Extensible Type System Framework for a Java Pre-Processor : EPP (DRAFT)

Yuuji ICHISUGI
ichisugi@etl.go.jp
Electrotechnical Laboratory, PRESTO JST

Abstract

The author is developing an extensible Java pre-processor, EPP. EPP is the framework for a Java source code processing system which can be extended by adding extension modules, *EPP plug-ins*. EPP has an extensible recursive descent parser and an extensible type checking system. A great variety of new language features can be easily introduced by implementing macro expansion functions for new language constructs. Because implementation of extensions are based on the mixin mechanism, extension modules render high composability. Operator overloading and multiple inheritance were implemented as pre-processor plug-ins. We are also planning to implement a parameterized class feature that is completely compatible with the GJ (Generic Java) system. One problem of implementing extended Java languages is separate compilation. This framework has solved the problem by introducing the notion of *FileSignature*, which can be applied to a wide range of extended languages.

1 Introduction: Making a Programming Language Like a PC Compatible Machine

Modern programming languages are a huge monolithic system. Recently in particular, the language features necessary for practical programming languages have been increasing and programming language specifications are becoming more and more complex. This has caused the following problems:

1. It has become very difficult for language researchers to experiment on new language features.

Examining whether a new language feature works effectively with real applications requires the implementation of an experimental language processor having the necessary language features. However, now that the number of necessary language features has increased, implementation is extremely difficult. One alternative means is to add a new language feature by altering part of the language processor whose source code is available. Although, this is quite difficult because of the gigantic, monolithic language processors.

2. Although the programmer wants some language features to be implemented, they are hardly taken into account by any of the present language processors.

   All language features are provided to programmers as tie-ins: programmers are hardly free to choose the features desired. Sometimes features selected by a language designer to suit his own tastes are imposed on the programmer. Thus, it is almost impossible for the latest results in the field of language study to be adopted to the language processor at hand, or it takes years to do so. This is due to the fact that the gigantic, monolithic programming language specifications make it extremely difficult to add new language features.

In order to solve the problems described above, the author has been trying to modularize programming language processors with the aim of having programming languages make the rapid progress as if personal computers have experienced. Personal computer hardware has been making great strides in performance over the past decade, triggered by the advent of the technical standard of its internal architecture. As a result, many manufacturers launched into the production of modules. Free-market competition encouraged the manufacture of higher-performance, diversified modules. The author aims at the same for programming languages: to facilitate the development of language features as a unit of a module by modularizing programming language processors and setting a standard interface for connecting modules.

The author has already proposed a method for constructing a modularized recursive descent parser using mixins and has confirmed the effectiveness of the method on the extensible pre-processor, EPP[9].

This paper describes the modularized, extensible type-checking mechanism newly implemented for the EPP, which is the next step for modularization of programming language processors.

The organization of this paper is as follows. Section 2 outlines EPP and Section 3 explains sample plug-ins of EPP which extends Java's type system. Section 4 mentions problems for designing an extensible type system framework. Section 5 explains plug-ins necessary in this framework implementation. Section 6 outlines the principle of extensible recursive descent parser. Sections 7, 8, 9 describes the design of extensible type system of this framework. Section 10 describes related studies, and Section 11 is the conclusion.

2 Outline of the EPP Extensible Pre-Processor

EPP[8, 9, 10] is an extensible Java pre-processor that can introduce new language features. The user can specify EPP plug-ins at the top of the Java source code by writing "#epp *plug-in-name*" in order to incorporate various extensions of Java. Multiple plug-ins can be incorporated simultaneously as long as they do not collide with each other. Emitted source codes can be compiled by ordinary Java compilers and debugged by ordinary Java debuggers.

EPP works not only as a Java pre-processor but also as a language-experimenting tool for language experts, a framework for extended Java implementation, and a framework for a Java source code analyzer/translator.

The EPP's source code is written in Java extended by the EPP itself. It was bootstrapped by the EPP written in Common Lisp[15]. The compiled byte-code of EPP is available under any platform where Java is supported.

