Introduction to the Batu Virtual Machine

Yngve Devik Hammersland

November 3rd, 2007

1. Introduction

BVM is a virtual machine for the Batu language. Batu and BVM was created by Roland Kaufmann for the institute of informatics in 2000. BVM is implemented as a java program with a graphical user interface which can be used to visualize the execution of a Batu program.

2. The Batu Language

A Batu program is a text file containing a set of directives, labels and instructions.

Directives denote amongst other things the number of globals variables are needed and how much stack space are needed by a subroutine.

The instructions of Batu takes one or none operands. The values that the instructions work with is residing on the stack. There are instructions to do arithmetics, comparisons branching, I/O and more.

Batu is actually a subset of the JVM[1] language not counting the `read` and `print` instructions.

3. The Runtime Environment

The Batu runtime environment consists of a set of global variables, a set of local variables and a stack. BVM has no registers.

3.1. Variables

The local and global variables are stored in arrays. There are instructions to access these arrays with indices. This is the only way to access them.

3.2. Stack

The stack is a LIFO structure and are manipulated through special instructions. As well as the mandatory push and pop instructions there are also instructions to duplicate and swap elements.

3.3. Datatypes

Batu has only two datatypes; integers and arrays. Integers are stored by value and arrays are stored by reference.

Variables are not typed but are type safe. This means that any datatype may be stored in a variable but a variable cannot be accessed as another datatype than it really contains. I.e. the iadd (integer addition) is only possible if there are two integers on the stack. Otherwise a runtime error will occur. All memory in BVM can contain both; both the stack and both local and global variables.

There are different instructions for manipulating the different datatypes. I.e. `istore` and `astore` stores integers and arrays respectively.

3.4. Functions and Jumping

A BVM program is made up of a number of functions. These in turn are made up of a sequence of instructions. It is possible to jump to any instruction within a function but not to another function. Cross-function jumping can only be done through an explicit function call and return.

Functions are addressed by name as opposed to addressing by address. This allows for generation of code without having to look up the addresses of functions in the symbol table.

BVM function names are *mangled*; this means that the function name, its type and the number and type of parameters are combined into a name. This allows for function overloading. The name "mangling" is described in the next section.

[1] Java Virtual Machine

A Batu function is declared using the `.function` directive and the mangled name of the function.

Every instruction in a Batu program must occur inside a function. The starting point of any program is the function with the (mangled) name `main()I`.

Jumping are accomplished through the jump instructions. These instructions take one operand which is the relative distance between the jump instruction and the target instruction. I.e. a jump length of 1 will jump to the next instruction, -1 to the previous and 0 will cause an infinite loop. Jumps outside the boundary of the containing function will cause undefined results.

3.5. Name Mangling

A function name in BVM is composed of:

- The identifier.
- Mangled type names for all its parameters enclosed in `( )`with no spaces or commas.
- The mangled name for its return type.

Mangled names for the simple types are:

- `I` for integers.
- `V` for void.

For arrays [followed by the size and base type. The size is only specified if known. See table 1 for examples.

3.6. Frame

When execution control is transferred to a function, it is executed inside its own frame. A new frame is allocated for each function call and is discarded as when the function call returns. Frames consists of local variables and a stack.

This frame serves as the activation record of the function call. A function only has access to its current activation record.

The size of the stack and the array of local variables are specified with directives in the Batu program. These directives follow immediately after the `.function` directive. They are named `.locals` declares the number of local variables required by the function and `.stack` declares the required stack size.

3.7. Local and Global Variables

All the functions have access to the global variable array, but they only have access to the local variable array in its own frame. There is no way to access local variables of other function.

These arrays are not used directly. There are special instructions to read and write integers and arrays in the local and global arrays. Since the variables are addressed by index as opposed to functions which are addressed by name, the compiler has the responsibility to assign indexes to variable names.

3.8. Local Stack

As with the local variables, the local stack resides in the current frame and only the current stack is available to the function.

Since a new frame is allocated when a function is called the stack is empty.

