Aspect-Oriented Design and Implementation of a Java Bytecode Analyzer Framework

Susumu YAMAZAKI*
Fukuoka Laboratory for Emerging & Enabling Technology of SoC
Fukuoka Industry, Science & Technology Foundation
816-8580, JAPAN
yamazaki@fleets.jp

Michihiro MATSUMOTO†
Fukuoka Laboratory for Emerging & Enabling Technology of SoC
Fukuoka Industry, Science & Technology Foundation
816-8580, JAPAN
michim@fleets.jp

Tsuneo NAKANISHI‡
Graduate School of Information Science and Electrical Engineering
Kyushu University
816-8580, JAPAN
tun@f.csce.kyushu-u.ac.jp

ABSTRACT

We propose a new type of Java bytecode analyzer framework based on aspect-oriented design and programming. We also observe that aspect-oriented design and programming improve separation of concerns of many of the characteristics of the design, including extensibility, type safety, and execution efficiency of its design and implementation, when compared to existing analyzer frameworks based on object-oriented design and programming. This paper reports how the following concerns are separated in our framework: the extension of elementary objects, the separation of parser and instruction set, the Visitor Pattern, binary operations and non-functional concerns such as verification.

1. INTRODUCTION

A Java bytecode analyzer framework has a wide range of applications, including bytecode-level optimizing compilers, ahead-of-time compilers, verifiers, aspect weavers. One of the most widely used frameworks is Soot [10], which has been created using extensible object-oriented design and implementation. As a result, it experiences some problems in separation of concerns, type safety, execution efficiency, *etc.*

Therefore, we propose a new type of Java bytecode analyzer framework based on aspect-oriented design and programming using AspectJ [5] (this framework uses Javassist [1] as a bytecode reader and writer). We observe that aspect-oriented design and programming improved separation of concerns of many characteristics of design, including extensibility, type safety, and execution efficiency of design and implementation of the framework, when compared to the existing analyzer frameworks based on object-oriented design and programming.

*Graduate School of Information Science and Electrical Engineering, Kyushu University

†Graduate School of Information Science and Electrical Engineering, Kyushu University

‡System LSI Research Center, Kyushu University

To appear in the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), March, 2004, Lancaster, UK.

1.1 Framework Overview

Java bytecode [7] is a variable length code based on the stack machine model. In our framework, a parser first translates the bytecode into a sequence of objects corresponding to each instruction. The instruction object is also based on the stack machine model. Although we do not currently support translation into 3-address code or the static single assignment (SSA) form, it may be supported later if necessary.

Our framework contains several analyzers. The most central of these is the dataflow analyzer, which can be easily customized. Our framework also contains interprocedural analyzers such as class hierarchy analyzers. Moreover, we can create a composite analyzer, consisting of other analyzers that are called when the composite analyzer is called.

1.2 Contributions and Organization

Through the design and implementation of the analyzer framework, we observed many advantages, both general and application specific. We also observed limitations in the current AspectJ. The rest of this paper outlines these advantages and limitations in the following order:

- We discuss the structured extension of elementary objects maintaining type safety and execution efficiency in Section 2.
- We propose the separation of an extendable bytecode parser from instruction sets in Section 3.
- We propose a simple process description of each instruction using the *Smart Instruction Visitor* in Section 4.
- We propose simple and extendable binary operations in Section 5.

- We discuss problems of description of a verifier as an aspect in Section 6.
- We discuss some related works in Section 7.

2. EXTENSIONS TO ELEMENTARY OBJECTS

It is often remarked that aspect-oriented programming improves separation of concerns. We point out that the most effective example of this is that of elementary objects of a framework, such as instruction objects.

For example, consider adding a new feature to an analyzer derived from a framework. A simple approach is to add fields or methods to the elementary objects in order to store the necessary information.

However, traditional object-oriented programming languages cannot add fields or methods to elementary objects structurally, and so they tend to 'bloat' chaotically. If a structure is enforced, the class hierarchy can become deep. In either case, maintainability and readability are degraded.

