拡張可能 Java プリプロセッサ EPP の型チェック機構フレームワーク

一杉裕志
ichisugi@etl.go.jp
電子技術総合研究所／科学技術振興事業団 さきがけ研究２１

概要

筆者が開発している拡張可能 Java プリプロセッサ EPP が有する、型チェック機構フレームワークについて述べる。本フレームワークを用いることによって、連想配列、多重継承、演算子オーバーローディング、パラメタ付きクラスなど、様々な言語機能が、 EPP に追加する部品 (EPP plug-in) の形で実装可能になる。型情報を利用した変換処理には、(1) 型同士・ファイル同士の相互参照の問題、(2) 変換によって型自身が変化し得る問題、(3) 分割コンパイル・自動再コンパイルの問題などが伴う。本フレームワークは *FileSignature* という概念を導入することでこれらの問題を自動的に処理し、plug-in プログラマーの負担を軽減している。

1 イントロダクション：プログラム言語の「ＰＣ互換機」化

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. 言語研究者が新しい言語機能の実験をすることを、非常に難しくしている。

   新たな言語機能が現実のアプリケーションに対して有効に機能するか調べるためには、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。別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原因になっている。

   すべての言語機能は「だきあわせ」の形でしかプログラマーに提供されない。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。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。

このような問題を解決するため、筆者はプログラミング言語処理系の部品化を試みている。これにより、パーソナルコンピュータに起きた進歩速度の向上を、プログラミング言語の世界にも起こすことを目的としている。パーソナルコンピュータのハードウエアは、最近１０年程度の間に飛躍的な機能向上を果たしている。これはパーソナルコンピュータの標準的アーキテクチャと個々の部品を接続する規格が公開されたことをきっかけとする。その結果、部品を製造する企業が多数現れ、自由競争の原理により、部品の高機能化・多様化が促進された。同様に、プログラミング言語処理系を部品化し、部品を接続する標準的なインターフェースを定めることにより、部品単位での言語機能の開発を容易にすることを目指している。

筆者はすでに、 mixin を用いて「部品化された再帰下降パーザ」を構築する方法を提案し、拡張可能プリプロセッサ EPP 上でその有効性を確かめている [9, 10]。

本論文では、プログラミング言語処理系の部品化に向けた次のステップとして設計・実装した、 EPP の型チェック機構について述べる。

2 EPP の概要

EPP は、 Java 言語に拡張機能を追加できるようにするプリプロセッサである。使いたい EPP plug-in を Java ソースコードの先頭で `#epp` *name* と書いて指定することで、 Java の文法を拡張し、新しい機能を追加することができる。 plug-in 同士の衝突が起きなければ、複数の plug-in を同時に取り込むこともできる。プリプロセッサが出力する Java ソースコードは、普通の Java コンパイラを通る。 EPP は入力プログラムの行を変えずに出力するため、普通にソースレベルのデバッグをすることができる。

EPP は、 Java 言語を拡張するためのプリプロセッサとしてだけでなく、言語研究者による新しい言語機能実験の道具、拡張 Java の実装のためのフレームワーク、 Java 言語のソースコード解析・変換ツールのためのフレームワークとして用いることもできる。

EPP のソースコード自身も「 EPP で拡張された Java 」で書かれており、「Common Lisp[18] で書かれた EPP」でブートストラップされる。コンパイルされたバイトコードは Java が動く環境ならばどこでも動く。

```
#epp jp.go.etl.epp.AssociationArray
class Test {
  public static void main(String[] args){
    // Declaring and initializing
    //an association array variable.
    String[String] a = new String[String];

    a["key1"] = "val1";
    a["key2"] = "val2";
    System.out.println(a["key1"].equals("val1"));//true

    // Type-safe assignment.
    Object[String] a1 = a;
    a = new String[Object];

    // Because of contra-variant rule,
    //the following code causes type error.
    // Object[Object] a2 = a;
  }
}
```

図 1: 連想配列 plug-in の使用例

```
#epp jp.go.etl.epp.MultipleInheritance
class MITest {
  public static void main(String[] args){
    C obj = new C();
    System.out.println(obj.m11() == 11); // true
    System.out.println(obj.m12() == 12); // true
    System.out.println(obj.m21() == 21); // true
    System.out.println(obj.m22() == 22); // true
  }
}
class C1 {
  int m11() { return 11; }
  int m12() { return 12; }
}
class C2 {
  int m21() { return 21; }
  int m22() { return 22; }
}
class C extends C1, C2 {
}
```

図 2: 多重継承 plug-in の使用例

EPP 設計目標は、「広い範囲の拡張が行なえること」、「複数の拡張機能を同時に利用できること（以下 composability と呼ぶ）」の 2 つである。EPP は、「ソースコードの編集」に匹敵する高い拡張性を持ち、なおかつ十分に高い composability を持った拡張可能システムである。

