Portable and modular exceptions in Neverlang2

Andrea Francesco Iuorio
Id. Number: 774151

Università degli Studi di Milano
BSc in Computer Science
Advisor: Prof. Walter Cazzola

UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

Dipartimento di Scienze e Tecnologie
Academic Year 2013/2014

Contents

1 Neverlang2 and exceptions

1.1 Introduction

Compilers and interpreters are one of the first types of software ever researched because the possibility to comunicate with a machine using a more human-like language made application development easier. We can trace the origin of compilers in a period where software engineering didn't exist: compilers' structure was often monolithic, with fixed formal grammars and development techniques that impeded to easily make changes to a language. The modern compilers aren't so different from their predecessors because languages are something static, with improvements in the arc of years or even decades. However, many languages have common features and syntaxes, but, because of the compiler's structure, developers often must rewrite the same feature for every language: this makes languages' development complex and expensive.

Today is useful having languages that can rapidly evolve and change. More and more people have the necessity to comunicate with a computer, and these people aren't computer experts: they need a way to utilise a computer for their work without learning something as complex as a general-purpose programming language. This leads to the creation of special languages, called Domain Specific Language (DSL), that can be easily used by an expert in one domain for doing their work on a computer. The creation of DSLs requires a more open approach in the creation of the language, since this kind of languages must be able to quickly evolve in order to meet the needs of the domain experts[3].

A language can be defined by a formal grammar. This grammar contains a set of productive rules[1]. This set can be partitioned, separating for example the rules for conditional jumps from the rules for loops. A programming language so can be intuitively represented as a set of little blocks, each one implementing a specific feature of the language. The language can be seen as the union of independent features[6].

Neverlang2 is a tool created by AdaptLab in order to make DSLs development simple and modular. It concentrates on the creation of little parts of the language representing a feature instead of complete languages: Neverlang2 views a language as a set of blocks, called modules, each one representing a specific feature. Since we are creating features instead of monolithic languages, these are independent from one another and the language can be created modularly, adding and removing modules[7]. One intuitive example of our approach can be seen in conditional jumps: in many languages this feature is implemented as a code block preceded by an *if* keyword. The monolithic structure of the compilers often prevents to reuse the code that implements the conditional jumps from one compiler into another, even if their syntax and semantic are the same. In Neverlang2 the *if* feature is a module that can be reused into another language. However, there are features that are difficult to separate from the rest of the language, or features that require the knowledge of the rest of the language for their implementation. This is a problem for the modularity Neverlang2 try to proposes. One good example of a feature that depends from the rest of the language and the execution environment is exception handling, since its relative module depends from the rest of that language. An exception can be triggered at any moment: the entire language must understand them.

This thesis' objective was the creation of an exception handling mechanism which is both architecture-independent and language-independent, permitting Neverlang2 to allows the reuse of modules that implements exceptions. In order to accomplish this goal we have defined and implemented two software: a code generation library for Neverlang2 and a runtime library. The code generation library is what permits the language independence of this solution: Neverlang2 modules use this library, and they can be reused in other languages. The runtime layer is what permits the architecture independence: the language use this layer for exception handling instead of the mechanism offered from the target host. For the sake of testing the validity of our project we implemented these two libraries for the Java Virtual Machine. We also created a compiler for an object-oriented language targeting the Java Virtual Machine that uses both the code generation library and the runtime library for implementing different types of exceptions, showing the power of our solution.

1.2 Introduction to Neverlang2

Neverlang2 is a framework created to make DSLs development easy and modular. The core principle of Neverlang2, as we told earlier, is to move the focus on what define a language: instead of seeing a language as a fixed formal grammar, Neverlang2 sees it as a set of blocks, each one representing a precise feature of the language. For example, a c-like language could be seen as a set containing an *if* component which implements conditional jumps, a *while* component that implements loops and so on. The advantage of this representation is clear: many languages possess common features, and Neverlang2 permits to reuse features written for one language into others[6]. Besides, extending a language is very easy: all we have to do is to add the module in

the language set.

