Array Modeling in Java Virtual Machine

Wu Weimin[1], Li Kailun[2], and Su Qing[3]

[1] Computer Faculty, Guangdong University of Technology, Guangzhou 510006, China
[2] Guangzhou Branch, People's Bank of China, Guangzhou 510050, China

Abstract. Array is an important feature in Java and Java Virtual Machine. In spite of its importance, it has not been modeled by any existing Java Virtual Machine Models. In this paper, we define an extending model which uses an existing model as a basis and give the hierarchy of these two models, to model the array. In the extending model, we model the array in three steps. The first step is adding array related instructions in a formal way. The second step is refining the type compatibility to include array types. The last step is implementing array loading process also in a formal way. In the last part of thesis, we give the future work of extending other important features in Java and Java Virtual Machine.

1 Introduction

Java is an object-oriented programming language with a widespread use, and the java compiler translates Java source code to bytecode, which executes on the Java Virtual Machine (JVM).[1]

Compared to other object-oriented languages, there are many distinct features in Java, and array is one of them. And array is also an important feature in Java and JVM. The distinction and importance of it are as follows. First, in Java programming language, array is a most frequently used data structure to contain elements. Second, in Java type system, array type is a kind of reference type (the other two are class type and interface type). Third, in object creating, other then any other kinds of objects, array is a full-fledged object and is dynamically created, and it generally includes basic type array and object reference array. Fourth, in object loading, array loading is different from class or interface loading. Last but not least, in JVM instruction set, the JVM uses special bytecode to handle array. [2]

For this distinct and important feature in Java and JVM, it is significant to describe it in essence to help us in designing, programming, etc. The best and precise way to describe the essence is to build a model in a mathematical way. And because the JVM involves type, object creating and loading, and instruction set, so the most suitable model to build on is JVM model. After building the model, we can take the advantages as follows. First, it can help us precisely find the compiling error or runtime error which is directly or indirectly caused by misusing of array. Second, it can help us know what happens to the array behind the scene in a mathematical way, and thus help us design more robust programs.

M. Xu et al. (Eds.): APPT 2007, LNCS 4847, pp. 617–623, 2007.
© Springer-Verlag Berlin Heidelberg 2007

So far, several formalizations of the JVM model have been proposed. However, they provide only insight into one or few aspects of the machine, not the whole machine, and the array is not modeled in any of these models. So it means we need to extend an existed model to support array. The most rigorous and comprehensive one among these model is the machine proposed by Egon Borger and Wolfram Schulte (for clarity, we call it BSM, namely, Borger and Schulte Model). [3]. This machine can be validated and verified by standard techniques because it is defined by Abstract State Machines (ASM), which have a simple but precise semantic foundation. [4]

In BSM, the model can be described as a hierarchy of four submachines. Fig. 1. shows the hierarchy. The basic stack machine VMI supports instructions which are used for compilation of imperative programs. Typical instructions are: load and store a variable, apply arithmetic and relational operators, and jump. VMI is upgraded to VMC by including instructions which are used for the compilation of Java static features, such as class fields, class methods and class initializers. VMC can be extended to VMO which supports instructions for Java object-oriented features, such as instance creation, instance field access, instance calls with early or lately binding and type casts. And the topmost machine VME provides instructions with respect to exception [3]. And this structural decomposition is based on the orthogonality of various language features of Java. [5,6]

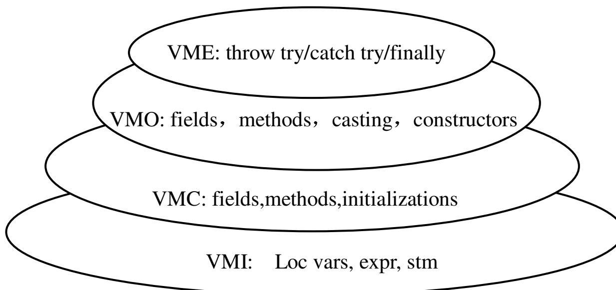

Fig. 1. Structural Decomposition of BSM

2 The Model of Extending BSM

Based on the current situation of JVM and the importance of array in JVM, we extend the BSM to support the function of array. And we call this extending model the Extending BSM (short for EBSM).