Within a framework, the extension of elementary objects is realized by using an indirect approach such as table or the Visitor Pattern [3], rather than by direct addition of fields or methods. For example, in Soot elementary objects are extended by adding tags. Tags are named, and may be requested by searching a table using this name.

The above techniques may sacrifice type safety or execution efficiency. Soot sacrifices both of them: the retrieved tag sacrifices type safety and must be cast downward before it may be used, and execution is inefficient because of the need for searching the table.

AspectJ can define fields and methods directly with classifications, as an aspect using an inter-type member declaration. For example, if we add a field or a method necessary for an analysis, we can define it structurally in an aspect concerned with the analysis.

Indirect extension mechanisms such as tags in Soot, are no longer needed. The type system of AspectJ ensures the type safety of added fields and methods, and because they are woven into classes directly, execution efficiency is improved when compared to indirect extension.

```
public class Parser {
    public static class Instruction {
        void setSucc
          (Instruction[] table, int pc) {}
        ...
    }
    public static interface Terminator {}
    public static interface Branch {...}
    public static interface Switch {...}
    static aspect addNextToSucc {
        pointcut addNextToSucc(Instruction inst,
          Instruction[] table, int pc)
            : call(void Instruction.setSucc
                   (Instruction[], int))
            && target(inst) && args(table, pc)
            && !target(Terminator);

        before(Instruction inst,
            Instruction[] table, int pc)
            : addNextToSucc(inst, table, pc) {
            ...
        }
    }
    static aspect addBranchTargetToSucc {
        pointcut addBranchTargetToSucc
          (Instruction inst,
           Instruction[] table, int pc)
            : call(void Instruction.setSucc
                   (Instruction[], int))
            && target(inst) && args(table, pc)
            && target(Branch);

        before(Instruction inst,
         Instruction[] table, int pc)
            : addBranchTargetToSucc(inst,
                  table, pc) {
            ...
        }
        ...
    }
}
```

Figure 1: Bytecode Parser using AspectJ

3. SEPARATION OF THE BYTECODE PARSER AND INSTRUCTION SETS

A bytecode parser scans binary class files, generates instruction objects corresponding to the byte sequences, and inserts labels. It also sets the relationships between instructions, for example using a succeed set, which is a set of instructions that may be executed after other instructions except those throwing exceptions in a manner similar to that of a Factored Control Flow Graph [2].

We now focus on setting succeed sets, which depend on the class of an instruction. Instructions are divided into non-terminator and terminator categories: a succeed set of a non-terminator includes the next instruction, while a succeed set of a terminator does not.

Instuctions may also be divided into non-branch, branch and switch categories: a succeed set of a non-branch instruction does not include any special jump target; a succeed set of a branch instruction includes one jump target; and a succeed set of a switch instruction includes two or more jump targets.

A succeed set can be determined by the classification, rather than by the instruction set, but the instruction set determines how a concrete instruction class is classified. In addition, another process, such as one detecting PEIs (potential exception-throwing instruction) [2], may require another classification. Therefore, because Java is a language that supports single inheritance and multiple supertypes, we must realize such a classification using `interface`.

However, Java does not allow `interface` to have concrete methods, so the process of setting succeed sets is distributed among code sections containing concrete instructions.

AspectJ solves this problem. Firstly, we provide two aspects to the parser. The first aspect is `addNextToSucc`, which adds the next instruction to the succeed set if the current instruction is a non-terminator. The second aspect is `addBranchTargetToSucc`, which adds the target instruction(s) to the succeed set if the current instruction is a branch or a switch. Secondly, we make a concrete instruction class implement the `interface` corresponding to the classification. Lastly, if the order of the succeed set is important, we can set the priority order using the `precedence` declaration between `addNextToSucc` and `addBranchTargetToSucc`. Figure 1 and Figure 2 show example code of a parser and a instruction set.

```
import Parser.*;

public class Aload extends Instruction {...}
public class Return extends Instruction
    implements Terminator {...}
public class Ifeq extends Instruction
    implements Branch {...}
public class Goto extends Instruction
    implements Terminator, Branch {...}
public class Tableswitch extends Instruction
    implements Terminator, Switch {...}
...
```

Figure 2: Java Bytecode Instruction Set Example

In general, if there are classifications into some given classes, and if the classifications determine the corresponding processes, we can write simple code so that the classifications and the processes are represented using `interface` and aspects, respectively.