A language feature is implemented in Neverlang2 as a *module*, which is composed by two different lists: a list of production rules and a list of semantic actions. A module represents the nodes of the Abstract Syntax Tree (AST)[1] that implements the feature. Neverlang2 therefore creates the AST joining all the nodes represented by the modules using their syntax rules. For example, this could be a module which implements an *if* conditional jump:

```
module IfStatement{
    reference syntax{
        Statement <- "if" "(" BoolExp ")" Statement;
        Statement <- "if" "(" BoolExp ")" Statement "else" Statement;
    }
    role(evaluation){
        0@{
            eval $1;
            if($1.value == true)
                eval $2;
        }.
        3@{
            eval $4;
            if($4.value == true)
                eval $5;
            else
                eval $6;
        }.
    }
}
```

role(syntax) represents the production rules used by Neverlang2 to parse the input, while *role(evaluation)* possess an action for every rule and represents the semantic action the compiler must execute when it passes into that node. We can notice that this module doesn't make any assumption about the language: a developer can reuse this module into another language for implementing this kind of *if* conditional jumps.

Neverlang2 views a language as a set of modules. This is implemented by a simple file, the *Language Descriptor*[7], in which there is a list of the modules that compose the language: this way, a developer can easily add a new feature in the language by inserting the new module in the Language Descriptor.

1.3 Introduction to exceptions

Exceptions are anomalies that occur during the execution that require to be treated in a special way, often separated from the application's execution flow[2]. Historically, we can define two types of exceptions: hardware exceptions when the event is physical, like a processor fault, and software exception when the event occurs during the execution of particular instructions that could ruin the computation. A classic example of software

exception can be seen in the following code:

```
int a;
read a;
int b = 10/a
```

This code has a problem when the variable *a* is equal to zero, because the processor would execute a division by zero, which is an undefined operation. This is a problem for the computation and this special case must be handled correctly. This event should also be handled separated from the program in order to prevent any possible modification or contamination of the program's execution flow. With *exception handling* we intend the mechanism that permits to stop the application's execution, handles the event in an isolate way and, if possible, continues the execution of the application as if it wasn't never interrupted. Languages and hardware architectures handle exceptions in different ways but we can see three common principles:

- *Protected code*: Protected code is a special area of the program where the exception mechanism should be ready to operate if necessary. An exception can be risen at any moment during execution but it may be a good idea for debugging purpose to have well defined areas of code that can raise an exception. For example in programming language like Java or C++ the protected code are statement blocks preceded by the *try* keyword.
- *Handler code*: Handler code defines the code that must be executed for handling an exception. Sometimes they are implicit in the language runtime or in the operating system but many languages allow the programmer to define handlers for specific exceptions. For example, in the Java programming language handlers can be defined by a statement block preceded by the *catch* keyword.
- *Throw code*: Throw code represents the special event and ask the exception mechanism to handle the problem in the correct way. This is the code that raises the signal we called exception. Many languages use the *throw* keyword for raising an exception.

Throwing an exception is a complex operation for the machine because it has to interrupt the execution flow, execute the correct handler separately from the rest of the application and, if possibile, return to the normal flow as if the exception wasn't raised. We can describe the throwing of an exception as a three-point process:

- *Exception rising*: an exception is raised by the throw code. The system has to stop the application in a way that can be resumed, identify the exception threw and choose the correct handler to handle it.
- *Exception carrying*: the exception has to travel across the system and arrive to the correct handler without touching the program to prevent any contamination. This is the major issue for our modularity: even if a feature isn't directly interested in managing exceptions we must be able to revert its actions or travels in it at any moment.
- *Exception handling*: the correct handler is executed and the system returns to the normal execution flow if possibile.

We can see an exception handler as a routine that must intercept every exception launched, chooses and calls the correct handler in an isolated way and possibly resumes the program. As one can imagine, all this operations require some sort of hardware support. Also, every language implements this procedure in its own way. This could make the creation of some type of exception handling mechanism on some architecture difficult or even impossibile. For example, the Java Virtual Machine exception mechanism impedes to resume the execution flow in the instant after the exception rising: instead it only permit to restart the execution after the protected code[5]. Besides, since an exception can be raised at any moment, the entire language has to know about it, making impossibile to maintain the modularity Neverlang2 promises. The goal of this thesis is the creation of an abstract layer that permits to implement any type of exception handling on any architecture. This way a Neverlang2 module that implements an exception system can be really portable and independent from the rest of the language and the hardware.