EPP のアーキテクチャは、Java 言語には依存しておらず、他の言語の解析・変換処理システムフレームワークとしても応用することができる。

3 EPP plug-in の例

3.1 Association array plug-in

図 1は、連想配列を実現する plug-in の使用例である。連想配列は、perl などの言語が持つデータ構造で、その実体はハッシュ表だが、プログラマーからは、配列と同じインターフェースで、扱うことができる。図 1の `String[String]` は、キーと値が共に String 型であるような連想配列を表す型名である。この plug-in は、contra-variant rule に従った型的に安全な代入が行なえる仕様を採用している。

連想配列へのアクセスは、プリプロセッサによって、ハッシュ表へのアクセスとキャストを使った式に変換される。

3.2 多重継承 plug-in

図 2は、多重継承を実現する plug-in の使用例である。

多重継承を行なっているクラスは、プリプロセッサによって、2 番目以降の「にせもののスーパクラス」をインスタンス変数として有するクラスに変換される。また、にせもののスーパークラスで定義されているメソッドが呼ばれたら delegate するメソッドが、クラスに追加される。

（なお、このような単純な実装では、this の値が正しくなくなったり、C のインスタンスを C2 型の値として使えないなどの問題がある。本当の多重継承に近付けるためには、もう少し工夫が必要である。）

4 型情報を利用した変換処理に伴う問題

Java 言語は、クラス間の相互依存を許している。クラス間の相互依存は、Java 言語のコンパイラの実装者が気をつけなければならない問題である。しかし、Java 言語の場合は、言語仕様が固定であるし、構文解析するだけでクラスの signature が確定するので、問題は比較的簡単に解決することができる。

EPP の型チェック機構フレームワークは、Java コンパイラが持つクラス間相互依存の処理機構をより一般化して、plug-in プログラマー向けの API として提供する。このフレームワークは、以下に述べるような、型情報を利用した変換処理に伴う問題を解決している。

- プログラムの変換が他のクラスの signature に依存すると同時に、その変換がクラスの signature を変えるものだとしたら、どのような手順で変換を行なうべきだろうか？例えば、多重継承 plug-in の場合、delegation するメソッドを追加するためには、にせもののスーパークラスがどのようなメソッドを持つかが分からなければならない。一方で、多重継承を行なっているクラスは、delegation するメソッドが追加されるため、signature が変化する。多重継承を行なっているクラスをさらに他のクラスが多重継承しても、期待される通り正しく変換されるだろうか？
- ファイル間の相互依存がある場合でも分割コンパイルができるだろうか？例えばファイル F_a がクラス A_1, A_2 を持ち、ファイル F_b が、クラス B_1, B_2 を持つとする。A_1 の変換が B_1 に依存し、B_2 の変換が A_2 に依存するとすると、この 2 つのファイルを正しく変換できるだろうか？また、ファイル F_a の内容が更新された時、F_a だ

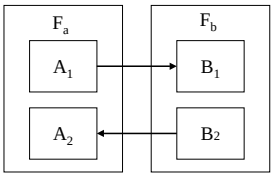

図 3: ファイル間の相互依存

けなく F_b も自動的に再コンパイルされるようにできるだろうか？

- 依存関係に解決不可能なループが含まれている時、システムは無限ループに陥らずに、それを検出できるだろうか？例えば、多重継承 plug-in を使用しているプログラマーが、間違ってクラス C のスーパークラスの 1 つにクラス C 自身を指定した時、それをエラーにできるだろうか？逆に、クラス C の変換が、本質的にクラス C の変換結果に依存していないにも係わらず、循環依存として拒否されることはないだろうか？

以上の問題は、もちろん、いずれも技術的には解決可能な問題である。フレームワークを設計する上で重要な目標はむしろ次の 2 点である。

1. これらのやっかいな問題をフレームワーク側が処理し、フレームワークのユーザ（plug-in の実装者）の負担を少しでも軽くする。
2. plug-in 実装者がやりたいことに不必要な制約を加えたりせず、可能な限り広範囲の言語拡張を可能にする。

本論文のフレームワークがこれらの目標をどの程度達成したかについて、十分な評価はまだ行なっていない。それを行なうためには、数多くのアプリケーションを実装してみる必要がある。しかし少なくとも現段階では、このフレームワークはうまく機能していると筆者は考えている。

本フレームワークがいかにして上に述べた問題を解決しているかについては、7章および 9章で述べる。

5 EPP の記述に必要な plug-in

EPP と EPP plug-in のソースコードは、それ自身が EPP で拡張された Java 言語で記述されている。この章では、EPP の記述に必要な plug-in である、Symbol plug-in, SystemMixin plug-in, BackQuote plug-in の機能について述べる。

