# The Extensible Java Preprocessor Kit and a Tiny Data-Parallel Java

Yuuji Ichisugi and Yves Roudier

Electrotechnical Laboratory, Japan

**Abstract.** We describe the extensible Java preprocessor EPP and a data-parallel extension of Java implemented with EPP. EPP can be extended by incorporating EPP plugins. Tiny Data-Parallel Java is an example of EPP plugin. High portability is guaranteed because the translated code and the run-time systems are pure Java code. Applications can be executed in parallel if the VM interpreter supports parallel execution of Java threads. We provide a preliminary performance evaluation of this system.

## 1 Introduction

The Java language [6] has recently become very popular among programmers. Java has socket and thread libraries which can be used on various platforms. Many JAVA VM (virtual machine) interpreters will soon support shared-memory multiprocessing and will then be useful platforms for parallel programming.

However, Java lacks facilities for language extension, which have been adopted by other object-oriented languages. For example, C++ has a macro preprocessor, operator overloading and template facilities. Smalltalk and CLOS[15] have closures and metaclass facilities. These features to extend language constructs and operators supplement the inheritance mechanism which extends data types. Without such extension possibilities, it is difficult to make parallel libraries which can be used easily by application writers.

To enhance Java in this regard, we developed an extensible Java preprocessor kit, EPP [8]. EPP can be used to introduce new language features, possibly associated with new syntax.

In this paper, we describe Tiny Data-Parallel Java, an extension that is implemented as an example application of EPP. It is based on the same language model as Data-Parallel C [7]. The source code of a Tiny Data-Parallel Java application is translated to standard Java code by a translator implemented using EPP. Because the run-time system is written in standard Java, high portability is guaranteed.

Tiny Data-Parallel Java does not have enough language features to support high-performance parallel programs; however, we think it shows convincingly the effectiveness of the combination of the high extensibility of EPP and of the high portability of Java, and their value for parallel programming. Many data-parallel extensions are currently being proposed for Java ([14, 13], for example), but none of these frameworks has been introduced by means of a general extension mechanism.

## 2 An Extensible Preprocessor Kit: EPP

In this section we review existing tools, and introduce and exemplify the Extensible Preprocessor Kit.

### 2.1 Existing tools

To implement new languages or extend existing ones, preprocessors or translators are often used rather than native compilers. Many language extensions and source level optimization tools for C/C++ are implemented as translators to C/C++. Recently, several extensions for the Java language have also been proposed which are implemented as preprocessors. Because of the simplicity of Java, it is also relatively easy to provide extension support libraries.

The merits of this style of implementation are its simplicity and high portability. Instruction-level optimization can be delegated to the compiler of the target language.

Although there are potentially many useful language extension systems, a user must select just one for a projects, because it is impossible to merge several language extensions or eliminate harmful features from the extended system. Systems with a meta-object protocol (MOP) such as CLOS [11], MPC++ [10], OpenC++ [5], EC++ [4] or JTRANS [12] have solved this problem. These systems allow the implementation of language extensions as modules that can be selected by users. However, extensibility of syntax is slightly restricted in these systems.

### 2.2 Description of EPP

The Extensible Java PreProcessor kit, *EPP*, is an application framework for preprocessor language extension systems. EPP is itself a source-to-source preprocessor of Java. The recursive-descent parser[1] of EPP provides many hooks for extensions (described in Section2.4). By using these hooks, the extension programmer can introduce new features, possibly associated with new syntax, without editing the source code of EPP. Because all grammar rules are defined in a modular way, it is also possible to remove some original grammar rules from standard Java.

Once the parsed program has been transformed into a tree, a preprocessor programmer can easily manipulate it from a program. The usefulness of this kind of tool has already been proven by Lisp implementations, and has been adapted to C++ by various systems like Sage++ [2], MPC++ [10] and OpenC++ [5].

EPP enables preprocessor programmers to write an extension as a separate module. We call the extension modules *EPP plugins*. Several plugins can be incorporated into EPP as long as they do not cause conflicts. users of the preprocessor can select any plugin that fits the characteristics of their projects.

Composability of EPP plugins can be realized thanks to a description language, *Ld-2* [9], an object-oriented package implemented in Common Lisp [15]. The inheritance mechanism of object-oriented languages makes it easy to implement extensible applications because all methods of objects can be considered as hooks for extensions. In addition to the traditional inheritance mechanism, Ld-2 provides a novel feature called *system mixin*, which supports extensible and flexible software.

Although the current target of EPP is only Java, the architecture of EPP is applicable to other programming languages.