2
Neverlang2 Exception Library

Neverlang2 is a framework for language development that wants to change how a language is defined: not a fixed formal grammar but instead as a set of independent features. However, some features are difficult to create independently from the others already present in the language. Exception handling is a good example of this behaviour: an exception can be raised at any moment, so we need some sort of support from the execution environment or the entire language, since the exception can travel in the system before it'll be handled and it can require to undo some semantic actions. We want to develop a solution that permits the developers to see the exception handling as an independent feature. The Neverlang2 Exception Library (NEL) is our solution to this problem.

One solution to our problem is using some sort of runtime support: this way we have a standard architecture, independent from the execution environment, that the language can use for provide exception handling. However, we want that our solution is compatible with every Neverlang2 module: we don't want to change anything in them for support our solution. To be really language-independent, we want a solution in which we only need to add the "exception handling" module to a language to adds exception support: this means we need some sort of support during compile-time too. We have composed NEL as the union of two libraries: a code generation library called *Neverlang2 Exception Code Generation Library (NECG)*, and a runtime layer called *Neverlang2 Exception Runtime Library (NERL)*. Neverlang2 modules use NECG in their semantic actions for generating commands for NERL. NERL is applied as an intermediate layer between the application and the execution environment, and intercepts these commands to catch and handle the exception.

We can also notice that the use of NECG isn't required: the application can send commands directly to NERL, allowing its use in interpreted languages. In this chapter we show how NERL and NECG are defined.

2.1 NERL - Neverlang2 Exception Runtime Library

The Neverlang2 Exception Runtime Library is the runtime support used by our solution. We can think of NERL as an intermediate layer between the application and the real execution environment: when a program needs to raise an exception, NERL intercepts the call and handles it. This way the language's exception handling can be created without thinking about the rest of the language nor the other features of the language must understand our exceptions. This also permits the creation of exception mechanisms that would be normally impossible on a specific hardware architecture. In this section we present an abstract representation of this layer and how it works.

2.1.1 NERL architecture

NERL internally is composed by three data structures: a call stack, an Exception Table and a Secure Area.

In order to handle an exception we must have a way to control the machine's state. In a typical Von-Neumann architecture, this means we have to control its memory: if we can do it, we can halt, save, restore or modify the state of the machine in any moment. Since every machine handles memory in different ways, we needed a standard memory layout. NERL internally sees the machine's memory as a stack: every function in the program is an element of this stack called *stack frame*, and the function actually in execution is at the top of this stack. We can define the stack frame structure in this way:

```
struct NERLstackFrame{
    int methodID;
    void *symbolTable;
};
```

methodID is a simple identification number for that function. This is used by NERL for discerning between functions during the exception carrying.
symbolTable is a data structure that maintain a copy of the variables of the function. This is necessary for restoring the call stack after an exception.

It is important to notice that this stack may not be the machine's real call stack. It could be mapped on other data structures, depending on how the machine handle its memory. The NERL stack is just a way we internally use for presenting to the developers a standard memory layout. This also means that we can use it in architectures that don't present a stack memory or that don't allow the modification of the stack pointer. In order to simplify the context switch and to maintain the NERL stack coherence with the machine's memory, NERL should provide a read/write barrier for each memory access. This isn't however required since an implementation could directly use the NERL stack as the application's stack.

Other than the stack, there is another essential data structure in NERL: the *Exception Table*. This is a look-up table containing pointers to handlers' instructions. When an application enters in the protected zone, it executes a procedure called *Handler Registration*, which saves in the Exception Table where the handler for a specific type of exception can be found. When NERL receive an exception, it calls the handler saved

in the Exception Table for that specific type of exception. It's important to notice that with type of an exception we don't mean a language's type but just a descriptor of that exception. An exception's type in NERL can be everything that permits to understand or discern the nature of the exception.

Finally, NERL possesses a Secure Area. This is a buffer of reserved memory where is possible to save informations that should be maintained and quickly retrieved during the handling of an exception. The throwing of an exception for NERL is just a signal sent from the program, and is useful to have the possibility to exchange data from the program to the handler.

2.1.2 NERL intermediate language

For maintaining as much modularity and independence as possible, NERL internally uses a small intermediate language. An application sends these instructions to NERL and our runtime library executes the exception. Putting this instructions inside the program may seems problematic for the modularity we want to provide but, thanks to Neverlang2, the AST nodes that generate these instructions can be easily added or removed from the language.