5.1 Symbol plug-in

Symbol は lisp などの言語が持つデータ型であり、実行時における identity の比較が文字列よりも高速に行なえることを特徴とする。図 4 は、EPP の “Symbol plug-in” を使った Java プログラムの例である。

```
#epp jp.go.etl.epp.Symbol
import jp.go.etl.epp.epp.Symbol;

public class TestSymbol {
  public static void main(String args[]){
    Symbol x = :foo;
    Symbol y = :"+";
    System.out.println(x == :foo); // true
    System.out.println(y == :foo); // false
  }
}
```

図 4: Symbol plug-in を使ったプログラム例

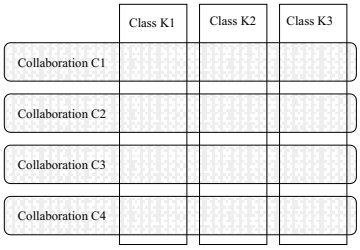

図 5: class と collaboration

Symbol のリテラルは、コロンとそれに続く identifier または文字列リテラルで表される。

Symbol は、EPP のソースコード中においては、抽象構文木や型の種類を表すタグを始めとして、様々な用途に使われている。

5.2 SystemMixin plug-in

5.2.1 collaboration-based design

クラスは再利用および情報隠蔽の単位として適当でないということが、以前から指摘されている [20]。この問題に関して [8][15] などの解決方法が提案されているが、近年特に collaboration-based design[21] の有効性を主張する論文がいくつか書かれている [21][19][16]。collaboration とは、図 5 にあるように、複数のクラスの中から関連の深い部分だけを集めたものである。

[19] では、C++ の template とクラスのネストの機能を用いて collaboration-based design を行なう方法を提案しており、この再利用の単位を mixin layer と呼んでいる。

SystemMixin plug-in は、Java 言語を拡張し、mixin layer とほぼ同じ機能（system mixin と呼んでいる）を実現可能にする plug-in である。mixin[2] は、flavors や CLOS などのシステムで使用可能な言語機能で、再利用可能なクラスの部品を定義するためのものである。system mixin は、複数のクラスの部品となる mixin をいくつかひとまとめにして扱いやすくしたものである。1 つの system mixin が 1 つの collaboration を実現する。

EPP および EPP plug-in のソースコードは、この system mixin の機能と、Java 言語の普通の機能

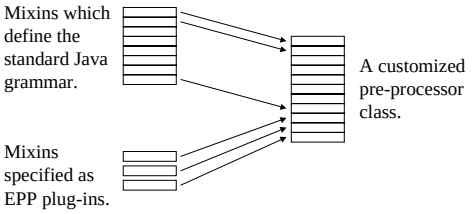

図 6: EPP を構成する mixin

を使った decorator pattern などを適宜使い分けて記述される。

EPP plug-in は、 system mixin と Java クラスの集合である。 plug-in は、分割コンパイルが可能であるため、言語拡張部品を、ソースコードを公開せずにバイナリ配布することも可能である。

5.2.2 EPP を構成する mixin

EPP の中心部分は、 `Epp` という名前のただ 1 つのクラスとして定義される。ただしクラス定義は複数の mixin に分割されている。 EPP を起動すると、まず標準の Java 言語パーザを構成する mixin と、ソースコードの先頭で指定された plug-in を構成する mixin をすべてつなぎ合わせて 1 つのプリプロセッサ (`Epp` という名前のクラス) を構築し、それを呼び出すことでソースコードの処理を開始する。

5.3 BackQuote plug-in

BackQuote plug-in は、 lisp の backquote macro と同じ機能を実現するもので、変換処理を記述する際、抽象構文木をソースコード中に簡単に埋め込めるようにするものである。 lisp の backquote macro と同様に、 “,” や “,@” の記号を使って、S 式の抽象構文木の中に任意の Java の式の値を埋め込むことができる。

6 拡張可能な再帰下降パーザ

拡張可能な再帰下降パーザの原理については、すでに [10] で述べているが、ここでは、簡単に説明する。

EPP が持つパーザは、 Yacc などのパーザ・ジェネレータは使わずに、再帰下降 [1] によって書かれている。このパーザは、左再帰ルールも含む広範囲の文法拡張を行なうための “hook” を多数提供している。例えば *Expression* という非終端記号に選択肢を追加する hook として、 `expressionTop`, `expressionRight`, `expressionLeft` というメソッドが定義されている。 Mixin を追加し、これらのメソッドの振舞いを継承し拡張することによって、非終端記号に新たな選択肢を追加することができる。

hook を使った拡張に加え、メソッドの再定義、バックトラック、文脈依存処理なども行なうことができ、極めて広範囲の文法拡張が行なえる。

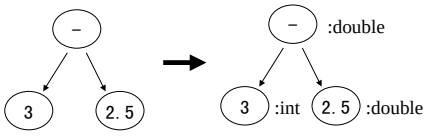

図 7: 抽象構文木への型情報の追加

7 拡張可能な型システム

7.1 型チェックパスの概要

型チェックパスでは、構文解析された抽象構文木の各ノードに、型情報を追加する。抽象構文木のノードは、 immutable なオブジェクトであり、 TypeChecker オブジェクトによって、型情報付きのノードに変換される。このように、型チェックパスは、オブジェクト指向パラダイムではなく、むしろ関数型プログラミングスタイルで記述されている。

