JEDI: An Interactive Interpreter for JAVA

Hermann Oliveira Rodrigues
hermann@dcc.ufmg.br

Carlos Camarão
camarao@dcc.ufmg.br

UFMG, Belo Horizonte 31270-010, Brasil
Phone: +55 31 3499 5889
Fax: +55 31 3499 5858

August 15, 2001

Abstract

JAVA is a general purpose concurrent class-based object-oriented programming language. This work presents an interactive program that interprets JAVA source programs directly. The cornerstone of its execution model is an execution graph, in which each node is constituted by three functions: a state transformer, a continuation and an exception handler.

1 Introduction

JAVA[1] is a general purpose concurrent class-based object-oriented programming language.

To support the use of JAVA in an interactive environment, it is important to support the interpretation of small code fragments supplied by the user, which should begin execution as quickly as possible. It is also interesting to make possible the execution of code fragments that do not correspond to full JAVA programs, in order to facilitate the experimentation with specific language features and, thus, the development of prototypes.

Based on these aspects, this work presents a way to execute JAVA programs using an interpreter that receives code fragments and translates it to a graph representation. This execution graph is traversed, producing the program output.

Since the JAVA virtual machine[2] is not used[1], an interactive usage is possible, and the results of the commands entered by the user are preserved until the end of the session. The results obtained previously can be used later in other code fragments in the same user session.

[1]The JAVA virtual machine only works with full class descriptions, which hampers interactive usage.

2 Overview

The JEDI interpreter provides the user with a console where it is possible to enter code fragments or directives to the interpreter. The directives are interpreter internal commands prefixed by character “:“. They are used to perform specific interpreter tasks, like loading code fragments from an external file or terminating the interpreter session, among others. Code fragments, on the other hand, are not prefixed by special characters.

The assumption adopted by the JEDI interpreter for accepting "free" commands is all of them are contained inside a implicit `main` method. This guarantees that commands entered in different moments will refer to variables defined in the same scope. Different scopes are used only when explicitly created by the user, using block declarations. Class definitions, when entered in the interactive console, are an exception to this rule because they are always considered as top level classes and never as inner ones.

Appendix A presents an example session of an interactive use of JEDI.

Our current prototype uses dynamic type-checking. This was done to shorten the implementation time. A new version using static type-checking is under development.

JEDI was implemented using the HASKELL programming language[6, 3] and compiled using the GHC[7] compiler. The CTK[8] library was used as our initial framework. The current version of the prototype is available in `http://www.dcc.ufmg.br/~hermann/jedi`.

The execution graph is traversed by a specific function, which evaluates the contents of each node and produces the intended result. The execution model is described in the following sections, in which we assume a basic knowledge of HASKELL.

3 Execution graph node

Each node in the execution graph is constituted by three distinct components: a state transformer, responsible for modifications in the interpreter state; a continuation function, which implements the graph transitions from the current node; an exception handler, which implements transitions caused by the occurrence of exceptions. See Figure 3. Clearest areas identify node state transformers, intermediate areas identify node continuation functions and darkest areas identify node exception handlers. Node components are dropped from figures, for clarity, when they are not important in the relevant context.

State Transformer

The JEDI state transformer is a monadic function[5] responsible for changing the interpreter state according to the semantics of each command.

The state changes are implemented using a set of monadic functions, called state transformer primitives. These primitives are responsible for several features like declarations and updating of local variables, object instantiation, operations on the value stack used for the evaluation of expressions, checking for errors raised by primitives previously executed etc.

The state transformer function can return an ordinary value or an exception, passed to the node’s continuation or exception handler function, respectively.

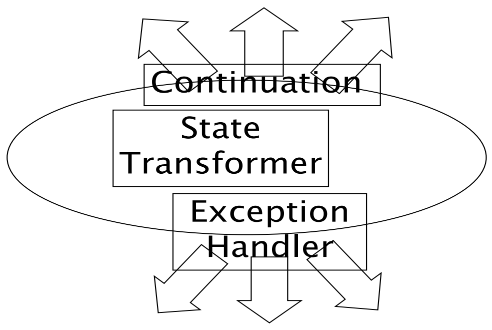

Figure 1: Structure of an execution graph node.