The stack data structure hasn't any special operations. We have the classic push and pop operations for adding or removing stack frames. There also must be a way to update the symbolTable data structure, which is implementation-defined.

The Exception Table has four instructions:

- *initNERL* initialises NERL and its data structures, requiring an handler policy function. This handler policy is the heart of NERL: it's a function executed after the raising of an exception that should utilise the NERL data structures to effectively call an handler. The ability to change the handler policy permits to generate different exception handler mechanisms for different languages. A NERL implementation can include a default handler policy.
- *registerHandler* starts the Handler Registration policy we described before. We indicate where we can find the handler and the type of exception that must be handled by it. With "type" of an exception, we don't intend the concept of type from a typical programming language, but a description that identifies the nature of the exception.
- *raiseException* is the instruction that throws an exception. It requires a description that represent the type of the exception thrown.
- *deleteHandler* removes the handler registered for a specific exception type, which can be used for example when the program exits the protected code.

Lastly, the Secure Area has a read and a write instructions.

2.1.3 NERL algorithm

Now that we have explained the internal structure of NERL, we can show how an exception is handled by our layer.

- The application installs the handler policy function.
- The application enters in the protected area and executes registerHandler
- The application executes the raiseException command when an exception needs to be raised.
- NERL executes the handler policy function, which uses the NERL internal data structures for choosing which handler to use and how this handler should be execute.
- The handler is executed and returns to NERL.
- NERL executes the second part of the handler policy and returns to the application.
- When the application exits the secure area, executes deleteHandler.

We can notice that the real exception handler mechanism is the handler policy function: the developer utilises the NERL data structures for deciding how the machine should operate when an exception is raised.

2.2 NECG - Neverlang2 Exception Code Generation

NERL can be used by sending commands directly to it, making it perfect for an interpreted language. However, Neverlang2 is also a compiler-making tool, so we must include NERL instructions into the program's instructions. In order to simplify this process, NEL provides a code generation library called Neverlang2 Exception Code Generation Library (NECG) for generating the NEL instructions during the compilation of a program. A Neverlang2 module can use this library during the semantic phase. This is a great advantage because we avoid to insert NERL-specific code in Neverlang2 modules: we can reuse these modules on every architecture and language.

NECG has a simple structure since most of the work is done by the runtime layer. We analyse the internal structure of NECG and how the Neverlang2 modules should utilise it.

2.2.1 NECG architecture

Since how NERL receives its instructions is implementation-defined, NECG abstract structure is quite simple as it just has to add the instructions for sending these commands during the AST evaluation. NECG views every code block, like an handler or a protected area, as a "black box" object, ignoring the instructions in it. During the semantic phase, for every block, NECG creates a box that represents it, receives the native code from the Neverlang2 modules and adds all the instructions that are necessary for using NERL, converting if necessary the statements used by the language to instructions for sending NERL's command. For example, in our implementation, at the start of every protected area NECG adds the code for sending the handler registration command to NERL. This way we can avoid to insert any specific NERL code in the

compiler's code, permitting us to reuse the Neverlang2 modules in other languages and architectures.

3 Neverlang2 Exception Library - JVM implementation

In this chapter we present our implementation for the Java Virtual Machine of the Neverlang2 Exception Library described previously. We selected the Java Virtual Machine as the target of our implementation because it impedes the manual manipulation of the call stack[5] and, regarding the exception handling, it forces the programmers to use its internal structure. Implement NEL on the JVM without using any special features is a good way for show the power of our solution.

This chapter is divided in two main sections, in which we analyse the implementation of NERL and NECG respectively.

3.1 NERL implementation

The Neverlang2 Exception Runtime Layer is a runtime library that must be applied to the native environment. For our implementation, we opted for a JVM package called *NERL* that must be inserted into the JVM CLASSPATH variable[5]. This way our application can easily access NERL data structures and commands.

NERL package is formed by 5 classes:

- *$PLClassEX* is used for the handler implementation. It's an interface that must be implemented by every JVM objects for catching the exception.
- *ExceptionRuntime* contains the implementation for NERL instructions, the Exception Table and the Secure Area. The application communicates directly with this class for raising and handling an exception.
- *StackTrace* is the NERL call stack implementation. Since the stack was the critical point regarding its implementation, we separate it from the rest of the NERL

implementation.