We add a VMA machine which supports array on top of the topmost model VME to extend the BSM. The main reason is as follows: when we extend the machine to support array, we should refine some functions and add some functions. If we separate these functions in four levels, then the description of the functions will not be centralized and the difference between BSM and EBSM will not be clear enough. And if we create a new level above the BSM, then these two problems will be solved. Fig. 2. shows the structural decomposition of EBSM.

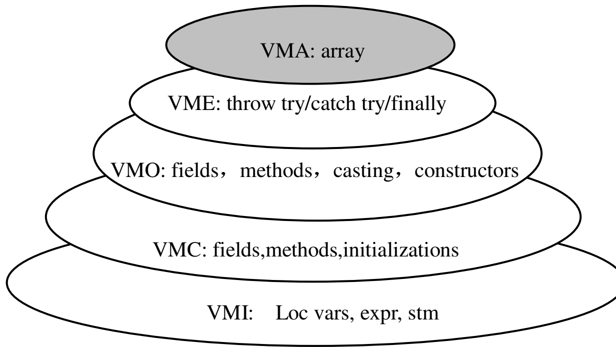

Fig. 2. Structural Decomposition of EBSM

3 VMA Modeling

We implement the VMA machine by three steps. First, we add 20 instructions which involving array into the instruction set of the JVM. Second, we refine the type compatibility to include the array type. Third, we refine the loading method to add the process of loading and linking array.

3.1 Adding Array Instructions

There are about 20 array related instructions in JVM. They can be divided into three kinds. The first kind is loading and storing array elements, contains aaload, aastore, baload, bastore, caload, castore, daload, dastore, faload, fastore, iaload, iastore, laload, lastore, saload, sastore. The most difference among these instructions are the type of the operand. Details of these instructions can be seen in [2]. The second kind is creating array, contains anewarray, newarray, multianewarray, which mean creating array of reference type, creating array of basic type and creating multiple array respectively. The third kind is getting length of array, which has only one instruction: arraylength. The first and the second kind are more important than the third kind, so we describe these two kind instructions in detail.

3.1.1 Instructions of Loading and Storing Array Elements

Because the execVM part of the BSM, which defines the process of instruction executing, uses the free data type to abstract the difference between non-array type, including reference type and basic type, so the first kind of instructions can be abstract to loadarrayelem and storearrayelem, which mean loading array element and storing array element respectively. Prog. 1. shows this kind of instructions.

Prog. 1. Instructions of loading and storing array elements

```
execVM (redef)==…
loadarrayelem()•
if newopd•wr•wi=opd• #wr =r• #wi=i  then
newopd(#newopd+1):=r[i]
```

```
pc:=pc+1
storarrayelem()•
if newopd•wr•wi•wv=opd• #wr=r• #wi=i• #wv=v then
r[i]:=v
pc:=pc+1
```

3.1.2 Instructions of Creating Array

To implement the instructions of creating array, we should first define a type called AState to denote the state of an array. Prog. 2. shows the AState type and its related State and InitialState.

Prog. 2. AState Type

```
Type                 AState:= NotInited | Inited
State                aState:ANm • AState
Initial State      aState(c)=NotInited
```

To implement the instruction newarray, we define five steps. First, we get the array type according to the basic type in parameter using the function arraytype. Second, if the class of the array is not already loaded, we should first load the class of the array type using callLink function. Third, we set the class and dimension of the new created reference using function aOf and countOf respectively. Fourth, we initialize the array using the default values. Last, we update the operand and the PC register. The implementation of instruction anewarray is similar to the newarray. For clarity, we do not describe it in detail.

The most distinct differences between newarray and multianewarray is the latter uses a function initArrayElem to init the elements of the multiple array using function elemType which returns the element type of the array argument. The element may also be an array, so this call may be recursive. Prog. 3. shows the instructions of newarray and multianewarray.