The specific monadic function of JEDI receives, implicitly, a state and returns a pair composed by a possibly modified copy of the the state and a value, which can be just an ordinary value or an exception. The state and the exception are left implicit, being handled internally in the monad.

It is responsibility of the monadic bind operator, denoted in HASKELL by >>=, to chain the execution of two monadic functions, passing to the second the state and result produced by the first function. This operator is responsible to ensure that, in any function binding f_1 >>= f_2, f_2 will not be applied to the state when f_1 returns an exception.

After an exception is raised by a primitive, it is propagated until it becomes the state transformer result. When this happens, we say the exception has reached the evaluation edge. After reaching the evaluation edge, an exception can only be handled by the node's exception handler.

Continuation

The continuation function receives an ordinary value resulting from the state transformer and returns the node that follows in the execution graph.

Continuations implement transitions between nodes in the execution graph. The separation of the roles of the state transformer and the continuation function facilitates the construction of nodes for each JAVA construct and allows the exchange of data between nodes without the need of using monadic functions.

Exception Handler

The exception handler is a function that receives an exception and returns the node that follows in the execution graph.

Any state transformer primitive can, potentially, raise an exception. The exceptions manipulated by a node's exception handler are those in the evaluation edge, returned by a state transformer.

An exception handler can accept or reject a given exception type. If the exception is rejected by a handler, the exception is re-thrown inside the state transformer of a node returned by this exception handler.

In the extreme point of this exception handler cascade, a predefined handler is used, that accepts all exception types and prints information about the uncaught exception. Other predefined exception handlers are responsible for throwing away the current activation record, when an exception is raised in a method call.

4 Graph execution

The function responsible for traversal of the execution graph (called the graph execution function) does not have any knowledge about the semantics of the interpreted language. When a node is presented to this function, that node's state transformer is evaluated. When the evaluation of a state transformer returns an ordinary value, the graph execution function calls that node's continuation function, passing the returned value as its argument.

The evaluation of the continuation function returns the node that follows and the execution graph function is recursively applied to it.

When the evaluation of the state transformer returns an exception, that node's exception handler is applied to the result. The evaluation of the exception handler returns the next node, where evaluation must proceed. The execution graph function is again applied recursively to this new node.

As we can see, the traversal function does not understand the semantics of the JAVA language but simply visits the right nodes of the graph representing the interpreted program.

The execution graph function is responsible for delivering the state implicitly returned by the last state transformer to the next node's state transformer. Using this approach the state "flows" through the visited nodes, suffering changes in the way.

When the graph execution function is applied to a final node, it returns the last state, returned by the last state transformer. This state will be given as the initial state in the graph execution corresponding to a next code fragment supplied by the user.

Explicit universal quantification, provided by the HASKELL GHC compiler as an extension to Haskell 98, plays an import role in the implementation of JEDI. Universal quantifiers allow the construction of nodes with different types in the same graph. Specifically, the use of universal quantifiers allow the construction of graphs where state transformers and continuation functions can return and receive, respectively, a value with a different type in each node (avoiding the creation of a new constructor for each kind of result yielded by the state transformer and used by the continuation function). The only condition to be satisfied is that the state transformer and the continuation function of each node must return and receive, respectively, a value of the same type.

5 Node examples

Figure 2 shows an example of a conditional branch node.

The node state transformer pops the topmost value of the evaluation stack and extracts a `boolean` value from it. This value is passed to the continuation function, which decides which node follows, based on this value.