Although multiple-inheritance has similar effects, this approach using aspects has two advantages: we can add a classification without modifying existing code, and we can also avoid the method confliction problem. For example, an instruction that is both a non-terminator and a branch is realized easily in AspectJ but cannot be realized naturally using multiple-inheritance.

4. THE SMART INSTRUCTION VISITOR BASED ON THE STACK-MACHINE MODEL

An operation corresponding to a given instruction often includes common processes. For example, because Java bytecode is based on the stack machine model, operations such as push or pop are commonly included in the operations corresponding to each instruction.

Therefore, we provide a Smart Instruction Visitor as part of our framework, based on the Java bytecode model, which is a domain-specific variant of the Visitor Pattern [3]. The programmer has access to four basic operations (`push`, `pop`, `load`, `store`) and the processes corresponding to each instruction. The programmer does not need to write all of these and can override only those necessary.

Because Java bytecode is a typed language, we provide variations of basic operations corresponding to different types. For example, `pushInt` corresponds to the type `int`. We also provide `push` and `pop` operations that handle values using the appropriate types for `getfield`, `putstatic`, *etc.* Moreover, we provide variations for basic operations corresponding to 32- and 64-bit types to satisfy the Java bytecode specification. Finally, we provide variations of `push` and `pop` corresponding to either two 32-bit types or one 64-bit type, for `dup2`, *etc.*

In our framework, we describe the process corresponding to each instruction as a method that is supplied an instruction object and zero or more arguments, and that returns zero or one result. For example, a process corresponding to the instruction `idiv` is defined as a method that is given an instruction object and two division values, and returns a result value object.

It is effective to define pointcuts for methods corresponding to instructions that have common features. This enables the methods to be defined structurally from various viewpoints.

```
public abstract class InstructionVisitor {
    ... // S1
    protected void push(Object value) {}
    protected void push2(Object value) {
        push(value);
    }
    ...
    protected void pushInt(Object value) {
        push(value);
    }
    ...
    protected void pushDouble(Object value) {
        push2(value);
    }
    ...
    protected Object pop() {
        return null;
    }
    ...
    protected void store
      (int index, Object value) {}
    ...
    protected Object load(int index) {
        return null;
    }
    ... // S2
    public static abstract aspect Pointcuts {
        pointcut intBinaryOperator
          (InstructionVisitor v,
           Instruction inst,
           Object value1, Object value2)
            : execution
              (Object InstructionVisitor+
               .at(Instruction+, Object, Object))
              && target(v)
              && args(inst, value1, value2)
              && args(Idiv, Object, Object)
              && ...;
        ...
    } // S3
    protected Object at
      (Iload inst, Object loadedValue) {
        return loadedValue;
    }
    protected Object at
      (Idiv inst,
       Object value1, Object value2) {
        return null;
    }
    ... // S4
    static aspect InsertCode {
        private abstract void Instruction.at
          (InstructionVisitor v);
        ...
        private void Iload.at
          (InstructionVisitor v) {
            Object value
              = v.loadInt(this.index);
            value = v.at(this, value);
            v.pushInt(value);
        }
        private void Idiv.at
          (InstructionVisitor v) {
            Object value2 = v.popInt();
            Object value1 = v.popInt();
            Object result
              = v.at(this, value1, value2);
            v.pushInt(result);
        }
        ...
    }
}
```

