由競争の原理による言語機能の性能向上・多様化を図る。

本研究の背景として、以下に述べるような状況がある。現在のプログラミング言語は、巨大な一枚岩のシステムである。特に近年、実用的プログラミング言語に必要とされる言語機能は増大し、プログラミング言語の仕様は肥大化の一途をたどっている。このことは、以下のような問題を引き起こしている。

1．　言語研究者にとって、新しい言語機能の実験を非常に難しくする原因になっている。

新たな言語機能が現実のアプリケーションの中でどのように使われるかを調べるためには、必要な言語機能を一通り持った実験用言語処理系を実装しなければならない。

しかし、必要な言語機能が増大した今日では、それは極めて困難になってきている。他の手段としてソースコードが公開されている言語処理系の一部を改造して新たな言語機能を追加する方法もあるが、言語処理系が巨大で一枚岩なため、それも容易ではない。

2．　プログラマーにとって、自分が望む言語機能が、手元の言語処理系になかなか採用されない、という問題が生じている。

すべての言語機能は「だきあわせ」の形でしか提供されない。プログラマーには、自分の欲しい機能を選択する自由はほとんどない。使用するプログラミング言語を選択した段階で、利用できる言語機能は決まってしまう。時には、言語設計者の好みによって機能が選択され、それがプログラマーに押し付けられることになる。最新の言語研究の成果が、手元の言語処理系に採用される確率は極めて小さく、採用されるとしても何年もの時間がかかってしまう。プログラミング言語の言語仕様が巨大で一枚岩なために、新たな言語機能を追加することが非常に難しいからである。

これらの問題を解決するために、本研究では、プログラミング MixJuice を開発した。この言語は、それ自身が「部品化された言語処理系」であると同時に、「部品化された言語処理系」を構築するための記述言語でもある。

■ 研究成果

1．「モジュール = 差分」

MixJuice では、「オリジナルのシステムに対する差分」をモジュールとして扱う。差分とは、 patch

file のようなものであり、オリジナルのシステムに対する変更部分だけを抜き出したものである。従来のオブジェクト指向言語における継承との違いは、差分の追加の対象が単一のクラスではなく、クラス階層全体である、という点である。

プログラマは複数のクラスにまたがった機能を１つのモジュールとして表現することで、それを再利用可能にできる。

各モジュールは分割コンパイルして、それぞれソースコード無しで配布することができる。

コンパイル時には、モジュール単位で型チェックが行われる。プログラムを実行する際には、プログラムのユーザが、使用するモジュールを指定する。指定された複数のモジュールは、以下のように１つのプログラムにリンクされる。まず各モジュールは、それぞれの依存関係に従ってトポロジカルソートされる。そして、ソート結果の先頭のモジュールに対して、２番目、３番目、・・・のモジュールで定義された差分が順に追加される。そして最終的にできあがったプログラムがリンク結果となる。リンク時にも、型の整合性が検査されるため、リンク結果のプログラムの安全性が保証される。

このように、MixJuice で書かれたプログラムのユーザは、複数のモジュールの中から自分が必要とする機能を選択して組み合わせて利用することができる。これは、従来の条件コンパイル(C プリプロセッサの　#ifdef)の用途の１つに相当する。

従来の patch や　#ifdef は文字列レベルで処理を行なうが、 MixJuice は、言語レベルで処理を行なうため、より安全である。

２．モジュール定義の構文

モジュール m の定義は次のように記述する。

```
module m          // このモジュールの名前
  extends m1, m2     // 差分の追加の対象となるモジュール
{  // モジュール本体（追加する差分）
   class S {...}  // 既存のクラスへの差分の追加
   define class A extends S {...}    // 新たなクラスの追加
   ...
}
```

extends　宣言は、モジュール m が差分を追加する対象となるモジュールの名前を宣言する。従来のオブジェクト指向言語におけるスーパークラスに似ているため、指定されたモジュールを super-module と呼ぶ。上の例では、 m1 と m2 という２つの super-module を指定している。この場合、モジュール m は、 m1 と m2 の両方を組み合わせたプログラムに対する差分を定義することになる。