- *MethodRuntime* represents a NERL stack frame. NERL creates a MethodRuntime for every method used by the application.
- *SymbolTable* is used as a gateway between the application and NERL for accessing memory.

3.1.1 NERL data structure

```
package NERL;
import java.util.*;
public class ExceptionRuntime{
        private static ExceptionRuntime ex;
        private HashMap <String, Integer > handlersID;
        private HashMap <String, $PLClassEX > handlersOB;
        private Object secureArea;
        private ExceptionRuntime()
        public static ExceptionRuntime getInstance()
        /*Code for NERL Intermediate Language*/
}
```

The core of NERL is implemented in the ExceptionRuntime class. It's a singleton[4] class containing the implementation of all the data structures and the NERL intermediate instructions.

Since the Java Virtual Machine doesn't permit unconditional jumps between different methods, we had to find a way for calling our handlers without knowing, at compile-time, their position. In order to resolve this problem we associate an unique identification number to every handler in the class even if they are associated with different methods or protected block. Then, all the handlers' code are united into a single method called *$handler*. This is the implementation of $PLClassEX:

```
package NERL;
public interface $PLClassEX{
        public void $handler(int id);
}
```

The $PLCLassEX interface requires this method so we can call it using the dynamic binding without knowing its position during compile time. NERL receives the handler's identification number during the handler registration and saves it in the Exception Table so that it can jump to the correct handler, ignoring the others. The Exception Table is formed by two HashMap objects: one for maintaining the relation between an exception type (identified by a String JVM object) and its handler and the other for maintaining a copy of the object's handler: this is necessary because when we call the handler, we need the object for the dynamic binding.

Finally, the Secure Area is simply an Object field: this way the application can save an object and transmit it to the handler.

3.1.2 NERL Stack implementation

NERL stack is implemented in the StackTrace class and contains the stack frames represented by the MethodRuntime class. Since the Java Virtual Machine doesn't permit to manually manipulate the call stack, this was the most difficult thing to implement in our NERL implementation. StackTrace is a stack of MethodRuntime objects, each one representing an object's method. Its prototype is:

```
package NERL;
import java.util.*;
public class MethodRuntime{
        public String methodSignature;
        public HashMap<String,Object> scopes;
        public MethodRuntime(String methodSig);
        public void addVariable( Object o, String s);
        public Object getVariable(String s);
}
```

methodSignature represents the JVM signature of the method[5], while scopes is the hashmap representing the symbol table of the method. When the application needs to access a variable, it doesn't use the JVM stack but its MethodRuntime: in this way NERL stack represents the memory of the machine. Since we are using NERL stack but the machine is using its own stack, we need a way to reflect the changes between the NERL call stack and JVM call stack, so that a change in the NERL stack can be reflected on the real execution stack.

The push operation is a method call. When a method is called, the called function puts its MethodRuntime object on the NERL stack during the prologue phase. However we also want the ability to remove a method anytime, since we don't know when an exception is raised and how much we would have to travel back in the call stack. Our solution is the creation of a call barrier. Every method, during the prologue, asks NERL an unique identification number. Then, it saves this number in a variable called $IDMETHOD and in a JVM local. Every time we call a function, we surround the call with a little "barrier". When the called function returns, we check the MethodRuntime at the head of the NERL stack and we get its $IDMETHOD. If it isn't the same as the caller method's ID we have in the JVM local, the caller method quickly returns since this means that the method was removed from the JVM stack. We call this type of return a spurious return, because the returned value has no means: we have removed the method from the call stack before its completion. Since the caller was called from some other method, our return action will be intercepted by another call barrier and so we effectively unwind the JVM stack. In this way, the removal of a MethodRuntime is mapped on the JVM call stack.

