Jeliot 3 Intermediate Code Specifications
MCode

Niko Myller and Andrés Moreno García

3rd March 2004

Contents

Chapter 1

Introduction

This document contains the specification for the MCode, intermediate language code used in Jeliot 3 program visualization system.

1.1 Purpose

This document main intention is to lay a solid ground for MCode, so future modifications, additions and queries to it will have a clear reference within this document.

1.2 Scope

This document attaches to Jeliot 3. MCode was developed in order to provide the execution information to an interpreter, which manages the visualization scene. The interpreted language is Java and the interpretation is done by DynamicJava. DynamicJava is a Java interpreter written in Java. Java and DynamicJava imposes some properties to the MCode. Furthermore, the original target users (programming novices) and their programs, also delimits the Java features that are currently supported by the MCode, such features as threads and reflection utilities are not supported. More on this will be explained in the following chapter.

1.3 Definitions, acronyms and abbreviations

Here it is included some definitions of used word to help its comprehension.

1.4 References

Interested readers should point to the master theses written by Niko Myller and Andrés Moreno, both can be found at Jeliot 3's webpage: `http://cs.joensuu.fi/jeliot/` . Niko's thesis (Myller, 2004) describes Jeliot 3 system and its implementation. Andrés' thesis (Moreno, 2004) explains further the decisions that shaped this intermediate code and compares it with different solutions, describing a taxonomy.

1.5 Overview

This language will be used to transfer evaluation information between Dynamic Java and Director class of Jeliot 3. The information flows to one direction, from Dynamic Java to Director (with a possible exception of the Input statements). Section 2 will provide a background, introducing Jeliot 3 system and some features of MCode.

Chapter 2

Overall Description

2.1 Jeliot 3 System Structure

2.2 MCode functions

MCode is the intermediate language used to communicate the visualization engine and the Java interpreter (DynamicJava). It mainly flows from DynamicJavato the Director. The information that carries along within is not only the information about modified variables and modified method stacks, but also the operations that produce such changes. and the results of those operations. This is the biggest difference from normal compiler intermediate codes; they only indicate what operations to perform to the assembler. In our case all operations are performed and the every result is sent to the Director.

2.3 User Characteristics

This document is addressed to the following people:

Developers of Visualization Systems This document may provide ideas and solutions to developers of new visualization systems, as well as provides of an existing solution that can be merged into their ongoing projects.

Maintainers of Jeliot 3 Future developers and maintainers of Jeliot 3 should refer to this document when modifying its source code, specially those files referring to DynamicJava and Jeliot 3's interpreter. They should incorporate here all changes done to MCode to provide a state of the art document as well.

2.4 Features and Constraints

While trying to produce the more generic intermediate code some features and constraints where placed due to several reasons. The most important ones are the following ones:

Java Being Java the language of choice, MCode supports most of its characteristics and abilities. It is object oriented, so things as method calls, objects and, to some extend, inheritance are supported.

DynamicJava Because of using DynamicJava as an interpreter from which to extract the execution information, some of the specified ordered sequences of instructions may be too constrained to the particularity of DynamicJava. However that is not the case for the most of instructions explained here.

Targeted audience MCode was designed to support the development of MCode, thus it was important to address their specific problems: MCode has a great detail in the evaluation of expressions and the normal flow of program, so novices will grasp the programming fundamental issues. Meanwhile, more advanced features are not within the scope of the MCode and have not been implemented. Those features include threads, reflection utilities and exception handling.

2.5 Dependencies

Right now MCode is only produced through the modified version of DynamicJava included in the distribution of Jeliot 3. However it is up to anyone to create a new high-level interpreter to produce its own MCode to be run by Jeliot 3's MCode interpreter or by one they develope. There is also the possibility to store the MCode a single text file and be retrieved later by a MCode interpreter to produce visualization orders. However, this configuration would not allow INPUT/OUTPUT operations as they provide information to the evaluation needed to carry on with following instructions.

Chapter 3

MCode Language

3.1 Introduction

MCode is defined to be an interpreted line by line and each line can be divided in the smaller pieces or tokens. Here we define all the commands that are part of MCode. First of all, we will introduce the notation used in this specification. as it is not standard. In our notation each token is now separated by single ’§’ character, this token will not be shown hereinstead a blank space will be used to help readability. Each token is first introduced as an English name. Later a sample instruction is given, consisting of the different tokens that build that particular instruction. If the token is in big letters it is a preserved word. If it is in small letters it will be replaced by variable value or integer. Table 3.1 shows the abbreviations used in the specification to refer to some common tokens:

Name	Abbreviation	Meaning
Instruction Reference	ir	Used to keep track of instructions used in the past.
Left Instruction Reference	lir	A reference to a previous instruction used as the left operand.
Right Instruction Reference	rir	A reference to a previous instruction used as the right operand.
Instruction Counter	ic	Counter of expressions, making instructions unique and easily referenciable
Type of instruction	ti	Refer to some m-code instruction (e.g. ADD)
Location	lo	Variable that holds the location of the expression in the source code, defined by "‘beginning of line"’, "‘beginning of column"’, "‘end of line"’, "‘end of column"’

Table 3.1: Token Abbreviations.