モジュール本体には、新たに追加するクラスや、既存のクラスに対する差分を定義する。記述方法は Java 言語とほぼ同じだが、 define というキーワードを指定する場合がある点が違う。 define というキーワードの付いたクラス・メソッド・フィールド定義は、新たなクラス・メソッド・フィールドを追加することを示す。 define　がついていないクラス定義・メソッド定義は、既存のクラスやメソッドに対する差分を定義することを示す。

3．名前空間の継承

extends 宣言は、名前空間の継承という機能も持っている。 extends 宣言で指定された各 super-module 内から参照可能な全ての名前（クラス名、フィールド名、メソッド名）は sub-module 内からも参照可能である。Java 言語が有する package と nested class という機構は、MixJuice には存在しない。これらの機構は名前空間のネスト構造を表現するためのものだが、ほぼ同等なことは、extends 宣言による名前空間の継承で、表現可能である。ネストの内側の名前空間は、外側の名前空間を継承していることに相当するからである。

さらに MixJuice では、名前空間の多重継承を用いることにより、ネスト構造だけでなく、重なりを持つような、より一般的な名前空間の構造も表現可能である。

```
abstract class S {
  abstract boolean equals(S x);
  boolean equalsToA(A x) {
    return false; }
  boolean equalsToB(B x) {
    return false; }
}
```

```
class A extends S {
  boolean equals(S x) {
    return x.equalsToA(this); }
  boolean equalsToA(A x) { ... }
}
```

```
class B extends S {
  boolean equals(S x) {
    return x.equalsToB(this); }
  boolean equalsToB(B x) { ... }
}
```

図 1　Java によるダブルディスパッチの記述例

```
module skeleton {
  define abstract class S {
    define abstract boolean equals(S x);
  }
}
```

super module　　super module

```
module a
  extends skeleton {
  class S {
    boolean equalsToA(A x) {
      return false; }
  }
  define class A extends S {
    boolean equals(S x) {
      return x.equalsToA(this); }
    define boolean equalsToA(A x) { ... }
  }
}
```

```
module b
  extends skeleton {
  class S {
    boolean equalsToB(B x) {
      return false; }
  }
  define class B extends S {
    boolean equals(S x) {
      return x.equalsToB(this); }
    define boolean equalsToB(B x) { ... }
  }
}
```

図 2 MixJuice によるダブルディスパッチの記述例

４．プログラム例

４．１　例１：ダブルディスパッチ

差分ベースモジュール用いることでうまくモジュール化できるプログラムの例として、ダブルディスパッチと呼ばれる技法を用いたプログラムについて説明する。ダブルディスパッチは、Java のようなシングルディスパッチの言語で、CLOS などの言語のマルチメソッドに近いものを実現する技法である。

まず、図１は、普通の Java 言語によるプログラムの記述である。抽象クラス S のメソッド equals は、S のサブクラスの２つのインスタンスが値として等しいかどうかを調べるメソッドである。x1.equals(x2) が呼び出されると、このメソッドは x1 のクラスに応じてディスパッチされ、その結果 equalsToA または equalsToB が呼び出される。そしてこれらのメソッドは、x2 のクラスに応じてディスパッチされる。もし２つのオブジェクトが異なるクラスであれば、これらのメソッドは false を返し、同じクラスであれば、A か Bの実装に依存した方法で同値性が判断され、true か false が返される。

クラスベースモジュールのもとでダブルディスパッチを記述すると、抽象クラス S の定義中にサブクラス A、B の名前が現れるため、モジュラリティが悪い。そのため、新たなサブクラス C を追加するためには、S の定義を編集しなければならないという問題がある。