7.2 Type オブジェクト

クラス Type は、 EPP の内部において型を表現するためのデータ型である。すべての型はタグを持っており、そのタグの名前によって、どの種類の型かを判別できる。

plug-in は、新たなタグを持つクラス Type のサブクラスを定義することによって、新たなデータ型を導入することができる。

Type オブジェクトは、型を表現するために必要な情報を保持するだけで、「型の意味」は一切定義しない。型の意味は、型同士の関係を定義するクラス `Epp` のメソッドと TypeChecker オブジェクトによって定義される。

クラス型を表す Type オブジェクトは、 :class というタグを持っている。クラス型に関するより詳しい情報を得るためには、クラス型を表す Type オブジェクトから、さらに ClassInfo というデータ型を取り出す。あるクラスがどのようなメソッドを持っているか、などの情報は、 ClassInfo から得ることができる。

ClassInfo は必要になった時に、 Type オブジェクトによって lazy に生成される。この機構のおかげで、相互参照するクラスも扱うことができる。クラスの中身を lazy に生成しようとしている、まさにその最中にクラスの中身が外部からアクセスされた時には、EPP は本質的に矛盾がある循環依存であると判断し、ユーザにエラーを報告する。例えば、あるクラスのスーパークラスに自分自身を指定するとこのエラーとなる。

ファイル F をコンパイルしている時に、あるクラス C の ClassInfo に初めてアクセスすると、「F は C が定義されているファイルに依存している」という情報が自動的に記録される。この情報は、更新されたファイルに依存しているファイルの自動再コンパイルのために使われる。

なお、 plug-in は ClassInfo のサブクラスを定義し、拡張されたクラス型を実装することができる。

7.3 TypeChecker オブジェクト

```
class ExpandOperatorOverloadingOfMinus
                    extends TypeChecker {
  public Tree call(Tree tree) {
    Tree[] newArgs = typeCheckArgs(tree);
    Type t1 = newArgs[0].type();
    if (t1.tag() == :class){
      // Return the AST of "e1.minus(e2)" .
      return `(invokeExp ,(newArgs[0])
               (id minus) (argumentList ,(newArgs[1])));
    } else {
      return orig.call(tree.modifyArgs(newArgs));
    }
  }
}
```

図 8: Operator overloading を実装する decorator

制御構造や演算子が、型をどのように扱うかを定義するものが、 TypeChecker オブジェクトである。

TypeChecker は、 call という名前のメソッドを持つ。これは、型情報のついていない tree を型情報付き tree に変換するメソッドである（図 7）。

TypeChecker は、ノードの種類と TypeChecker を関連付けるハッシュ表に登録されている。型チェックパスでは、ノードごとに、そのノードのタグに対応する TypeChecker を呼び出して、そのノードの型チェックを行なう。

TypeChecker は、 tree を引数にとって tree を返すというインターフェースさえ満たせばよく、マクロ展開などの処理を行なってもかまわない。 plug-in は、文法を拡張して制御構造や演算子を追加し、それを普通の Java 言語にマクロ展開する TypeChecker を定義することで言語拡張を実装できる。

すべての TypeChecker は、 decorator pattern [5] によって拡張することができる。図 8は、二項演算子 “-” の TypeChecker に追加する decorator の例である。この decorator は、左のオペランドの型がクラス型であるならば、

```
e1.minus(e2)
```

というメソッド呼び出し式にマクロ展開して返す。このように plug-in は、既存の制御構造や演算子の意味を拡張することもできる。

なお、 decorator pattern を用いているため、同時に適用された他の plug-in が同じ TypeChecker を拡張したとしても、システムは正しく動作する。つまり、高い composability を達成することができる。

7.4 TypeNameChecker オブジェクト

TypeNameChecker オブジェクトは、型チェックパスにおいて、型名（ “`String`” など）や、タイプコンストラクタ（“`String[]`” など）を表現する抽象構文木が、実際にどのような型を表すかを決定するオブジェクトである。