In designing EPP, the author aimed at a wide-range of extensibility and simultaneous usability of multiple extension features–hereafter called composability. Because the EPP's architecture is independent of Java, it is applicable to the framework for the processing system of other languages.

3 EPP Plug-In Examples

3.1 Association Array Plug-In

Figure 1 shows a program using a plug-in implementing association arrays. An association array is a data structure that languages such as perl have. Although it is in fact a hashtable, a programmer can handle it with the same interface as that for an array. This plug-in has adopted specifications that allow type-safe assignment in accordance with the contra-variant rule. Access to an association array is translated into an expression to access the hashtable by the pre-processor.

3.2 Multiple Inheritance Plug-In

Figure 2 shows a program using a plug-in implementing multiple inheritance. The multiple inheritance class is translated into the class having extra pseudo super classes as instance variables. Also, methods that delegate messages to methods defined at the pseudo super classes are added. (Of course, such simple implementation does not

```
#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;
  }
}
```

Figure 1: A program using association array plug-in.

realize multiple inheritance in the true sense of the word, because it will provide an incorrect value of "`this`" and it does not allow the `C` instance to be used as a `C2`-type value.)

4 Problems with the Pre-processor Extending the Type System

Java allows mutual dependence between classes, a problem which a Java compiler implementor has to be careful about. In the case of Java, however, this problem is not serious, because Java has fixed language specifications and the compiler can determine class signatures simply by parsing.

EPP has generalized the mechanism for solving the interdependence between classes provided by the Java compiler. The generalized mechanism is provided as the Application Program Interface (API) for plug-in programmers. Designing a framework in which such extension features as multiple inheritance plug-ins can be implemented requires especially careful consideration. For instance, the following problems are encountered:

```
#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 {
}
```

Figure 2: A program using multiple inheritance plug-in.

- Suppose the program translation depends on the signature of another class, and at the same time the translation modifies the signature of the class, what kind of procedure should the translation use? For example, in the case of multiple inheritance plug-ins, in order to add methods to delegate, it is necessary to know what methods pseudo super classes has. Meanwhile, in the class that allows multiple inheritance, methods to delegate are added and the signature is modified. If this class becomes a pseudo super class of another class, can the program be translated correctly?

- Separate compilation makes the problem more complicated. Suppose file F_a has classes A_1 and A_2, and file F_b has classes B_1 and B_2 (Figure 3). If the translation of A_1 depends on B_1 and that of B_2 depends on A_2, can these two files be translated correctly? Also, when the content of file F_a is updated, can F_b, as well as F_a be re-compiled automatically?

- When a dependent relationship contains loops, can the system detect them without lapsing into endless loops? For example, when class C itself is specified as a pseudo super class of class C, can it be reported as an error?

Of course, technically, all the problems raised above are not difficult to solve.

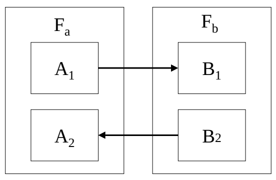

Figure 3: Interdependence of files.

Rather, significant aims in designing a framework are the followings:

1. The framework should handle the demanding problems and lighten as many burdens as possible imposed on framework users (plug-in implementors).
2. The framework should give plug-in implementors language extensibility over the widest possible range and remove unnecessary restrictions on what they want to do.

The author has not evaluated thoroughly as of yet to what extent this paper's framework has achieved the aims stated above; thorough evaluation requires the implementation of a large number of applications. However, it seems that the framework has been working well at this stage.

5 Plug-Ins Necessary for the EPP's Description

The source codes of EPP and EPP plug-ins are written in Java extended by the EPP itself. This section describes the functions of Symbol plug-in, SystemMixin plug-in and BackQuote plug-in, which are necessary for the EPP's description.

5.1 Symbol plug-in

Symbol, a data type that languages like lisp have, features higher-speed comparison of identity than strings. Figure 4 shows a Java program using the EPP's Symbol plug-in. A literal of a Symbol is represented as a colon followed by an identifier or a string literal.

In the EPP's source code, Symbol is used for various purposes such as tags of types or abstract syntax tree nodes to identify their kinds.

```
#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
  }
}
```

Figure 4: A program using Symbol plug-in.