Figure 3: The Smart Instruction Visitor

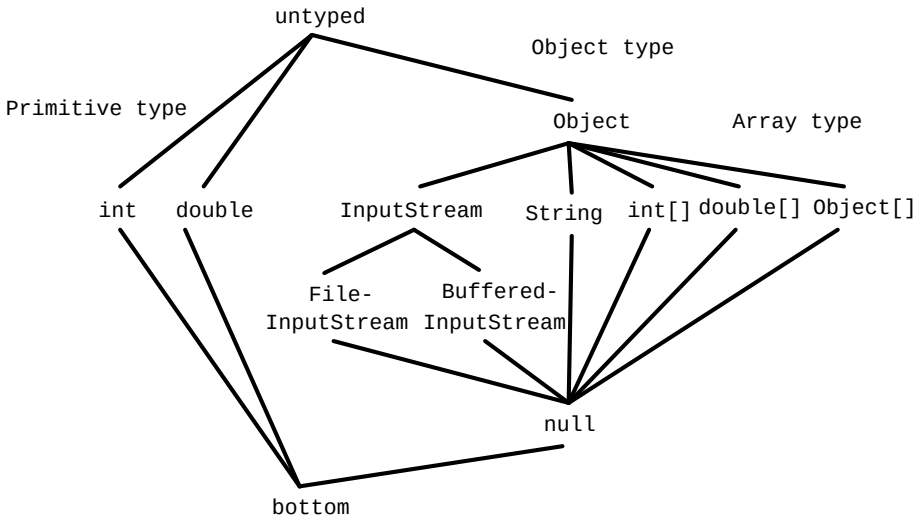

Figure 4: The Type Property Space for Java

Figure 3 shows the implementation of the Smart Instruction Visitor. The basic operations, various pointcuts, processes corresponding to instructions and inner processes, are defined from `S1`, `S2`, `S3` and `S4`, respectively.

The basic behavior is as follows: Methods receiving a Visitor are first defined using inter-type method declarations (`S4`). The corresponding basic operations and processes are called in these methods. For example, the inner method of `idiv` calls `popInt`, twice. The process corresponding to `idiv` is called with the instruction object and the returned values, and `pushInt` is called with the returned value.

We provide default implementations of basic operations and processes corresponding to each instruction. Relationships between the variations of basic operations are represented as an invocation from more constrained variation methods to the less constrained (`S1`). Therefore, all a programmer must do is to override the necessary methods.

5. SIMPLE AND EXTENSIBLE BINARY OPERATION

To realize a data flow equation as a framework, we need to implement the binary operation of properties. In type checking, for example, we must calculate the least upper bound ($\cup$) of the type property at the merge points [8].

Figure 4 shows a lattice representing the property space for type checking [6]. Bottom $\bot$ represents an initial value, so the least upper bound of property P and $\bot$ is P ($\bot \cup P = P \cup \bot = P$). Top $\top$ in type checking means untyped, and the least upper bound of property P and $\top$ is $\top$ ($\top \cup P = P \cup \top = \top$).

Next, the least upper bound of the same primitive type, such as `int`, is the type and the least upper bound of a different primitive type is untyped. For example, $P_{\texttt{int}} \cup P_{\texttt{int}} = P_{\texttt{int}}$, $P_{\texttt{int}} \cup P_{\texttt{float}} = \top$. Note that the rule of top and bottom precedes this rule, *i.e.* $P_{\texttt{int}} \cup \bot = \bot$.

Next, the least upper bound of the object type is a common ancestor. For example, $P_{\texttt{FileInputStream}} \cup P_{\texttt{BufferedInputStream}} = P_{\texttt{InputStream}}$. Note that the rules for bottom, top, and primitive types precede this rule. Moreover, the least upper bound of an object type is a class with zero or more interfaces, because Java provides single inheritance of class but multiple subtyping of interfaces.

Last, the least upper bound of `null` and the primitive type is untyped, and the least upper bound of `null` and the object type is the object type. Note that the rule of bottom precedes this rule.

Consider the implementation of binary operations with a least upper bound based on these rules. A naive implementation may use `instanceof`. For example, Figure 5 shows the implementation of a primitive type property, but this implementation is less extensible and maintainable. If we add a new type property, we must modify all `meet` methods, which calculate the least upper bounds. Moreover, if we change the order of precedence of the rules, we must swap the order of `if` in some methods.

Next, consider the implementation using double-dispatch, as shown in Appendix A. The advantage of this technique is that maintainability is improved because each method is simplified. However, the problems remain, as we must modify all property classes to add a new type property. We also must modify many classes to swap the precedence order of the rules. Moreover, we need to write the same process as many methods, such as the implementation of `Bottom` and `Untyped`.