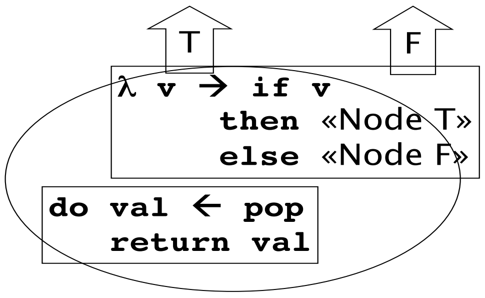

Figure 2: Conditional branch node

Nodes can be grouped to perform more complex tasks. Figure 3 illustrates the combination of nodes for the interpretation of the conditional expression `true ? 1 : 2`.

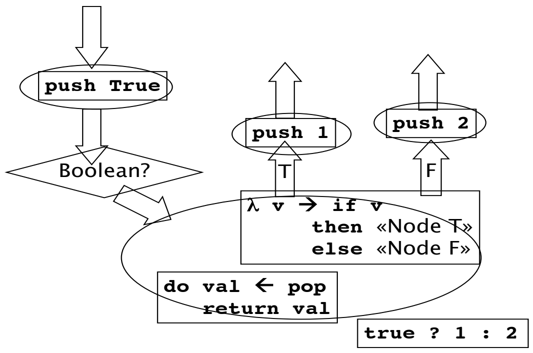

Figure 3: Node array used to evaluate the expression `true ? 1 : 2`

6 Error Detection Nodes

Error detection nodes are one of the cornerstones of the dynamic error-checking model implemented in the current prototype. They are inserted at specific points in the graph to cancel the execution of the program, when errors are detected, in a graceful manner.

The function used to traverse the execution graph does not know about the existence of errors, nor how to identify them.

To allow this transparency, error detection nodes are inserted at specific points of the graph, during graph construction. When an error detection node is evaluated, it checks whether any error has been raised, by a state transformer primitive evaluated in any previous node. The detection of any context-sensitive error in the input program causes a branch of the execution to a final node.

Figure 4 shows a generic error detection node.

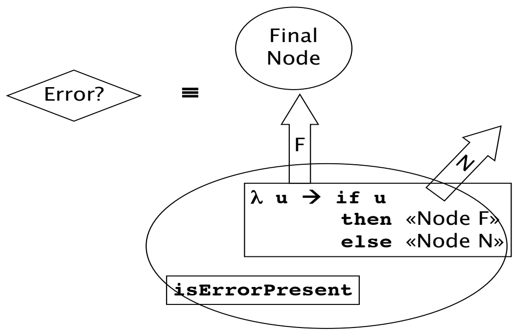

Figure 4: Generic error detection node.

Type error checking is so frequent that there exists a special node definition for its implementation. In this node, it is analyzed whether the result pushed onto the value stack by an expression evaluation has the expected type. Each node for an expression has an expected expression type.

Figure 5 shows a type error detection node.

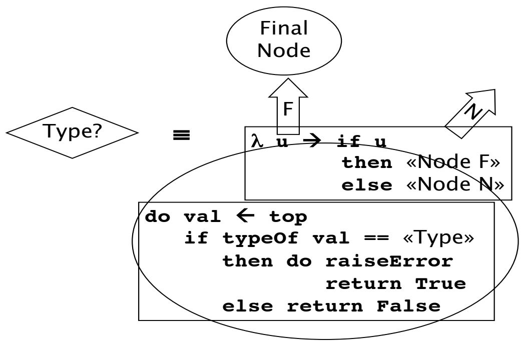

Figure 5: Type error detection node.

Figure 3 illustrates an expression evaluation, using a type error detection node to type check a `boolean` expression.

7 Expression evaluation

The evaluation of an expression is done by translating each term of this expression into a set of nodes, representing it in polish reverse notation. The operation arguments are evaluated and the result is pushed into the value stack of the interpreter state. The operation node pops its arguments from the stack, performs the operation and pushes back the result.

Figure 3 shows a simplified example of expression evaluation.

The evaluation of an expression involves some stages that are not present in Figure 3. Before each operation the interpreter needs to check if the arguments have the expected type. The JAVA language defines a set of rules to determine the situations when an explicit conversion (using the cast operator) is required, and when the arguments of an operation do not have compatible types.