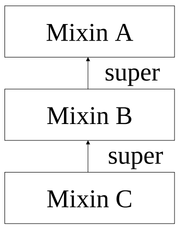

Figure 5: Mixins composing a class.

5.2 SystemMixin Plug-In

5.2.1 What is a Mixin?

SystemMixin plug-in implements its own object-oriented language on top of Java language. The language supports the special mechanism called mixins[2], that can be used in language systems such as Flavors and CLOS[15].

Mixins are re-usable fragments of classes. Usually, in an object-oriented language, a particular super class name is specified when defining a subclass; a mixin is a subclass defined with no particular super class specified. The mixin's super class is determined when it is multiply inherited and linearized by another class afterward(Figure5).

Bracha[2] showed that the mixin mechanism can simulate the same inheritance mechanisms as SmallTalk, BETA and CLOS. VanHilst[17] suggested a variation of mixin-based inheritance which enhances reusability of object-oriented programs. EPP has achieved a high extensibility and composability by modularizing the entirety using

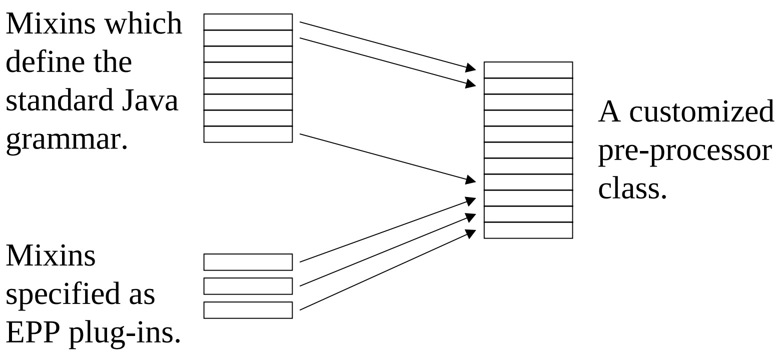
Mixins which define the standard Java grammar.
Mixins specified as EPP plug-ins.
A customized pre-processor class.

Figure 6: Mixins composing EPP.

the mixin mechanism.

5.2.2 Mixins Composing EPP

The EPP's main body is defined as a class named `Epp`, with the class definition divided into multiple mixins. Starting EPP combines all the mixins composing the standard Java parser and mixins composing the plug-in specified at the top of the source code to construct a customized version of the class `Epp`(Figure 6). The main routine of EPP then generates the class instance of `Epp` and invokes the starting method to begin pre-processing the source code.

5.3 BackQuote Plug-In

EPP translates the input program by manipulating the AST (abstract syntax tree) as a result of parsing.

A BackQuote plug-in implements the same mechanism as the backquote macro of lisp in that it enables the AST to be embedded into the source code in describing the translation process.

Inside EPP, the AST is represented as having a structure similar to the S-expression of lisp. For example, the internal representation of the AST for a Java expression: `(a + b)` is the following:

```
        (+ (id a) (id b))
```

To represent this in the plug-in source code using Java, The plug-in programmer should write the following code:

```
Tree t = new Tree(:"+",
                  new Identifier(:a),
                  new Identifier(:b));
```

Clearly, this is rather complicated. With the backquote macro, it is much easier to write AST-generating expressions. For example, the backquote macro makes the above expression more concise as follows:

```
Tree t = `(+ (id a) (id b));
```

Also, like the lisp backquote macro, it is possible to embed Java expressions into the AST arbitrarily with "`,`" and "`,@`". For example,

```
Tree t = new Tree(:"+",
                  new Identifier(:x),
                  exp);
```

This can be rewritten as follows:

```
Tree t = `(+ (id x) ,exp);
```

6 Extensible Recursive Descent Parser

The principle of the extensible recursive descent parser[1] has been described in another paper[9]. The following outlines this principle.

Figure 7 shows a framework for a method of parsing the non-terminal `exp`. This program itself does nothing but invoke the method `term`. However, it has hooks for the grammar extension named `expTop`, `expRight` and `expLeft`. It is possible to add alternatives to the non-terminal by adding mixins to extend these methods.