TypeNameChecker の使われ方は TypeChecker と同じで、メソッド call の引数として渡された Tree を、型情報付きの Tree に変換して返す。

```
boolean isSuperType(Type t1, Type t2){
  if (t1.tag() == :assocArray){
    if (t2.tag() == :assocArray){
      AssocArrayType a1 = (AssocArrayType)t1;
      AssocArrayType a2 = (AssocArrayType)t2;
      // Contra-variant rule.
      return isSuperType(a1.getKeyType(),
                         a2.getKeyType())
          && isSuperType(a2.getValueType(),
                         a1.getValueType());
    } else {
      return false;
    }
  } else {
    return original(t1, t2);
  }
}
```

図 9: isSuperType メソッドの mixin による拡張

plug-in は、標準の Java にないタイプコンストラクタを導入するために、新たな TypeNameChecker を定義して、テーブルに登録することができる。

7.5 型の間の関係

２つの型の間の関係など、システム全体に影響する型に関する意味を定義するためのメソッドが、 mixin によって拡張可能なメソッドとして定義されている。 plug-in は、 mixin を追加することによって型の間の関係を変更・拡張することができる。

図 9 は、連想配列 plug-in のソースコードの一部である。メソッド `isSuperType` は、第一引数の型が、第二引数の型の super type かどうかを判断するメソッドである。図 9 では、 contra-variant rule に従って、連想配列同士の間の関係を定義している。（プログラム中で `original` という名前のメソッド呼び出しがあるが、これは、 SystemMixin plug-in が導入した構文で、意味的には super class のメソッド呼び出しと同じである。）

7.6 新たな型の導入例

以上の拡張メカニズムを用いることによって、非常に幅広い範囲の言語拡張が、 composability の高い plug-in として実現可能になる。

具体例として、3章で述べた連想配列の実装のアウトラインについて述べる。この plug-in は、２１０行で実装されている。

- 連想配列型を構文解析できるように文法を拡張。
- `isSupetType` など、型同士の関係を定義するメソッドの拡張。
- 連想配列型を表現する、 Type のサブクラス、 AssociationArrayType の定義。
- 連想配列型を表す抽象構文木の意味を定義する TypeNameChecker の定義。この TypeNameChecker は、連想配列型が内部では AssociationArrayType 型として扱われ、出力されるコード中では、 `Hashtable` になるように実装される。

- 配列のアクセスと、代入文の TypeChecker を拡張する decorator の定義。この decorator は、連想配列へのアクセスを、 `Hashtable` へのアクセス式に展開する。
- 連想配列のために定義された TypeNameChecker と TypeChecker のクラスのインスタンスが有効になるように、テーブルに登録。

8 型チェックパスの設計理由

8.1 なぜ visitor pattern を使わないか？

visitor pattern は、拡張性の高いコンパイラを記述するための design pattern である。コンパイラを visitor pattern に従って実装しておけば、抽象構文木を traverse する処理が、あとから追加可能になる。

しかし、 visitor pattern には、新しい機能は追加しやすいが、新しいノードの追加はしにくい、という欠点がある。すなわち、言語仕様はすでに確定しているが、最適化などの機能は今後追加される可能性があるようなコンパイラの記述には向いているが、EPP のように、新たな構文が追加されうるシステムには向いていない。

8.2 なぜ型の意味を Type オブジェクトのメソッドにしないのか？

型 t1 が型 t2 の super type かどうか、など、2 つの型の間の関係は、オブジェクト指向的には書きにくい。

オブジェクト指向的に記述する 1 つ方法として、double dispatch （あるいは、もし記述言語がサポートしていれば、マルチメソッド）を使う方法がある。この方法を使えば、処理対象の言語 (source language) の型の階層構造を、記述言語 (description language) のクラスの階層に当てはめることにより、簡潔に記述することができるだろう。

しかし、この方法は、「幅広い範囲の言語拡張をサポートする」という EPP の目的には合わない。なぜなら、記述言語の継承構造に当てはめられない型システムを、実現不可能にしてしまうからである。

非オブジェクト指向的記述スタイルの欠点は、処理が 1 つの関数に集中し、モジュラリティ・拡張性が悪いことにあったが、 mixin を使った記述により、この問題は、完全に解決されている。

8.3 なぜ、mixin と decorator pattern が混在しているのか？

このフレームワークでは、拡張モジュールの composability を高くするためのメカニズムとして、 mixin と decorator pattern の両方を使っている。2 つのメカニズムは非常に似ているが、それぞれ一長一短がある。

mixin は、オリジナルのクラスに新たなメソッドを追加することができるが、decorator pattern では、あらかじめ決まった種類のメソッドを拡張することしかできない。したがって、 mixin の方が拡張性が高いと言える。一方、 mixin の組み合わせはクラスの定義時に決まるのに対し、 decorator の組み合わせは動的に変更できるという特徴がある。しかし、 EPP ではプリプロセッサ起動時にシステムの構成が決まり実行中に変化することはないので、この特徴は不要である。