### 2.3 An Example of an EPP Plugin

The user of EPP can specify one or more *EPP plugin files* at the start of a Java program. Figure 1 is a simple example program using an EPP plugin that defines a swap macro. The plugin file is actually an Ld-2 program file that will be dynamically loaded by EPP before preprocessing.

```
#epp load "swap"
public class test {
  public static void main(String[] argv){
    int a = 1, b = 2;
    swap(int, a, b);
  }
}
```

**Fig. 1.** A Program Using a Swap Plugin

### 2.4 System Mixin

The first program in Figure 2 defines a method named `:m` which contains nested `if` statements. This method is written in a traditional, “monolithic” way. When a programmer wants to add a new `else-if` clause to this method, the only way to do so is to edit the source code. By using the inheritance mechanism, some editing can be avoided, but this changes the identity of the method. It is indeed impossible to extend the behavior of the method without defining a new subclass.

Ld-2 provides a novel mechanism, *system mixin*, which enables the program to be split into small reusable modules. Bracha[3] showed the flexibility and reusability of *mixin-based* inheritance. System mixins are similar to mixins, but the unit of inheritance is not a class but the whole program.

```
  (class <foo> ()
    (defmethod :m (data)
      (if (eq data 'B)
          (do-B)
          (if (eq data 'A)
              (do-A)
              (do-default-behavior)))))

(defsystem <<skeleton>> ()
  (class <foo> ()
    (defmethod :m (data)
      (do-default-behavior))))
(defsystem <<A>> ()
  (class <foo> (data)
    (method :m (data)
      (if (eq data 'A) (do-A) [:original data]))))
(defsystem <<B>> ()
  (class <foo> (data)
    (method :m (data)
      (if (eq data 'B) (do-B) [:original data]))))
(setup (list <<B>> <<A>> <<skeleton>>))
```

**Fig. 2.** Method Definitions in Ld-2

The second program in Figure 2 uses the system mixin feature and has the same meaning as the first one. However, its structure is much more modular. A `defsystem` defines a system mixin, which is a program fragment. The `setup` statement in the last line specifies the system mixins that should be composed into the complete program. The expression `[:original data]` is similar to the method invocation to the super class in traditional object-oriented languages. When the method `:m` is called, the fragment of method `:m` defined in `<<B>>` will be called. If the fragment evaluates `[:original data]`, the next fragment defined in `<<A>>` will be called.

The recursive-descent parser of EPP uses the system mixin feature. Each function which parses a non-terminal has hooks for extension that are actually methods which can be extended by other system mixins. The plugin programmer can add new `else-if` clauses to the methods to add new operators or new statements.

## 3 An Example of an EPP Plugin

In this section we introduce *Tiny Data-Parallel Java* as an example of an EPP plugin, show how it is translated, and discuss its performance.

```
#epp load "datap01"
public class CalcPi {
  public static void main(String argv[]){
    // Number of virtual processors.
    int num = 10000000;
    // Number of Java threads used for parallel execution.
    DpCalcPi.PE_n = 16;

    DpCalcPi.width = 1.0 / num;
    DpCalcPi p = new DpCalcPi(num);
    p.run();            // Execute the data-parallel method.
    System.out.println(DpCalcPi.result * DpCalcPi.width);
  }
}
parallel class DpCalcPi {
  static double result; // Mono variable.
  static double width;  // Mono variable.
  double x;             // Poly variable.
  public void run(){    // Data-parallel method.
    x = (VPID() + 0.5) * width;
    // Reduction. Calculate Σ 4 / (1 + x^2) .
    set_sum(result, 4.0 / (1.0 + x * x));
  }
}
```

**Fig. 3.** A Program That Calculates $\pi$.

### 3.1 Tiny Data-Parallel Java

As an example of EPP plugin, we implemented the *Tiny Data-parallel Java plugin* using the same translation technique as Data-Parallel C [7]. Some methods of Java objects are treated as data-parallel methods. From the programmer's point of view, the methods are executed on a large number of virtual processors.

Figure 3 shows a program which calculates the value of $\pi$. The program launches ten million virtual processors at the same time, each calculating a part of the surface giving us the value of $\pi$. The `run` method of class `DpCalcPi`, which is defined with the special modifier `parallel`, is the data-parallel method which is executed in parallel by virtual processors. EPP translates this method into a standard Java program that uses thread libraries and synchronization primitives. Other methods are not changed by the translator.

In parallel classes, there are two type of variables, mono- and poly-variables. Mono-variables are shared by all virtual processors and poly-variables are owned by each virtual processor. Static variables defined in the `parallel` class are mono-variables, while instance variables are poly-variables. Mono- and poly-variables can be used in arbitrary expressions in the data-parallel method. It is possible to access a poly-variable on another virtual processor by using the expression `x@[i]`.

The current implementation imposes some restrictions on how to write data-parallel methods. For example, no local variable can be declared within the data-parallel method, and remote variable accesses are not allowed in the middle of other expressions. In addition, current implementation performs no optimization. However, we think that this extension is a good demonstration of the suitability of EPP to language extension, in particular in the field of parallel programming.