3.9. Parameter passing

When calling the BVM will take the parameters from the stack and move them to the local variables of the newly created frame. The last parameter is on top of the stack. Also the local variable at index 0 will receive the first parameter, the variable at index 1 the second and so forth. Note that the parameters are removed from the callers stack.

This means that the number of local variables needed is including the number of arguments. Also, since arrays are stored by reference, they will also be passed by reference. I.e. any array parameter will

Function Declaration	Mangled Name
`int main() {...}`	`main()I`
`void foo(int i, int j) {...}`	`foo(II)V`
`int bar(int i[], int j, int k[]) {...}`	`bar([II[I)I`

Figure 1

retain any changes made by the called function.

When returning, if the function has a return type, the value at the top of the stack is transferred to the top of the callers stack. And in true type safe fashion there are distinct instruction to return integers, arrays or nothing (void functions).

Execution then resumes where it left off when the function was called.

4. Batu Program Files

Batu program files are stored in a plain text file. This file must conform to the following format:

- Most importantly, no whitespaces are allowed other than the ones indicated in the following rules.
- The format is line based. This means that a newline separates the directives and instructions.
- Directives are placed at the start of a line and there must be only a single space between the directive and the argument:
  `.directive argument`
- An instruction is always prefixed with a tab character. As with the directives only a single space must separate the instruction and its argument. If the instruction has no operand it must be immediately followed by a newline.
- A program starts with the declaration of the number of global variables required. I.e. a program requiring 4 variables are declared as follows:
  `.globals 4`
- A function is declared with 3 directives. One declares its name whereas the other two specifies the number of local variables and its required stack size. I.e:
  ```
  .function foobar(I)V
  .locals 2
  .stack 3
  ```
- Instructions names must be written entirely in lower case.
- Comments are prefixed with `;` and terminated by newline. (Hint: They are very useful when debugging your compiler.)
- Comments do not count as instructions. That is, they do not count when performing jumps.
- An `*` in front of an instruction denotes a breakpoint.

5. Example

```
;int foo(int i) {
.function foo(I)I
.locals 3
.stack 1
;   int a, b;
   iconst_0
   istore 1
   iconst_1
   istore 2
;   return i;
   iload 0
   ireturn
;}
   iconst_0
   ireturn
;int main(void) {
.function main()I
.locals 1
.stack 2
;   int a[10];
   ldc_w 10
   newarray
   astore 0
;   foo(a[0])
   aload 0
   ldc_w 0
   iaload
   invokestatic foo(I)I
   pop
;}
   iconst_0
   ireturn
```

Figure 2

6. Hints

- To make the frame of a function then:
  - All parameters must be assigned to a (local) variable index
  - All local variables must be assigned to a variable index
  - The number of local variables must be calculated

  - The maximum stack depth must be calculated.

- Variables must be initialized before they can be used. This is safest to do immediately after declaration.
- Remember that the number of local variables includes the number of parameters (but the must obviously not be initialized)!
- You can insert the `output(I)V` and `input()I` functions into the symbol table before the analysis starts. When generating code for function calls you can check whether the call is for either of these functions and generate the appropriate instruction. Otherwise you generate an `invokestatic` instruction using the mangled function name.
- There is no instructions for comparison operators; you have to translate these to a small branch where you leave 0 (false) or 1 (true) on the stack according to the result of the operation. Remember that comparison result can be stored in variables.
- If all nodes in the syntax tree returns a vector containing its generated code, then its length will tell you how far you have to jump to clear it.
- Don't write code straight out. Rather collect the instructions in a vector. Each production generates a vector using its children's vectors along with some code to glue these together.
- There is no instruction for logical NOT; instead you may use the pattern:
  `iconst_1, ixor`
- To find the maximum stack depth is a classical DFS algorithm. The stack depth equals the order of the syntax tree.
- Expressions must leave their value on the stack. Since assignment also is an expression and `i/a/iastore` removes the value it stores you need to duplicate its value using the `dup` instruction.