しかし現在の SystemMixin plug-in の実装では、mixin のメソッド呼び出しが通常の Java クラスのメソッド呼び出しに比べて 1 桁程度遅いという問題がある。このオーバーヘッドを避けるため、現在のところパーザ以外では mixin は用いていない。なお、パーザにおいては、 decorator pattern を使った方がむしろオーバーヘッドが大きくなる。パーザは 100 以上もの mixin から構成されている。これを decorator pattern で実装すると、1 つのメソッド呼び出しのたびに 100 回以上の delegation が起きることになる。

なお、 SystemMixin plug-in のメソッド呼び出しの実行速度は将来改善する予定である。効率の問題が解決されれば、 decorator pattern は使わずに mixin だけを使うことにより、アーキテクチャをよりシンプルにすると同時に拡張性もより高くすることができるだろう。

9 分割コンパイル

9.1 Java コンパイラの動作

Java 言語の言語仕様上の大きな特徴として、 forward declaration をしなくてもクラス間やファイル間で相互参照することができ、ヘッダーファイルや Makefile を書かなくても、コンパイラが分割コンパイル・自動再コンパイルしてくれる点がある。

オブジェクト指向プログラミングでは、クラス間の複雑な依存関係が本質的に必要な場合が多く、 Java 言語のこの特徴は、ソフトウエアの生産性を向上させる上で重要な役割を果たしている。

Java コンパイラは、次のようにして分割コンパイルを実現している。ヘッダーファイルと Makefile に相当する情報、すなわちクラスの signature と依存関係の情報は、 Java コンパイラがソースコードから自動生成し、 class file の中に書き込んでいる。分割コンパイル・自動再コンパイルは、 class file に書かれた情報を適宜読み込むことによって行なわれる。class file がまだ存在しない場合は、クラス名からそのクラスが存在するはずのソースファイルを推測し、そのファイルを構文解析することで、必要なクラスの signature を確定する。

9.2 EPP の動作

EPP は、 Java コンパイラが行なっている動作をより一般化したメカニズムを持ち、 API として plug-in プログラマに提供している。

EPP のフレームワークでは、ファイルが外部から参照される時に必要となる最低限の情報を、 FileSignature と呼ぶデータ構造で表現する。 FileSignature は、ファイル内に含まれるクラス名と、そのクラスを定義する抽象構文木の骨組みとの対応表である。「クラス定義の骨組み」とは、そのクラス定義の抽象構文木から、メソッド定義の本体を省いたものである。クラスの詳しい型情報 (ClassInfo) は、この FileSignature の情報を参照することによって、構築するこ

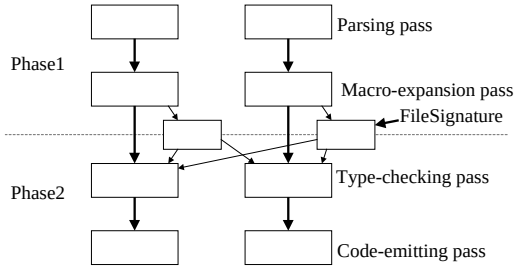

図 10: 変換の 2 つの phase

とができる。

具体的には、 EPP による変換処理は、以下のように進む。EPP の 1 つのファイルに対する処理は、phase1 と phase2 に分かれる。phase1 は、構文解析パスと、型情報に依存しないマクロの展開パスから成る。phase1 は他のファイルに一切依存せずに行なわれる。そして、 phase1 が終った時点で、そのファイルの FileSignature が確定し、FileSignature は、ファイルに保存される。phase2 は、型チェックパスおよびコード出力パスから成る。phase2 では、他のファイルの FileSignature を必要に応じて参照しながら、型チェックを行なったり、型情報を利用したマクロ展開を行なったりする。複数のファイルを EPP によって同時に変換する場合は、まずすべてのファイルの phase1 を行ない、すべての FileSignature が確定した後、すべてのファイルの phase2 にとりかかる。

plug-in は、新たな型を導入するために、対応表 (FileSignature) のエントリに、独自のデータ構造を入れることができる。これにより、 parameterized class のような、「拡張されたクラス」を実装することができる。また、クラスに限らず、例えばユーザ定義のマクロなど、外部から参照可能なあらゆる情報を、 FileSignature のエントリに入れることができる。

ただし、plug-in の実装者に対して、次のような重要な要請がある:

「すべての型は、 FileSignature のみを用いて、確定することができなければならない。」

この要請は、plug-in ができることに制約を加えてしまう。例えば phase2 での変換を行なわないと型が確定しないような言語拡張は、本フレームワークの分割コンパイルメカニズムのサポートの範囲外であることを意味する。しかし、逆に plug-in はこの要請さえ満たせば、 EPP のフレームワークが提供する分割コンパイルのサポートの恩恵にあずかることができる。