### 3.2 The Translator

The source code of Tiny Data-Parallel Java is expanded into plain Java code that contains `for` statements simulating virtual processors, thread or process creation, and synchronization. Figure 4 shows some source code using the virtual processor ID (VPID) and its translation into plain Java. Virtual processors can

```
x = VPID();
System.out.println("VPID = " + x);

for (vpi = (0); (vpi) < (vpn); vpi++) {
  (vp_x)[vpi] = (((PE_i) * (vpn)) + (vpi));
  System.out.println(("VPID = ") + ((vp_x)[vpi]));
}
sync();
```

**Fig. 4.** Source Code and Translated Code of a Tiny Data-Parallel Java Program

be allocated to several Java threads, `PE_i` being the ID of each thread; `vpn` gives the number of virtual processors allocated to threads. In reality, `vpi` is a loop counter used to simulate the allocated virtual processors. Each poly-variable is represented as an array whose size is `vpn`. Therefore, a poly-variable `x` is translated to `(vp_x)[vpi]`. The method invocation `sync()` executes a barrier synchronization of all threads. This synchronization call is inserted in several places, such as before and after remote data access or reduction function calls.

### 3.3 Performance

Parallel execution of Tiny Data-Parallel Java has been tested in the following environments:

- A dual-processor Pentium Pro machine running Windows NT and JDK;
- A Cray CS6400 (a shared-memory machine with 32 SPARC processors) runing Solaris 2.5 and Toba, a Java-to-C translator which supports multi-threading.

Figure 5 shows execution time of the program which calculates $\pi$ with 10 million virtual processors.

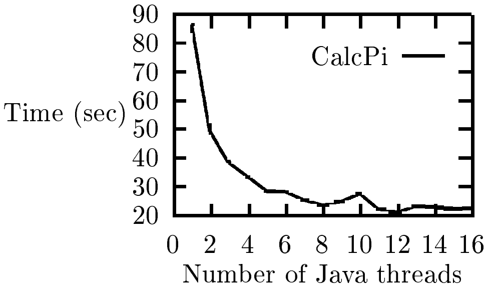


**Fig. 5.** Performance of $\pi$ Calculation on CS6400

## 4 Conclusion

We described the extensible preprocessor kit EPP, its description language Ld-2, and Tiny Data-Parallel Java as an example of EPP plugin. Ld-2 has a feature called system mixin which enables the program to be split into reusable modules. EPP is an application framework for preprocessors. The preprocessor programmer can implement EPP plugins, which are reusable and composable extension modules. EPP plugins can add new grammar rules to Java. Users can easily select the EPP plugins they need.

EPP and Ld-2 are currently implemented in Common Lisp, but we are implementing an EPP plugin which adds system mixin features to Java. EPP may then be rewritten in Java with this plugin.

## References

1. A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing company, 1987.
2. F. Bodin et al. Sage++: A Class Library for Building Fortran and C++ Restructuring Tools. In *Proc. of Object-Oriented Numerics Confs., Oregon*, April 1994.
3. G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. of ECOOP/OOPSLA'90*, 1990.
4. D. Caromel et al. Europa Parallel C++ version 2.1. In *Future Directions for Parallel C++*, June 1997. http://www.dcs.kcl.ac.uk/Future_Directions/.
5. S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, volume 30(10) of *ACM Sigplan Notices*, pages 285–299, Austin, Texas, October 1995. ACM Press.
6. J. Gosling, B. Joy, and Steele. G. *The Java Language Specification.* Java Series. Sun Microsystems, 1996.
7. P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.
8. Y. Ichisugi. EPP and Lods home page. http://www.etl.go.jp/etl/bunsan/~ichisugi.
9. Y. Ichisugi. Ld-2 users manual (DRAFT). Technical report, Electrotechnical Laboratory, 1997. can be obtained from [8]. In Japanese.

10. Y. Ishikawa. Meta-level Architecture for Extendable C++, Draft Document. Technical Report Technical Report TR-94024, Real World Computing Partnership, 1994.
11. Gregor Kiczales, Jim des Rivières, and Dan Bobrow. *The Art of the Meta Object Protocol.* MIT Press, 1991.
12. A. Kumeta and M. Komuro. Meta-Programming Framework for Java. In *The 12th workshop of object oriented computing WOOC'97, Japan Society of Software Science and Technology*, March 1997.
13. P. Launay and J.L. Pazat. Integration of control and data parallelism in an object oriented language. In *Sixth Workshop on Compilers for Parallel Computers (CPC'96) Aachen, Germany*, December 1996.
14. NPAC. HPJava reference and workshops site. http://www.npac.syr.edu/projects/javaforcse/.
15. G.L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.