Figure 8 shows a method which add a left associative binary operator alternative. Adding mixins with the method laid out in Fig. 8 to mixins with the method laid out

```
Tree exp(){
  Tree tree = expTop();
  while (true){
    Tree newTree = expLeft(tree);
    if (newTree == null) break;
    tree = newTree;
  }
  return tree;
}
Tree expTop(){ return expRight(exp1()); }
Tree expRight(Tree tree){ return tree; }
Tree expLeft(Tree tree){ return null; }
Tree exp1(){ return term(); }
```

Figure 7: A framework for an extensible non-terminal parser.

```
Tree expLeft(Tree tree) {
  if (lookahead() == :"+") {
    matchAny();
    return new Tree(:"+", tree, exp1());
  } else {
    return original(tree);
  }
}
```

Figure 8: An extension of a left associative binary operator.

in Fig. 7 extends the behavior of the parser. (The method-invocation expression, "`original`", introduced by SystemMixin plug-in, corresponds to the super method-invocation in traditional object-oriented languages.)

The EPP's parser is implemented in the style described above. Some parts, however, are implemented by means of ad-hoc techniques such as back-tracking because the Java grammar is not LL(1).

7 Extensible Type System

7.1 Outline of the Type-Checking Pass

In the type-checking pass, type information is added to each node of the AST. An AST node, an immutable object, is translated into a node with type information by a *TypeChecker* object. A type-checking pass is described in the functional programming style rather than in the object-oriented paradigm. The reason for this design is described in Section 8.

7.2 Type Object

A class `Type` is a data type for expressing types within EPP. All types have a *tag*, with a name representing the kind of type. For example, class type has a tag named `:class`.

Plug-ins can introduce new data types by defining the subclass of the class `Type` with a new tag. A Type object only preserves information necessary to express a type; it does not define "the meaning of a type" at all. The meaning of a type is defined by methods of the class `Epp` and TypeChecker objects.

Each Type object with the `:class` tag has another object named *ClassInfo* which has detailed information of the class. The ClassInfo gives information such as what kind of methods the class has. The ClassInfo is generated by the Type object lazily when necessary. This lazy generation mechanism enables the handling of interdependent classes. If the content of a class is accessed from outside in the midst of the lazy generation of the content, it is reported as an error on the basis of circular dependence with intrinsic contradiction. For example, such an error is reported if the user specified a class for the super class of itself.

If the user accesses ClassInfo of a particular class C for the first time when compiling file F, the information saying, "F depends on a file where C is defined" is automatically recorded. This information is used for automatic re-compilation of the file that depends on the updated file.

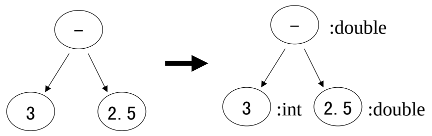

Figure 9: Adding type information to the AST.

7.3 TypeChecker Object

TypeChecker objects define how control structures and operators handle types. A TypeChecker has a method named `call`, which translates a tree without type information into a tree with type information(Figure 9).

A TypeChecker is registered in a hashtable which associates a tag of a node with the TypeChecker. The type-checking pass recursively invokes `call` of TypeChecker objects associated to each node of AST from top to down.

Figure 10 shows a TypeChecker associated with the binary operator for four basic mathematical operations. The method `coerce` checks if the second argument tree can be used as an expression with the first argument type. If it is not, a type error is reported with a user-friendly error message.

A TypeChecker can be extended by decorator pattern[5]. For example, Figure 11 shows a decorator handling the special semantics for strings in the Java binary operator "+". The decorator regards the type of the operation result as String type if either the right or left operand is String type; and in the other cases it invokes the original TypeChecker. The decorator is added only to the TypeChecker object associated with the node having the tag "+".