9.3 FileSignature を使った plug-in の実装例

本フレームワークを利用した例として、多重継承 plug-in の実装のアウトラインを説明する。この plug-in は、２５９行で実装されている。

- 文法拡張。
  実は、 EPP のパーザは最初から extends のうしろに複数の型名があっても受け付けるので、何もしなくてよい。
- クラス定義の抽象構文木から FileSignature への変換メソッドの拡張。
  同様の理由で、実は何もしなくてよい。
- FileSignature から ClassInfo への変換メソッドの拡張。
  多重継承を行なっているクラスの型を、本物のスーパークラスとにせもののスーパークラスの FileSignature から決定する。
- class を表すノードの TypeChecker の decorator による拡張。
  多重継承を行なっているクラスの抽象構文木を、メソッドをにせもののスーパークラスに delegate する、単一継承のクラスに変換する。

このように実装するだけで、4章で述べた問題をすべて解決する多重継承 plug-in が実現できる。

10 関連研究

文法やデータ型を拡張できる拡張可能言語は従来から数多くあるが、 EPP のように広範囲な型システムの拡張ができる上、相互参照と分割コンパイルの問題を解決した拡張可能言語システムは他にない。

言語処理系のモジュール化が行なえるコンパイラ生成システムとして、Eli[7] がある。Eli は、属性文法に基づいた文法と意味の定義から、言語処理系を自動生成する。既存の定義部品から一種の継承を行なうことによって、新たな言語を定義することができる。一般に属性文法に基づいたシステムの場合、 ad hoc な処理や複雑な記述のデバッグ方法などまで含めると、習得にはかなりの時間が必要になると思われる。EPP のようにフレームワークに基づく言語拡張手法では、記述言語にコモディティの手続き型言語の言語処理系を用いるため、複雑な処理の記述やデバッグに関しては、プログラマーのスキルをそのまま生かすことができる。また属性文法のような宣言的記述の利点の 1 つとして再利用性の高さがあるが、EPP では collaboration-based design を用いることにより、宣言的記述と同等の高い再利用性を達成していると筆者は考えている。

コンパイル時 MOP(Meta Object Protocol)[13] を提供することによって言語を拡張可能にするシステムに、MPC++[12], OpenC++[4], JTRANS[14], OpenJava[22] がある。これらのシステムは、 EPP と同様に、構文解析後の抽象構文木に対して複雑な変換処理を行なうことを目的としている。文法も限られた範囲の中で拡張可能になっている。例えば MPC++ では、新しい演算子や文などを追加できるようになっている。また、 MPC++, OpenC++, OpenJava システムでは、メタクラスを定義することでクラスの機能を拡張したり、型情報を利用したマクロ展開が可能である。しかし、クラスという枠組を越えたデータ型の追加などは行なえない。

CLOS はすべてのデータ型をオブジェクトとして扱うと同時に実行時 MOP[13] を提供することによって、極めて強力な言語拡張 (特にデータ型に関して) を可能にする言語である。しかし一般に実行時 MOP はそれを解釈する runtime system の存在を必要と

するため、言語拡張の範囲もその runtime system でできる範囲内に自ずと制限される。もちろん EPP による言語拡張の範囲も、出力言語である Java 言語の機能によって制限される。しかし EPP のアーキテクチャをほぼそのまま用いて、C 言語を出力言語とするトランスレータを作ることが可能である。その場合は runtime system や出力言語による制約はほとんどなくなり、完全に自由な言語拡張が可能なシステムになる。筆者は現在そのようなトランスレータの実装を計画している。

EPP と同様に、部品を追加することで文法の拡張が可能なプリプロセッサとしては、Camlp4[17] という Objective Caml のプリプロセッサがある。しかし型システムを拡張する機能は持っていない。

11 まとめ

拡張可能 Java プリプロセッサ EPP が有する、型チェック機構フレームワークについて述べた。このフレームワークは、FileSignature という概念を導入することで、型情報を利用した変換処理に伴い生じるクラス間・ファイル間の相互参照の問題や分割コンパイルの問題を解決している。

言語の拡張部品である EPP plug-in は、system mixin と decorator pattern の 2 つの機構を用いて、collaboration-based design に基づいて実装される。これにより高い composability が達成される。

FileSignature を使った型チェックのフレームワークは実装ずみであり、本論文で述べた plug-in の例は、すべて動作している。すべてのソースコードは、www によって配布中である [11]。

本論文で述べた分割コンパイルおよび自動再コンパイルのメカニズムはまだ実装されていないが、今後実装を進めて行く予定である。また、plug-in の composability が実際にどの程度高いのか、何が plug-in どうしの衝突の原因になりやすいかを調べることも今後の課題である。