3.1.3 NERL intermediate language implementation

The NERL intermediate language is implemented as a list of methods, one for every instruction. This way the application can call the method for execute a specific NERL command. This is a prototype of intermediate language methods:

```
package NERL;
public class ExceptionRuntime{
        public void installHandler(String type, int id, $PLClassEX o);
        public void removeHandler(String type);
        public void raiseException(String type);
}
```

The implementation of every command is simple:

- *initNERL* doesn't exist: since ExceptionRuntime is a singleton class, it's automatically instanced the first time it's used. Our implementation only provides a default handler policy.
- *installHandler* receives an handler with its ID in the $handler method and a reference of the object in which they reside and adds all these information into the ExceptionTable. removeHandler works in the same way, except it removes the handler.
- *raiseException* contains the default handler policy: it receives the exception types and, using the data in the Exception Table, manipulates the NERL stack in order to modify the machine's state, and calls the $handler method with the selected handler's identification number.

3.1.4 NERL handler policy

Our implementation of NERL provides a default handler policy. While this isn't required, we avoid to provide a way for selecting different handler policies in this implementation for the sake of simplicity. This default policy is implemented as follows:

```
public void callException(String typeException){
        for(MethodRuntime m : NERLStack){
                if(m.getVariable("$TRY_METHOD") != null )
                        break;
                StackTrace.getInstance().popMethod();
        }
        Integer i = handlersID.get(nameException);
        $PLClassEX o = handlersOB.get(nameException);
        o.handler(i.intValue() );
        StackTrace.getInstance().popMethod();
}
```

This works like the handler policy used by the Java programming language: we unwind the call stack until we have reached a try block, then we select the handler from the Exception table and execute it. After that, we remove the try block from the call stack

so that the computation will be resumed after the try block.

3.2 NECG implementation

As we saw, NERL sometimes requires to add some instructions in the bytecode generated for the program. We have to use NECG for adding this bytecode while avoiding to generate it in the Neverlang2 modules. In our implementation, *NECG* is a simple JVM package used by the compiler. It contains:

- *ExceptionSupport*, which represents the center of NECG. It handles all the other data structures in the package for generating the final bytecode.
- *ExSecureCode* is the class representing a protected block. When the compiler finds a protected block, it creates an ExSecureCode object. Internally, ExSecureCode adds all the bytecode necessary for the Handler Registration procedure.
- *ExHandlerCode* is the class representing an handler. Internally it generates all the instructions necessary to the $handler method.
- *MemoryHandler* is a simple collections of bytecode instructions for all the memory operations necessary for NERL. It doesn't execute any real work in our solution.

3.2.1 Handler block

ExHandlerCode represents an handler block and, starting from the bytecode that represents the handler's execution code, it generates all the necessary instructions for using this specific handler in the *$handler* method. Every handler used in the class must be inserted in the *$handler* method, so we needed a way to jump to the correct handler. Our solution takes the form of a simple *tableswitch* instruction[5]: every handler has its own number and this number is saved in the NERL Exception Table during the handler registration. With this procedure we can retrive a specific handler without problems.

3.2.2 Protected block

ExSecureCode represents a protected block. It needs to receive all the ExHandlerCode objects representing the handlers associated with the protected area in order to internally generate the handler registration and handler deletion bytecode. It also generate the method that represent our protected code block. Then, using these informations, the compiler can generate a separate method for the protected block and the correct calling instruction.

For the handler registration, ExSecureCode generates a string formed by "typeException,ID", for every handler, with the id in the $handler method. Then, this string is passed to NERL using the installHandler method. For example, the handler registration in the bytecode for this simple block is:

```
try{
        //Try code
}
catch(Integer i){
        //Handler code
}
catch(Exception e){
        //Handler code
}
```

```
invokestatic NERL/ExceptionRuntime/getInstance()LNERL/ExceptionRuntime;
ldc "Integer,2,Exception,3"
aload 0
invokevirtual NERL/ExceptionRuntime/installHandlerList
              (Ljava/lang/String;LNERL/$PLClassEX;)V
```

Ultimately, ExceptionSupport is a class that contains all the ExSecureCode generated during compilation and also generates all the necessary bytecode for using the protected block methods and the general handler method. All this bytecode is appended to the class the compiler is evaluating.

4
Case study: PanzyLang

PanzyLang is an object-oriented, compiled language we have developed using Neverlang2 to show the potentiality of our exception library. It's implemented by a compiler that generates JVM mnemonic bytecode. This bytecode is then converted in native JVM classes using Jasmin (http://jasmin.sourceforge.net) as bytecode assembler.