全く同じプログラムを MixJuice を使って記述したものが図２である。クラス名 A、B に依存したコードを S の定義から分離して記述することができるため、よりモジュラリティが高い。また、同様のモジュールを追加することで、既存のモジュールを再コンパイルすることなしに、S にサブクラス C を追加できる。このように、MixJuice では、すでに存在するクラスに対してメソッドを追加することが可能であるため、従来のオブジェクト指向言語で書かれたプログラムよりも拡張性が高くなる。

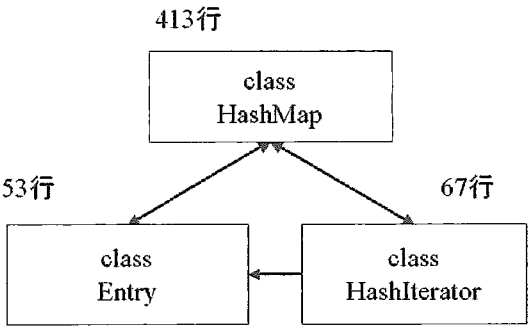

図 2　HashMap.java 内部のクラス間の依存関係

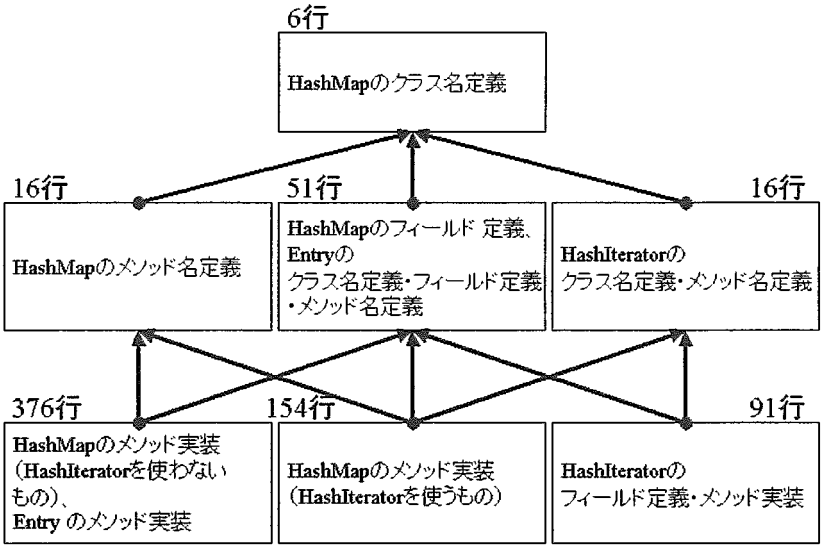

図 4 MixJuice を用いて書き直した HashMap.java のモジュール間の依存関係

４．２ 例２： HashMap

もう少し複雑な例として、JDK1.2 に付属するクラスである java.util.HashMap について述べる。

HashMap は、 inner class を巧みに用いることにより、外部には必要最低限の名前しか公開されておらず、その意味ではモジュラリティが高い実装になっている。しかし、 HashMap の内部は、モジュラリティの高い実装にはなっていない。図３は、ソースファイル HashMap.java に含まれる３つのクラス（そのうち２つは inner class ）の間の依存関係を示したものである。３つのクラスは相互に依存している上、お互いの field や method が直接参照できる関係にあり、コードの一部の修正が、ファイル内のどの部分に影響を与えるかが非常にわかりにくくなっている。

HashMap.java はコメントを除いて約５００行である。より高機能なクラス TreeMap のソースファイルは約１０００行もあり、そこで定義される５つのクラス間の依存関係はいっそう複雑である。これらのソースコードは、従来のクラスベースモジュール機構が、高機能なクラスの内部をモジュール化する能力を有していないことを示している。