本フレームワークを用いて第三者が実用的な plug-in を実装可能かどうかを調べる実験として、現在 GJ[3] と完全にコンパチブルなパラメタ付きクラス plug-in の実装を行なっている。実装は、最初に plug-in の骨組み (700 行程度) をまず筆者が書き、残りの実装を第三者に依頼するという形で行なった。現在のところソースコードは 1600 行程度になり、GJ のほとんどの機能の実装がすでに終っている。

また、本フレームワークを用いたプロジェクトとして、JavaMetrics プロジェクト [23] がある。このプロジェクトでは Java プログラムのソースコードの性質を１００項目以上の様々な観点から数値的に計測するメトリックスツールと、テストプログラムによるテスト終了率を計測するテストカバレージツールの実装を現在行なっている。EPP が提供する構文解析・型チェック・抽象構文木操作の機能を利用することにより、高い生産性を上げている。

参考文献

[1] Aho, A.V., Sethi, R. and Ullmann, J.D.: "Compilers: Principles, Techniques and Tools.", Addison-Wesley Publishing company, 1987.

[2] Bracha, G. and Cook, W.: "Mixin-based Inheritance", In Proc. of ECOOP/OOPSLA'90, pp.303–311, 1990.

[3] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P.: GJ, http://www.cis.unisa.edu.au/~pizza/gj/

[4] Chiba, S.: "A Metaobject Protocol for C++", In Proc. of OOPSLA'95, pp.285–299, 1995. http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: "Design Patterns", Addison Welsley, 1995.

[6] Gosling, J., Joy, B. and Steele. G.: "The Java Language Specification.", Java Series, Sun Microsystems, 1996.

[7] Gray,R.W., Heuring,V.P., Levi,S.P., Sloane,A.M. and Waite,W.M.: "Eli: A Complete, Flexible Compiler Construction System", Communications of the ACM 35 (February 1992), pp.121–131.

[8] Harrison, W. and Ossher, H.: "Subject-Oriented Programming(A Critique of Pure Objects)." In Proc. of OOPSLA'93, pp.411–428, 1993.

[9] Ichisugi, Y. and Yves Roudier: "The Extensible Java Preprocessor Kit and a Tiny Data-Parallel Java", In Proc. of ISCOPE'97, LNCS 1343, pp153–160, California, Dec, 1997.

[10] 一杉裕志: "高いモジュラリティと拡張性を持つ構文解析器", 情報処理学会論文誌：プログラミング, Vol.39 No.SIG 1(PRO 1), Dec., 1998.

[11] Ichisugi, Y.: EPP, http://www.etl.go.jp/~epp/

[12] Ishikawa, Y.: "Meta-level Architecture for Extendable C++ Draft Document", Technical Report TR-94024, RWCP, 1994. http://www.rwcp.or.jp/lab/mpslab/mpc++/mpc++.html

[13] Kiczales, G., des Rivieres, J. and Bobrow, D. G.: "The Art of Metaobject Protocol", MIT Press, 1991.

[14] Kumeta, A. and Komuro, M.: "Meta-Programming Framework for Java", The 12th workshop of object oriented computing WOOC'96, Japan Society of Software Science and Technology, March, 1997.

[15] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.M. and Irwin, J.: "Aspect-Oriented Programming." Invited Talk. In Proc. of ECOOP'97, LNCS 1241, pp.220-242, 1997.

[16] Mezini, M. and Lieberherr, K.: "Adaptive Plug-and-Play Components for Evolutionary Software Development", In Proc. of OOPSLA'98, pp.97–116, Oct. 1998.

[17] Rauglaudre, D.: Camlp4, http://pauillac.inria.fr/camlp4/

[18] Steele, G.L.: "Common Lisp the Language 2nd edition.", Digital Press, 1990.

[19] Smaragdakis, Y. and Batory, D.: "Implementing Layered Designs with Mixin Layers", In Proc. of ECOOP'98, pp.550–570, LNCS 1445, 1998.

[20] Szyperski, C. A.: "Import is not Inheritance – Why We Need Both: Modules and Classes.", In Proc. of ECOOP'92, LNCS 615, pp.19–32, 1992.

[21] VanHilst, M. and Notkin, D.: "Using Role Components to Implement Collaboration-Based Designs", In Proc. of OOPSLA'96, pp.359–369, Oct. 1996.

[22] 立堀，千葉: "ユーザにとって直観的なコンパイル時 MOP" SPA'98 , March, 1998. http://www.softlab.is.tsukuba.ac.jp/~mich/openjava

[23] 塚本：JavaMetrics プロジェクト http://www.etl.go.jp/etl/bunsan/metrix/