AspectJ solves these problems simply (see Figure 6). The actual processes are implemented using `around` without `proceed` in the coordinator aspect. For example, `BottomCoordinator` describes the process of involving bottom and something else. Moreover, the process of combining different types is described in a *combination coordinator*. For example, a process of involving a combination of object types and primitive types is described in `ObjectAndPrimitiveCoordinator`.

Next, sort coordinators in `precedence` in topological order of lattice from the bottom. A combination coordinator precedes the coordinator of each property. The content of the method `meet` in the class `Property` is meaningless except when throwing an exception, when it is called with an

```
public class PrimitiveType extends Property {
    public Property meet(Property p) {
        if(p instanceof Bottom) {
            return this;
        }
        if(p instanceof Untyped) {
            return p;
        }
        if(p instanceof ...) ...
        ...
    }
}
```

Figure 5: The Implementation of the Primitive Type Property using `instanceof`

unexpected combination of properties.

This implementation solves the above problems. If we add a new type property, we must only write a coordinator and give the appropriate precedence order. If we must write a special behavior for combination with other properties, we must only write an appropriate combination coordinator. If we change the order of precedence, we must only modify the `precedence`. Moreover, we do not need to write the same process in many methods.

6. THE VERIFIER AS AN ASPECT

Our framework provides a bytecode verifier using a parser and a type checker. One of the advantages of aspect-oriented programming is the ability to unify the cross-cutting concern of a non-functional features such as verification. We have actually implemented the verifier in this way. An overview of our current implementation of the verifier is shown in Appendix B. During implementation, we found that there are two limitations in current AspectJ.

Firstly, AspectJ is not expressive enough to structure aspects. In our implementation, the verifier depends strongly on the inner structure of the parser and the type checker. So, not only must we modify the verifier whenever we modify the parser or type checker, but we cannot reuse the verifier with, for example, another instruction set. This problem is partially solved by using aspect structuring, *i.e.* dividing the verifier into parts that are dependent and independent of instruction sets. However, AspectJ cannot currently separate the verifier in this manner. Abstract pointcuts are useful, but insufficient to perform this separation.

It may not be possible to provide advice with information only from a pointcut. For example, we cannot provide advice to detect the overflow of the operand stack naturally, because its pointcut gives only an operand stack object as a parameter, and the object does not provide a method to retrieve the maximum stack size defined in each method.

This may be solved by defining advice and an inter-type field declaration by adding information about the corresponding method to the stack object. However, this is a specific and *ad hoc* approach.

7. RELATED WORK

Joeq [11] is an extensible virtual machine and compiler infrastructure. Of course, it can be used as a bytecode analyzer framework, and it has many sophisticated features.

Joeq provides the Visitor framework, enabling a simple analyzer implementation. Joeq can realize an analyzer by

```
public aspect Coordinator {
    declare precedence: BottomCoordinator,
        ...
        ObjectAndPrimitiveCoordinator,
        ObjectTypeCoordinator,
        PrimitiveTypeCoordinator,
        ...
        UntypedCoordinator;
}
public abstract class Property {
    public Property meet(Property p) {
        throw new RuntimeException
          ("unsupported property:"
           + this + ", " + p);
    }
}
public abstract aspect PropertyCoordinator {
    pointcut meet(Property p1, Property p2)
        : execution(
          Property Property+.meet(Property+))
        && target(p1) && args(p2);
}
public class Bottom extends Property {}
public aspect BottomCoordinator
  extends PropertyCoordinator {
    Property around(Property p)
        : meet(Property, p)
        && target(Bottom) {
        return p;
    }
    Property around(Property p)
        : meet(p, Property)
        && args(Bottom) {
        return p;
    }
}
...
public class ObjectType extends Property {}
public aspect ObjectTypeCoordinator
  extends PropertyCoordinator {
    Property around
      (ObjectType p1, ObjectType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        // calculate least upper bounds on types
    }
}
...
public aspect ObjectAndPrimitiveCoordinator
  extends PropertyCoordinator {
    Property around
      (ObjectType p1, PrimitiveType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        return new Untyped();
    }
    Property around
      (PrimitiveType p1, ObjectType p2)
        : meet(Property, Property)
        && target(p1) && args(p2) {
        return new Untyped();
    }
}
```

Figure 6: The Implementation of the Type Property using AspectJ

overriding the defined methods *in advance* for some situations, such as field accesses in an instance so, a programmer cannot unify arbitrary methods with the same behavior. On the contrary, our Smart Instruction Visitor can realize the analyzer using pointcuts, which can be defined freely by a programmer without performance penalty.