Adding more decorators to the TypeChecker object of a binary operator enables implementation of operator overloading. Figure 12 shows a decorator to add to the binary operator "-". If the left operand type is a class type, the expression is translated to the following method invocation expression:

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

```
// +, -, *, /
class DoubleOperatorTypeChecker extends TypeChecker {
  public Tree call(Tree tree) {
    checkArgsLength(tree, 2);
    Tree[] newArgs = typeCheckArgs(tree);
    Symbol s1 = newArgs[0].type().tag();
    Symbol s2 = newArgs[1].type().tag();
    Type t;
    if (s1 == :double || s2 == :double){
      newArgs[0] = Type.coerce(Type.Tdouble, newArgs[0]);
      newArgs[1] = Type.coerce(Type.Tdouble, newArgs[1]);
      t = Type.Tdouble;
    } else if (s1 == :float || s2 == :float){
      newArgs[0] = Type.coerce(Type.Tfloat, newArgs[0]);
      newArgs[1] = Type.coerce(Type.Tfloat, newArgs[1]);
      t = Type.Tfloat;
    } else if (s1 == :long || s2 == :long){
      newArgs[0] = Type.coerce(Type.Tlong, newArgs[0]);
      newArgs[1] = Type.coerce(Type.Tlong, newArgs[1]);
      t = Type.Tlong;
    } else {
      newArgs[0] = Type.coerce(Type.Tint, newArgs[0]);
      newArgs[1] = Type.coerce(Type.Tint, newArgs[1]);
      t = Type.Tint;
    }
    return tree.modifyTypeAndArgs(t, newArgs);
  }
}
```

Figure 10: A TypeChecker for binary operators.

```
class StringPlusOperatorTypeChecker extends TypeChecker {
  public Tree call(Tree tree) {
    checkArgsLength(tree, 2);
    Tree[] newArgs = typeCheckArgs(tree);
    Type t1 = newArgs[0].type();
    Type t2 = newArgs[1].type();
    if ((t1.tag() == :class &&
         t1.classInfo().getName() == :"java.lang.String")
        ||
        (t2.tag() == :class &&
         t2.classInfo().getName() == :"java.lang.String")){
      return tree.modifyTypeAndArgs(Type.TString,
                                    newArgs);
    } else {
      return orig.call(tree.modifyArgs(newArgs));
    }
  }
}
```

Figure 11: A TypeChecker decorator.

```
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));
    }
  }
}
```

Figure 12: The decorator which implements operator overloading.

```
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);
  }
}
```

Figure 13: Extension of the method isSuperType by mixins.

Because a decorator pattern is used here, the system works correctly even if some other plug-in extends the same TypeChecker simultaneously. (The reason why mixin mechanism is not used here is discussed in Section 8.)

7.4 TypeNameChecker Object

A *TypeNameChecker* object determines what kind of type an AST expressing a type name (like "`String`") or a type constructor (like "`String[]`") actually represents. TypeNameChecker, like TypeChecker, has a method named `call` which translate a tree without type information to a tree with type information.

Plug-ins can define new TypeNameCheckers in order to introduce new type constructors that do not exist in standard Java.

7.5 Relationship Between Types

Methods for defining the relationship between two types are defined as extensible methods of the class `Epp`. Plug-ins can modify the relationship between types by adding mixins which extend these methods.

Figure 13 shows an extension of a method in the source code of an association array plug-in. The method `isSuperType` determines whether the first argument type is a super type of the second argument type. In the method, the relationship between association arrays is defined in accordance with the contra-variant rule.

7.6 Introduction of New Types

The extension mechanisms described in Section 7 offer extremely wide-ranging language extension with high composability.

This section outlines the association array implementation described in Section 3. This plug-in is implemented in 210 lines.

- The definition of grammar for association arrays.
- The extension of methods such as `isSuperType`, which define the relationship between types.
- The definition of `AssociationArrayType`, a subclass of the class `Type`, representing an association array type.
- The definition of a TypeNameChecker that defines the meaning of the AST which expresses type constructors of association array types. This TypeNameChecker is implemented such that an association array type is handled as an AssociationArrayType type internally, while it is translated to a `Hashtable` in the output source code.
- The definition of decorators that extend TypeCheckers of array access expression and assignment statement. These decorators translate access to an association array into an access expression to a `Hashtable`.