Prog. 3. Instructions of newarray and multianewarray

```
execVM (redef)==…
newarray(t) •
at := arraytype(t)
if newopd •wc = opd • # wc =c then
if aState(at) := NotInited
    callLink(cLd(meth), cNm(f))
aOf(r) := at
countOf(r) := c
for all e in afield(at)
elem(r ,e) := default(e)
```

```
newopd :=new opd • [r]
pc := pc +1
where r = new (dom(aOf))

multianewarray(t, d) •
at := multiarraytype(t, d)
if newopd •wc1…•wcd = opd • # wc1 =c1 •… # wcd =cd then
if aState(at) := NotInited
    callLink(cLd(meth), cNm(f))
aOf(r) := at
i := 1
countOf(r) := ci
for all e in afield(at)
elem(r ,e) :=(i = d) ?default(e): initArrayElem
(elemtype(at),ci+1 )
newopd :=newopd • [r]
pc := pc +1
where r = new (dom(aOf))

initArrayElem(at, ci)
aOf(r) := at
countOf(r) := ci
for all e in afield(at)
elem(r ,e) := (i = d) ?default(e)
:initArrayElem(elemtype(at),ci+1 )
where r = new (dom(aOf))
```

3.2 Type Compatibility of Array Type

Because the propagateVM part (which defines the process of verifying byte code) and the execVM part of the BSM involves type compatibility, so we refine the function compat and the operator '≤'to include type compatibility of array.

Prog. 4. shows the refined function and operator. According to the JVM specification[2], it is compatible when two arrays are of the same dimension and the element types of the two arrays are type compatible. In model, the function isArray returns true if parameter is actually an array, and the function dim returns the dimension of the array argument.

Prog. 4. Type compatibility of array type

```
isArray(C1) • isArray(C2) • dim (C1) = dim (C2) •elemType
(C1) • elemType(C2)

• Compat(C1, C2) = true

isArray(C1) • isArray(C2) • dim(C1) = dim(C2) •elemType
(C1) • elemType(C2)

• C1 • C2
```

3.3 Array Loading

For array loading, if the element type of the array is a reference type, then according to the JVM specification[2], JVM first loads the element type (may lead to recursively loading), and then adds the array type to the name space of the environment. If the element type of the array is a basic type, then just adds the array type to the name space of the environment.

Because no matter what classLoaders (system or user-defined) are defined, the function of loading and linking are eventually found in findSystemClass, defineClass and resolveClass in class called classClassLoader, so we refine the InvInstance method taken each of these three method names as argument in the execVM of BSM for array loading. Prog. 5. shows the refining rule for findSystemClass. We refine similarly the execution rules for the other two methods. For clarity, we do not show it in Prog. 5. The black part of the program shows what we have refined.

Prog. 5. Instructions of array loading

```
execVM (redef)==…

InvInstance(bind, findSystemClass) •

if cinitd(cNm(findSystenClass)) •newopd •[ld, cn] = opd
•ld <> null then

  let c = (sysLd, cn) in

if unloaded(c) then

     if ¬ isArray(c) then

        loadVM(c)

     else

        bc := elementType (c)

        if(isReferenceType(Class(bc))) then

            loadVM(bc)

            addArrayToEnv(c, lc)

        else

            addArrayToEnv(c, lc)

else if ¬ cinitd(c) then
```

```
      if ¬ isArray(c) then
linkVM(c)
      else
            bc := elementType (c)
          if(isReferenceType(Class(bc))) then
               linkVM(bc)
               addArrayToEnv(c, lc)
          else
             addArrayToEnv(c, lc)
else
     opd := newopd•[ldEnv(c)]
    pc := pc +1
```

4 Conclusion

Array is an important feature in Java and JVM. The best and precise way to describe the essence of it is to build a model in mathematical way. In this paper, we define an extending model which uses an existing model as a basis and give the hierarchy of these two models, to model the array. In the extending model, we model the array in three steps.

Acknowledgment

The future work is to model the other important features in Java and JVM which have not been modeled by existing JVM . models.

References

1. Gosling, J., Joy, B., Steele, G.: The Java(tm) Language Specification. Addison-Wesley, Reading (1996)
2. Lindholm, T., Yellin, F.: The Java(tm) Virtual Machine Specification. Addison-Wesley, Reading (1996)
3. Borger, E., Schulte, W.: Modular Design for the Java Virtual Machine Archicture. In: Archicture Design and Validation Methods (2000)
4. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Borger, E. (ed.) Specification and Validation Methods, Oxford University Press, Oxford (1995)
5. Borger, E., Schulte, W.: Defining the Java Virtual Machine as platform for provably correct Java compilations. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, Springer, Heidelberg (1998)
6. Borger, E., Schulte, W.: A programmer friendly modular definition of the semantics of Java. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java(tm), Springer, Heidelberg (to appear, 1999)