We created three versions of this language: a base version and two versions extending it, adding the support for different types of exceptions. We added to the base language a try-throw-catch exception policy similar to the one used by the Java programming language [5] and then we extended it with the *retry* keyword. We chose these versions for show that our solution permits us to add the support for exceptions upon a language that doesn't support them. In this chapter we present how our language and its extensions are implemented.

4.1 Base language

The base language is a Java-like programming language. We don't spend too much time describing its features but instead we only present a list of them.
PanzyLang supports:

- integer, double floating point, string and reference types.
- integer and double floating point arithmetics
- boolean operators
- if-else conditional jumps
- while loops
- functions
- classes

- new keyword for creating objects
- cast between objects

Each one of these features is implemented in a single Neverlang2 module that can be removed or reused in other languages.

4.2 Try-Catch-Throw language

Starting from the base language, we added the support to exceptions with just two Neverlang2 modules, one for the throw statement and one for the try-catch statement blocks.

4.2.1 Throw module

```
module panzyLang.Ex.Throw{
imports{base.*;NECG.*;}
        reference syntax {
                Statement <- "throw" Expr ;
        }
        role(evaluation) {
        0@{
                MethodInformation m = CodeUtility.currentMethod;
                String buffer = m.generateQuickReturn();
                int numArg = m.getNumArgs();
                ExprInformation i = $1.ExprInfo;
                $0.Text =
                (String)$1.Text +
                i.convertToObject +
                PanzyLangMain.exSupport.throwExceptionCode(i.javaSigType)+
                PanzyLang.exSupport.getCallBarrierBytecode(buffer, numArg);
        }.
        }
}
```

Our throw syntax is a statement with the keyword *throw* and an expression. This expression is the object, eventually converted, that will be saved in the NERL's Secure Area and represents the arguments that will be passed to the selected handler. The throw semantic node obtains the expression's bytecode and its type, convert it to an object for saving it in the NERL Secure Area and generates the bytecode for the throw instruction. The throw instruction is a method call to the raiseException method in NERL. Finally we surround our call with a call barrier for the stack manipulation.

4.2.2 Try-Catch module

```
module panzyLang.Ex.TryCatchJava{
imports{base.*; java.util.*; NECG.*;}
        reference syntax {
                Statement <- "try" "{" StatementList "}" CatchList;
                CatchStatement <- CatchOpen "{" StatementList "}" ;
                CatchOpen <- "catch" "(" Type Identifier ")" ;
                //CatchList rules
        }
        role(evaluation) {
        0.{
                ExSecureCode ex = new ExSecureCode(PanzyLangMain.exSupport);
                MethodInformation back = CodeUtility.currentMethod;
                CodeUtility.currentMethod = new MethodInformation
                                            (ex.returnTrySignature(),"V",false);
                eval $1;
                eval $2;
                for(ExHandlerCode h: $2.catchList)
                        ex.newHandler(h);
                String invocationCode;
                MethodInformation currentMethod = CodeUtility.currentMethod;
                String t = ex.createTryCode((String)$1.Text);
                currentMethod.appendMethodCode(t);
                invocationCode = currentMethod.getInvocationCode(back);
                tryMethodList += currentmethod.getSourceCode();
                CodeUtility.currentMethod = back;
                $0.Text = invocationCode();
        }.
        3@{
                ExHandlerCode ex = $4.handlerObj;
                ex.appendSourceCode((String)$5.Text);
                $3.handlerObj = ex;
        }.
        6@{
                ExprInformation t = $7.ExprInfo;
                $6.handlerObj = new ExHandlerCode
                                (t.javaSigType,(String)$8.idText);
        }.
        //Catch list implementation
        }
}
```

The catch block is a StatementList, the list of statements in our code block, and, thanks to our code generation library, we can avoid all the problems for handling the catches' different natures and types. For every catch block, we generate a specific ExHandlerCode object, evaluate its StatementList and then we put the code we have generated in the ExHandlerCode object. The CatchList generates an ArrayList of all the

ExHandlerCode.

The try implementation is more complex: we consider it a new JVM method so we have to stop the evaluation of the actual method, create a new method and then evaluate its StatementList and CatchList. We obtain the bytecode of the try block and the ExHandlerCode objects associated to it. Using this list, NECG automatically generates the handler registration bytecode and puts it in the prologue of our newly generate method. Then our module restores the previously evaluated method so that the rest of the compiler can continue its evaluation. Finally our module return to the compiler a call instruction to our try method. For the rest of the language this was just a simple evaluation that generate a method call instruction.