Joeq also provides a dataflow framework, where the binary operations of properties are defined in the centralized dataflow problem class. Though Whaley does not show an implementation detail for binary operations, it would be complicated for some analyzers, such as a type analyzer.

Though OVM [9] focuses on the virtual machine, its design policy can apply a bytecode analyzer. The main advantage of OVM is its memory efficiency; the OVM intermediate representation (OvmIR) uses the Flyweight Pattern [3]. Our current implementation, on the other hand requires more memory than OVM.

OVM also adopts the Runabout Pattern [4], making it more extensible, but with worse execution performance than the Visitor Pattern approach. This tradeoff is unavoidable when using Java and Java-based languages such as AspectJ. OVM focuses on the customization of the intermediate representation, so OVM has opted for extensibility and the Runabout approach but, because the main target of our framework is Java bytecode, we choose to optimize performance by using the Visitor approach.

Ideally, our approach should be mixed: in the early stage of development, we should take the Runabout approach, and when the specifications of the intermediate representations are almost fixed, we should switch to the Visitor approach. To make the switch easier, we will need an automatic code translator to convert from the Runabout to the Visitor.

8. CONCLUSIONS AND FUTURE WORK

We have built a Java bytecode analyzer framework that uses aspects, and have observed five advantages. Firstly, we realized extensions of elementary objects structurally and maintained type safety and execution efficiency. Secondly, we implemented a bytecode parser that is independent of any single concrete instruction set. Thirdly, we simplified the description of processes for each instruction using the Smart Instruction Visitor based on the stack machine model. Fourthly, we realized binary operations that are simple, extensive, and easy to maintain. Finally, we unified the description of a cross-cutting concern of a wide ranging non-functional features such as verification.

However, we also observed that AspectJ currently has two limitations: it is not expressive enough to structuralize aspects deeply on the basis of their inner structure; and it does not provide a general approach to write advice that cannot be described with its pointcut only.

In the future, we will build a bytecode translator framework based on aspect-oriented software development, enabling us to build many applications such as a bytecode-level optimizing compiler.

9. ACKNOWLEDGMENTS

This research was partly supported by a grant from the Cooperative Link of Unique Science and Technology for Economy Revitalization (CLUSTER) by Ministry of Education, Culture, Sports, Science and Technology (MEXT).

10. ADDITIONAL AUTHORS

Additional authors: Teruaki KITASUKA (Kyushu University, email: `kitasuka@f.csce.kyushu-u.ac.jp`) and Akira FUKUDA (Kyushu University,
email: `fukuda@f.csce.kyushu-u.ac.jp`).

11. REFERENCES

[1] S. Chiba. Load-time structural reflection in Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2000), Sophia Antipolis and Cannes, France.*, pages 313–336. Springer-Verlag, June 2000.

[2] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 21–31, Sept. 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] C. Grothoff. Walkabout revisited: the Runabout. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany.*, pages 103–124. Springer-Verlag, July 2003.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001), Budapest, Hungary.*, pages 327–353. Springer-Verlag, June 2001.

[6] X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999. Second Edition.

[8] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, Berlin, 1999.

[9] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, pages 67–76. ACM Press, June 2003.

[10] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *CASCON'99*, Sept. 1999.

[11] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME'03)*, pages 58–66. ACM Press, June 2003.

APPENDIX

A. DOUBLE-DISPATCH FOR BINARY OPERATION

Figure 7 shows a binary operation using double-dispatch. The behavior of this operation is somewhat complicated.

When the method `meet` is called, it calls the method `with*` corresponding to the class of the receiver `this`. For example, if the receiver is `Bottom`, it calls the method `withBottom`. Note that the receiver and the argument of the call is swapped. This realizes binary operations by defining processes corresponding to each class of receiver and the argument of `meet`.