Widening conversions (also called promotions) can be left implicit. Narrowing conversions, on the other hand, always require explicit casts from the user, as described by the JAVA language specification. Other conversions include identity conversions, string conversions and forbidden conversions. This last kind always generates an error, detected by the error detection nodes, and cancels the graph execution.

Basically, the JEDI interpreter pops the operands and figures out which conversion is necessary, in the given operation. If a widening conversion, identity conversion or string conversion is required, the conversion is applied silently.

The expressions involving method calls are checked for type-correction similarly to other expressions.

Like any other construction, expressions can raise exceptions during the execution. This can be done, for example, by a method call, a division by zero or accessing field/methods of a null reference.

8 Name decoding

Decoding of identifiers is a complex stage in the interpretation of code fragments. The JAVA language specification describes a set of rules that must be followed to determinate the meaning of a given (qualified or unqualified) name. A name like `x.y.z.k` can represent a field or method of a class, a field or method of an instance, a class, a class package or a local (method) variable.

Figure 6 shows an example of name decoding.

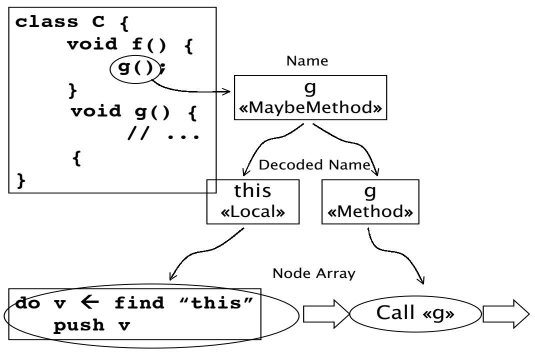

Figure 6: Method name decoding.

The JEDI interpreter implements the rules described in the specification of the JAVA language, as follows:

1. The name is pre-qualified, according to the program context in which it is used. This context classifies it into an expression, method, type or package. Other name components, in qualified names, are classified based on this initial pre-qualification. This scheme is often not sufficient for a unique classification.

2. The pre-qualified name is decoded using the rules described in the JAVA language specification. This decoding produces a more complete name classification, where each name component is classified as a class (packages and classes are grouped together), a field, a method, a local variable or none. In this stage the implicit reference to the local variable `this` becomes explicit. If some of the name components do not correspond to entities in the interpreter state, the whole name is classified as invalid.

3. If the decoded name is valid, a chain of nodes is built. This node chain is responsible for the evaluation of each name component, producing the intended result. The nodes in this chain depend on the classification of each name component, the program context in which the name is used, and whether a field is a class or an instance field. Reading and writing a variable — field or local — is represented by different node chains.

9 Method definition

Methods are represented in two ways. The first contains the method signature, its arguments and modifiers[2]. This representation is used to determine which method is called and whether its use is valid in the context of the call.

The second representation is a component of the execution graph, which is visited when the method is called, following the conventions below, in the given order:

- Each parameter is bound to the corresponding value, passed in the value stack.

- A new scope is opened.

- The activation record is popped from the stack, after obtaining the return point from this record.

  To detect the absence of a `return` statement during the execution of a method call (see section 10), a special node is inserted as the last node of the method body (which will be visited if no `return` statement, present in the method, is visited).

10 Method call

The implementation of method calls starts by name decoding, as described in section 8), of the method name and object expression or class name.

The nodes built after the decoding stage describe the method to be called. The validity of each node is tested (for example, it is tested whether the node representing the target object is not `null`). This

[2]Modifiers are qualifiers like `public`, `static` and `final`, among others.

description contains a reference to the first node in the method definition. This reference is passed to the continuation function and used by the graph execution function as the next node to be visited in the graph. A new activation record is pushed into the method call stack.

In method exit points, a special node is inserted to obtain the method call return point from the current activation record, which is then popped from the method call stack.

The same procedure is used by methods returning no value (a method whose return type is `void`), i.e. a node is inserted to force the return from the current method.