The usual MCode sentence will consist of:

Expression/Statement code A shortcut for every Java statement or expression is used: e.g. AE stands for Add Expression. The chosen names are heavily related to the ones used in DynamicJava.

Reference Every Expression/Statement sentence is identified by a number. This way nested statements and expressions can be formed up from previous m-code sentences. itemRelated References Most of the m-code sentences refer to previous m-code sentences. One Add Expression will refer to the references of both sides of expression. Flow-control statements will refer to a condition expression, and so on.

Value Most sentences will return the value resulting from the executing of an expression. If it is a flow control statement it will return a Boolean value indicating the result of the condition.

Type Every expression that has a result must specify its type.

Location This contains the location of the expression in the original source code file.

3.2 Constants

Table 3.2 contains the constants used in the implementation of MCode to support portability and maintainability.

Constant	Meaning
DELIM	Used to separate tokens on a single instruction. They have to be explicitly added to the m-code instruction
LOC DELIM	Used to separate the different coordinates that locate some source code.
UNKNOWN	When some variable value cannot be accessed it is given an UNKNOWN value.
NO REFERENCE	??
REFERENCE	??
NOT FINAL	??
FINAL	??
TRUE	String that represent the TRUE Boolean value in the working environment.
FALSE	String that represent the FALSE Boolean value in the working environment

Table 3.2: Constants.

3.3 Auxiliary Instructions

Some auxiliary instructions are defined in order to ease the interpretation of the m-code. These instructions are not related to any particular Java construct and are used by those that need them.

3.3.1 BEGIN

```
BEGIN ti ir loc
```

Instruction used to mark the beginning of those instructions that admit instructions to be encapsulated within them. Those instructions are Assignment, Return, Parameter, Array Access and all unary and binary operations; and they are referred through it. The referred instructions get they ir assigned in the BEGIN instruction.

3.3.2 LEFT and RIGHT

```
LEFT/RIGHT ir
```

These instructions are similar to BEGIN. Both of them are used to mark the beginning of the left/right side of a binary operation. The ir obeys the same reason than in BEGIN, "‘reserves"’ the reference number for the following instruction.

3.3.3 TO

```
TO ir
```

This instruction is used in assignments and reflects the movement of the value to the left hand side of the assignment. ir points to the qualified name that will hold the value. As said before, this instruction is used in assignment and more concretely in Assignment, Variable Declaration (those with initializer) and Compound Assignment.

3.3.4 ERROR

```
ERROR errorMessage loc
```

Parser and execution errors are reported to the visualization engine with the ERROR instruction. errorMessage is a string that can contain HML and it is what will be visualized.

3.3.5 END

```
END
```

END is produced at the end of the m-code program and indicates the visualization to terminate.

3.3.6 SCOPE

```
SCOPE 1/0
```

New blocks of Java code are delimited in m-code through SCOPE. This instruction second token indicates whether it is opening a new one (1) or closing one (0).

3.3.7 CONSCN (Constructor Call Number)

3.4 Statements

3.4.1 A (Assignment)

```
A ir rir lir value type loc
```

The assignment instruction is composed by its own reference (ir) and the references to the left and right hand sides (lir, rir). Furthermore it contains the assigned value and its type.

It is worth to mention that compound assignments are decomposed into the operation and a simple assignment.

3.4.2 VD (Variable Declaration)

```
VD name NO_REFERENCE value type FINAL/NOT_FINAL
loc
```

When declaring a variable the corresponding the m-code instruction needs to be complemented with its name, value, type and the modifier (FINAL or NOT_FINAL). NO_REFENRENCE

3.4.3 Binary Operations

```
binaryCode ir rir lir value type loc
```

Binary instructions are composed by its bynaryCode,its own reference (ir) and the references to the left and right sides of the expression(lir, rir). Furthermore it contains the assigned value and its type. Binary codes can take any of the following values:

Boolean Operators	MCode	Arithmetic operators	MCode	Bitwise operators	MCode
AND &&	`AND`	ADD +	`AE`	AND &	`BITAND`
OR ‖	`OR`	SUBTRACT	`SE`	OR ‖	`BITOR`
XOR ˆ	`XOR`	MULTIPLY *	`ME`	XOR ˆ	`BITXOR`
LESSER THAN <	`LE`	DIVIDE /	`DE`	LEFT SHIFT «	`LSHIFT`
GREATER THAN >	`GT`	REMAINDER %	`RE`	RIGHT SHIFT »	`RSHIFT`
EQUAL ==	`EE`			UNSIGNED SHIFT »>	`USHIFT`
NOT EQUAL !=	`NE`				
LESSER OR EQUAL THAN >=	`LQE`				
GREATER OR EQUAL THAN =<	`GQT`				