8 Reason for Designing Type-Checking Pass

8.1 Why is visitor pattern not used?

Visitor pattern[5] is a kind of design pattern for describing a highly extensible compiler. If a compiler is implemented in accordance with a visitor pattern, it is possible to add the traversing process of an AST afterward.

A visitor pattern, however, has a drawback: there is the difficulty of adding new nodes. That is, a visitor pattern is suitable for the description of a compiler which has fixed language specifications and the possibility that some features, such as optimization, will be added to it in the near future. It is unsuitable, on the other hand, for such systems as the EPP to which new syntax can be added.

8.2 Why meaning of type is not defined as Type object's methods?

Using the object-oriented language style, it is difficult to describe the relationship between two types, e.g. whether type t_1 is the super type of type t_2.

One way of sticking to the object-oriented description is to use double dispatch (or the multi-method, if supported by the description language). Applying the hierarchical structure of the source language type to the hierarchy of the description language class would simplify the implementation. However, this way conflicts with the EPP's aim of "supporting wide-ranging language extension" in that it never realizes the type system inapplicable to the hierarchical structure of classes of the description language.

The drawback of the non object-oriented description style was that all the processes proceeded through some particular functions, resulting in low modularity/extensibility. The mixin-based description has completely solved this problem by defining each function split into several mixins.

8.3 Why are mixins and decorator patterns both used?

In order to achieve high composability of extension modules, both the mixin and decorator pattern mechanisms are utilized in this framework. Although they are very much alike, each of them has its own advantages and disadvantages.

The problem of current implementation of mixin mechanism is the method invocation overhead of mixins. In order not to generate such overhead, the framework is designed such that the minimum number of mixins is used only where necessary.

On the contrary, the parser will generate greater overhead with the decorator pattern because the parser consists of more than 100 mixins. Its implementation with the decorator pattern would generate a large number of method delegations.

Intrinsically, defining every class in the system as mixins renders the highest extensibility. The author intends to improve the method invocation overhead of the SystemMixin plug-in in the near future.

9 Separate Compilation

9.1 How the Java Compiler Operates

Java allows interdependence between classes or files without any forward declaration. The Java's compiler executes separate compilation and automatic re-compilation without header files or a makefile. This is a remarkable feature of the Java language. This feature play a significant role in improving software productivity, because

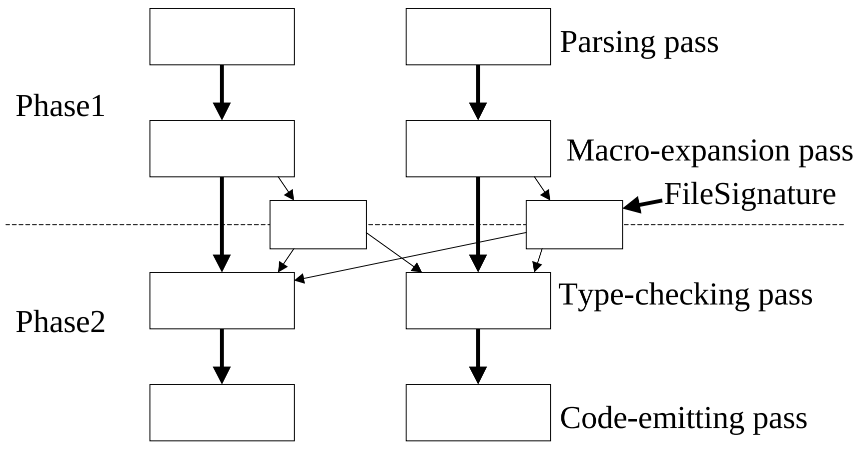

Figure 14: Two phases of the translation.

in many cases object-oriented languages intrinsically require complicated dependent relationship between classes.

As for the header files and a makefile, we can understand that they are in fact automatically generated out of the source code by the Java compiler and written in the class file. When a class file does not exist, it is possible to determine the class signature by parsing the class definition.

9.2 How EPP Operates

EPP incorporates more generalized operations than those of the Java compilers and provides them to plug-in programmers as an API.

Within the EPP framework, the data structure called *FileSignature* is the minimum information needed when a file is externally referenced. FileSignature is a table corresponding to the class name included in the file and the digest form of the AST defining the class. Detailed class type information (ClassInfo) can be constructed using FileSignatures.