図４は、HashMap のソースコードを MixJuice で書き直し、７つのモジュールに分割した場合の、モジュール間の依存関係を示したものである。このように、依存関係にサイクルがなく、しかもサイズが小さく安定したモジュールのみにメソッド実装が依存する形に、モジュール分割することができた。これにより、プログラムの保守性は大きく向上すると考えられる。

差分ベースモジュールではクラスとモジュールの機構が独立しているため、このように３つのクラスを７つに分割して記述することが可能である。また、モジュールの分割によってプログラム自体の動作が変化することはないため、プログラムの動作の理解のしやすさや実行効率が損なわれる心配もない。

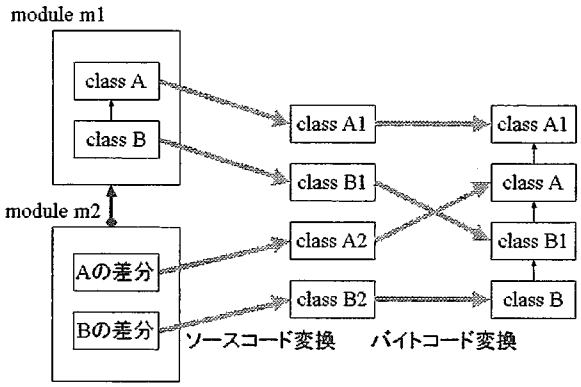

図 5 プリプロセッサとポストプロセッサによる実装

5．実装

MixJuice で書かれたプログラムのソースコードは、プリプロセッサ（ソースコード変換）、Java コンパイラ、ポストプロセッサ（バイトコード変換）によって処理され、最終的に Java 言語のバイトコードに変換される。

クラスの差分の追加を実現するには、JavaVM が本来持っている継承メカニズムを利用する（図５）。モジュール本体に含まれるクラスの差分は、プリプロセッサによって、ダミーのスーパークラスを持つクラスに変換される。変換結果のソースコードは、Java コンパイラによって普通にコンパイルされる。リンク時には、各差分が一列のクラス階層になるように、各差分のスーパークラスがポストプロセッサによって書き換えられ、適宜クラス名も変更されて、「差分を追加した結果」に相当するクラス階層が構築される。

なお、プリプロセッサとポストプロセッサは、Java コンパイラによる型チェックと、JavaVM による verifier のチェックを通るように、注意深く変換を行っている。

■ 今後の展開

今後の展開としては、MixJuice 言語の完成度の向上と、MixJuice 言語を用いた「部品化された言語処理系」の再実装を考えている。

MixJuice 言語に関しては、現在の言語仕様および実装は不完全な点が多いため、言語処理系を再び０から書き直しブートストラップすることを考えている。これにより、処理速度・安定性の向上、ユーザによりわかりやすいエラー処理などを達成する。また、パラメタ付きモジュール、表明検査などの新たな機能追加を行なう。

「部品化された言語処理系」については、現在公開中のソフトウエアである「拡張可能 Java プリプロセッサ EPP」の次期バージョンという形で発展させたいと考えてる。EPP の現在のバージョンは、変換速度、API の統一性、安全性などに問題がある。MixJuice 言語を用いて EPP を書き直すことにより、これらの問題を解決しようと考えている。

■ 発表リスト

雑誌論文

(1) 一杉裕志:コンピュータソフトウエア
「シンプルかつ強力なモジュール機構を有するオブジェクト指向言語 MixJuice の提案」
(2001 年 11 月 掲載予定)。

その他論文・口頭発表

(1) 一杉裕志:
“シンプルかつ強力なモジュール機構を有するオブジェクト指向言語 MixJuice の提案”、
ソフトウエア科学会第１７回大会、Sep. 2000.

(2) Yuuji ICHISUGI:
“MixJuice : An object-oriented language with simple and powerful module mechanism”、
extended abstract of OOPSLA2000 poster session, Oct 2000.

公開ソフトウエア URL

″The programming language MixJuice″
http://staff.aist.go.jp/y-ichisugi/mj/