B. OUR CURRENT IMPLEMENTATION OF THE VERIFIER

Figure 8 shows a section of the verifier code.

We can divide this into parsing-time verification and type-checking-time verification subsections. The former subsection includes the pointcut `insertLabel` and the `after` advice of `insertLabel`. The parser calls the method `insertLabel` when it finds a branch instruction. If the branch refers to a location outside the bounds of the code, `insertLabel` throws an `IndexOutOfBoundsException`. The advice of the verifier catches the exception, and rethrows a `VerifyException`.

The latter subsection includes the `stackUnderFlow` and `stackOverFlow` parts. The `TypeChecker` class extends our dataflow analyzer framework, and uses `LinkedList` in the Java class library as the operand stack.

We designed our framework to separate an analyzer from the target bytecode, *i.e.* the analyzer should not hold any analysing state information about the target bytecode, and the target bytecode should hold all of this analyzing state information. The operand stack then belongs to the target.

The `LinkedList` throws a `NoSuchElementException` when the list is empty and the method `removeFirst` is called, so the pointcut and the advice of `stackUnderFlow` catches the exception and rethrows a `VerifyException`.

In contrast, the implementation of `stackOverFlow` experiences a problem when attempting to retrieve the maximum stack size. The advice of `stackOverFlow` can access the operand stack and this join point. We may extract information about the type checker classes from this join point. According to our design policy, however, the type checker classes do not hold any code information, such as the maximum stack size.

On the other hand, the operand stack originally does not hold the maximum stack size because it is an instance of `LinkedList` in the Java class library. If we wish to add the maximum stack size to the stack, we must establish the maximum stack size of the list in advance. It is difficult to ensure this setting for general cases.

```
public abstract class Property {
    public abstract
      Property meet(Property p);
    protected abstract
      Property withBottom(Bottom p);
    protected abstract
      Property withUntyped(Untyped p);
    protected abstract
      Property withPrimitiveType
        (PrimitiveType p);
    ...
}
public class Bottom extends Property {
    public Property meet(Property p) {
        p.withBottom(this);
    }
    public Property withBottom(Bottom p) {
        return p;
    }
    public Property withUntyped(Untyped p) {
        return p;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        return p;
    }
    ...
}
public class Untyped extends Property {
    public Property meet(Property p) {
        p.withUntyped(this);
    }
    public Property withBottom(Bottom p) {
        return this;
    }
    public Property withUntyped(Untyped p) {
        return this;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        return this;
    }
    ...
}
public class PrimitiveType
  extends Property {
    public Property meet(Property p) {
        p.withPrimitiveType(this);
    }
    public Property withBottom(Bottom p) {
        return this;
    }
    public Property withUntyped(Untyped p) {
        return p;
    }
    public Property withPrimitiveType
      (PrimitiveType p) {
        ...
    }
    ...
}
```

Figure 7: The Implementation of the Type Property using Double-Dispatch

```
public aspect Verifier {
  pointcut insertLabel(Instruction inst, int pc)
    : call(void Instruction
      .insertLabel(Instruction[], int))
    && target(inst) && args(Instruction[], pc);
  after(Instruction inst, int pc)
    throwing (IndexOutOfBoundsException e)
    : insertLabel(inst, pc) {
      throw new VerifyException(
        "The target branch is out of bounds: "
        + pc + ":" + inst);
  }
  pointcut stackUnderFlow()
    : call(Object LinkedList.removeFirst())
    && within(TypeChecker);
  after() throwing (NoSuchElementException e)
    : stackUnderFlow() {
      throw new VerifyException
        ("stack under flow");
  }
  pointcut stackOverFlow(LinkedList stack)
    : call(void LinkedList.addFirst(Object))
    && target(stack)
    && within(TypeChecker);
  before(LinkedList stack)
    : stackOverFlow(stack) {
    int maxStack = ...; // how can we get?
    if(stack.size() >= maxStack) {
       throw new VerifyException
         ("stack over flow:" + maxStack);
    }
  }
}
```

Figure 8: The Implementation of the Verifier