More specifically, the translating process of one file by the EPP proceeds as follows:

It divides the process into two procedures called Phase1 and Phase2(Figure 14). Phase1 consists of the parsing pass and the macro-expansion pass that does not depend on the type information. Phase1 is executed independent of all other files. When Phase1 has finished, FileSignature of the file is determined. The FileSignature is saved in a file. Phase2 consists of the type-checking pass and the code-emitting pass. In Phase2, the EPP executes a type-checking pass–while referencing the FileSignature in other files if required–and macro expansion utilizing type information.

When EPP translates multiple files simultaneously, the EPP executes Phase1 for all files first to determine all FileSignatures, and then begin executing Phase2 for all files.

In order to introduce new types, EPP plug-ins can write their original data structures in the FileSignature. This enables the implementation of extended classes such as a parameterized class. Also, in addition to classes, any kind of information that can be referenced externally, such as a user-defined macro, can be included in the FileSignature entries.

Note that the EPP has an important requirement for plug-ins:

- *All types should be determined using only FileSignatures.*

This is a strong demand that greatly restricts the capabilities of plug-ins. For example, a language extension in which the types cannot be determined without results of Phase2 translation is not supported by the EPP framework. However, as long as its plug-ins meet the requirement, they can gain support of the separate compilation provided by the EPP framework.

9.3 An Example of Plug-In Implementation Using FileSignature

The following is an outline of multiple inheritance plug-in implementation. This plug-in is implemented in 259 lines.

- The grammar extension. Actually, no grammar extension is needed because the EPP's parser accepts multiple class names after the `extends` keyword.

- The extension of the translating method from a class-defining AST into the FileSignature entry. Actually, no such extension is needed for the same reason as stated above.

- The extension of the translating method from FileSignature into ClassInfo. This method determines the signature of the class doing multiple inheritance from the FileSignatures of the true super class and pseudo super classes.

- The definition of a decorator which extends the TypeChecker of the node representing the class-defining AST. The decorator translates the AST of the class doing multiple inheritance into a single inheritance class that delegates methods to pseudo super classes.

This implementation process renders multiple inheritance plug-in that solves all the problems discussed in Section 4.

10 Related Work

Eli[7] is a compiler generator which modularizes grammar and semantic definitions. It automatically generates a language processor using grammar and semantics definitions based on attribute grammar, and defines a new language by a kind of inheritance using existing definition modules.

MPC++[11], OpenC++[4], JTRANS[13] and OpenJava[16] are extensible systems which can introduce new language features by providing MOP (Meta Object Protocol[12]) during compilation. Like EPP, their task is to perform translation on an AST after parsing. Also, the grammar is extensible in a limited range. For example, MPC++ allows addition of new operators and statements. Also, MPC++, OpenC++ and OpenJava allow the utilization of type information during macro expansion as EPP does.

Traditionally, there have been many "extensible languages" around that allow grammar modification. For example, Camlp4[14], Objective Campl pre-processor, is an pre-processor in that its grammar can be extended by adding modules.

Nevertheless, EPP is the only system that allows a wide-ranging extension of the type system.

11 Conclusion

The Java pre-processor framework, EPP, which allows type system extension, has been described.

The type-checking framework using FileSignature has been implemented already and all the plug-ins described in this paper have been working well.

The separate compilation and automatic re-compilation mechanisms described in this paper have not been implemented yet; they will be implemented in the near future. Also, large-scale and practical plug-ins such as GJ (Generic Java)[3] compatible plug-ins will be implemented.

The source code of EPP and sample plug-ins are distributed on the EPP web page[10].

References

[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 OOSPLA'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] 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.

[9] Ichisugi, Y.: "Modular and Extensible Parser Implementation using Mixins", Transaction of SIG PRO, Information Processing Society of Japan, December, 1998. (In Japanese)

[10] Ichisugi, Y.: EPP,
http://www.etl.go.jp/˜ epp/

[11] 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

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

[13] 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.

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

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

[16] Tatsubori, M.: OpenJava,
http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/

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