This makes optional an explicit `return` statement in the method source code. In methods where the return type is different from `void`, this same node is used to detect whether an explicit return of a value is missing and informs the user of this mistake by means of the message "*Reached end of non 'void' method*".

Figure 7 shows the node used to execute method calls.

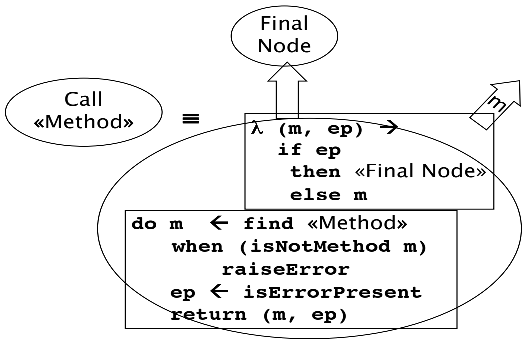

Figure 7: Generic method call node.

11 Conclusions

This work presents an interactive interpreter for JAVA, JEDI, which uses an internal code representation constituted by a graph, where every node is composed by three functions: a state transformer, a continuation and an exception handler.

JEDI supports the use of JAVA in an interactive environment, allowing a swifter interpretation of small code fragments supplied by the user, that need not correspond to full JAVA programs. This facilitates the experimentation with specific language features and, thus, the development of prototypes.

The error-detection nodes impose a significant overhead due to an excessive number of tests performed during graph execution. For example, in the execution of iterative statements (`while`, `do-while` and `for`), the type of the result yielded by the conditional expression is checked once for each iteration. A statically checked version, which is under development, will overcome this problem. In the static version, only a few error detection nodes are required for the implementation of all operations.

References

[1] GOSLING, James; JOY, Bill; STEELE, Guy; BRACHA, Gilad. JAVA *Language Specification*. Second Edition. Sun Microsystem, 2000. http://java.sun.com/docs/books/jls

[2] Lindholm, Yellin. JAVA *Virtual Machine Specification*. Sun Microsystem, 2000. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[3] THOMPSON, Simon. *The Craft of Functional Programming*. Addison-Wesley, 1999.

[4] APPEL, Andrew W. *Modern Compiler Implementation in ML: Basic Tecniques*. Cambridge University Press, 1997.

[5] WADLER, Philip. *Monads for functional programming*. Advanced Functional Programming, Springer Verlag, LNCS 925, 1995. http://cm.bell-labs.com/cm/cs/who/wadler/topics/monads.html#marktoberdorf

[6] HASKELL Programming Language. http://www.haskell.org.

[7] The GHC Team. *Glasgow* HASKELL *Compiler*. http://www.haskell.org/ghc

[8] CHAKRAVARTY, Manuel M. T. *CTK Compiler Tookit*. http://www.cse.unsw.edu.au/~chak/haskell/ctk.

A Examples

An example of interactive session in the JEDI interpreter is provided. The session shows an evaluation using the small Java class defined in the file "Counter.jes" and listed below:

```
package counter;
class Counter {
     long j;
     void setCounter(long j) { this.j = j; }
     long count() { return j = j + 1; }
     long getCounter() { return j; }
}
```

In the example session below, the file "Counter.jes" is loaded and the class `Counter` is used in a code fragment entered by the user from the console.

```
% jedi
Welcome to console of Jedi 0.5.2 (build June 21 2001)
This software is distributed under the terms
of the GNU Public Licence.
NO WARRANTY WHATSOEVER IS PROVIDED.
Jedi> :load script “Counter.jes”
Completed after 0.000 sec.
Jedi counter> \
counter> counter.Counter c = new counter.Counter();
counter> while (c.getCounter() < 1000)
counter>     Output.println("The counter is " + c.count());
counter> Output.println("Done");
counter> .
The counter is 1
The counter is 2
The counter is 3
...
The counter is 999
The counter is 1000
Done
Completed after 1.270 sec.
Jedi counter> c.getCounter();
1000
Completed after 0.000 sec.
Jedi counter> c.count();
1001
Jedi counter> :quit
%
```