Table 3.3: Binary Operators

3.5 Unary Operations

```
unaryCode ir reference value type loc
```

As with binary operatoions the unary instructions take the same tokens. The only difference is that there is only one reference to another instruction. value, type and loc maintain the same meaning. As before there are several Java unary operators, all of them are listed in the following table:

Boolean Operators	MCode	Arithmetic operators	MCode	Bitwise operators	MCode
NOT !	NO	POSTINCREMENT ++	PIE	COMPLEMENT ˜	COMP
		POSTDECREMENT –	PDE		
		PREINCREMENT ++	PRIE		
		PREDECREMENT –	PRDE		
		PLUS +	PLUS		
		MINUS -	MINUS		

Table 3.4: Unary Operators

3.6 Literal constant and variable access

3.6.1 Qualified Name

```
QN ir name value type
```

Qualified names are those local variables to the method or block. mcode instruction contains the reference (ir) of the instruction and the name value and type of the qualified name. If the variable is not initialized value will be UNKNOWN.

3.6.2 Literal

```
L ir value type loc
```

Literals are the constants values in the source code(e.g. 3 is one integer literal and "‘3"’ is one string literal). The m-code instruction contains all the information needed for the visualization as well as its reference (ir): value, type and loc.

3.7 Control Structures

3.7.1 If Statements

```
IFT/IFTE condition value loc
```

There are two possible instructions for an "‘If"’ statement. IFT if there is not else statement or IFTE if there is. The composition, however, is similar. condition is the reference to the instruction that evaluate the condition. value holds the result of the evaluated condition and will tell which branch execution is following. loc as usual contains the code location.

3.7.2 While For and Do-While Statements

```
WHI/FOR/DO condition value round loc
```

These statements produce a similar to the previous one. They only differ in the round token. This token holds the number of iterations the while loop has made.

3.7.3 Switch

Three m-code instructions are related to the switch statement

```
SWIBF selector ir loc
```

```
SWITCH loc
```

```
SWITCHB loc
```

3.7.4 Break and Continue

```
BR/CONT statement loc
```

Break and continue asserts instructions only specify which statement they are in. Allowed values for statement are WHI FOR IFT IFTE SWITCH, those were you can find these statements.

3.8 i.Input and Output

Missing !!!!!

3.9 Arrays Handling

3.9.1 Array Allocation

```
AA ir arrayHashCode type dimension
dimensionsReferences dimensionsSizes loc
```

Array allocation instruction is a complicated one, as it carries a lot of information about the array. As usual a ir is provided. Following the arrayHashCode of the object created to allocate it. It can be any other number that identify one-to-one the allocated object on the interpretation. type contains the type of the array components. dimensionReferences is a comma separated list with the references to the instructions that evaluated the sizes' expressions of the array. Finally dimensionsSize is another comma separated list where each element is the size of a dimension. new Integer[4][5] will produce AA ir 456744 Integer 2 ir1,ir2 4,5 loc. Where ir1 and ir5 must be references to the literal instructions of "'4"' and "'5"'.

3.9.2 Array Access

```
AAC ir arrayNameReference deep cellReferences
cellNumbers value type loc
```

Array accesses instructions consist on different tokens. The common ones (ir, value, type and loc) are also present. But we can find very specific ones. arrayReference points to the instruction produced when visiting an array name, normally a qualified name. deep refers to the level of deepness of the array access. This is useful for multidimensional arrays when they are not accessed till the last level, the one holding the data value. cellReferences and cellNumber meet the same purpose than dimensionReferences and dimensionsSizes in array allocation (AA) instruction. cellReferences points to the instructions that evaluated the value of each cell pointer. cellNumbers are the actual values of the cell access. Both of them are presented as a comma separated list. For example in array[3][5], cellReferences will point to the literal instruction of "'3"' and "'5"' and cellNumbers will contain "'3, 5"'.

3.9.3 Array Length

```
AL objectCounter "length" value type loc
```

3.10 Object Oriented

Bibliography

Moreno, A., Jun. 2004. Taxonomy of Intermediate Codes Used in Visualization and its Aplication in Jeliot 3. Master's thesis, University of Joensuu, Computer Science Department, andres' thesis.

Myller, N., Mar. 2004. The Fundamental Design Issues of Jeliot 3. Master's thesis, University of Joensuu, Computer Science Department, niko's thesis.