As we can see, NECG handles almost everything about the bytecode generation so that we don't have to put any specific instructions in our Neverlang2 modules. When we change the execution machine, we'll find a specific NECG for generating the new machine's assembly necessary for the NERL intermediate language and then we can reuse these Neverlang2 modules.

4.3 Try-Catch-Throw-Retry language

```
module panzyLang.Ex.Retry{
imports{base.*;NECG.*;}
        reference syntax {
                Statement <- "retry" ;
        }
        role(evaluation) {
        0@{
                MethodInformation m = CodeUtility.currentMethod;
                String buffer = m.generateQuickReturn() ;
                int numArg = m.getNumArgs();
                $0.Text =
                PanzyLangMain.exSupport.retryCode()+
                PanzyLang.exSupport.getCallBarrierBytecode(buffer, numArg)+
        }.
        }
}
```

Our third version adds the the *retry* keyword upon the precedent version. We decided to add this keyword for two main reasons: the first one is to prove that NEL permits us to write exception handling procedures unsupported by the execution machine since the JVM doesn't support natively this keyword. The second reason is to show that our exception modules are modular too since we can add or remove features to it just like every other part of the language. In our implementation, when the keyword *retry* is encountered in a catch block, we return the execution at the start of the try block, repeating it.

Implementing our retry is simple thanks to NERL. All we have to do is to unwind the call stack until we find our try method, pop it and then recall it. This way we delete

all the previous computation and we return at the start of the try block, effectively retrying it.

5 Results and conclusions

The Neverlang2 Exception Library gives the possibility to add the support for exceptions into language lacking this feature but, since it uses an intermediate layer between the application and the real hardware, our solution can present an overhead in terms of execution time compared to a native application. For better understanding the impact NEL has on performances, we have developed the small language we presented in the previous chapter, PanzyLang. In this chapter we analyse the drawbacks of using our library and we discuss about improvements that our solution can see in the future for mitigate the problems we had found.

Calculating the differences in compilation time or generated code size is useless in our case study: PanzyLang is a toy language and, since NERL can be used without NECG, we don' t find this kind of data to be significative. We limited ourself to study the differences in execution time between an application written in PanzyLang that use NERL and an application in one other native programming language. We obviously expect that the native solution is faster but with this experiment we want to quantify the decrease of performance and see if it's justifiable by the modularity of our solution.

5.1 Performance analysis

Calculating execution time for exceptions is difficult. We want to understand how quickly an application can locate and execute an handler after the exception is raised and how quickly the application can resume its normal execution flow after the execution of the handler. This isn't enough however: as we saw in Chapter 3, our implementation of NERL introduces a write barrier in every memory access. We chose to compare the try-throw-catch PanzyLang version with the Java 1.8 programming language: both the languages have similar syntax, compile in native bytecode and use the same procedure for exception handling.

We need an application, written in both languages, to use as a benchmark. However, this application needed to meet some specific criteria:

- The application should make a lot of method calls in the try block. This is important because we want to understand how the stack traversal of our exception impacts on the execution time after the exception is raised.
- The application should execute a lightweight handler: we must execute an handle for understanding how quickly the application can resume its normal execution, however we don' t want to spend too much time in it since we want to see how the handling mechanisms performs, not the single handler.
- The application should frequently access memory without allocating it: NERL's write barriers are the major concern about performance and we want to understand how they impact on it. At the same time, we don't want to trigger any garbage collection, so we must avoid allocating objects during execution.

We developed a little application that calculates the factorial of a number using the recursive algorithm. However, we have removed the base step, so we trigger an exception when we try to multiply by zero. This causes the execution of an handler that updates a counter and repeats the procedure. This application meets all of our criteria: the recursive nature of our factorial function permits us to fill up the stack, the handler is lightweight and does both a read and a write access to an integer variable, so that we can study how our barriers impact on performance. We put the try block in a while block for repeating the same passages, so that out x variable represents the number of iteration.

```
public class Benchmark{
        public static int fact(int i){
                if(i==0)
                        throw new RuntimeException();
                else
                        return i * fact(i-1);
        }
        public static void main(String args[]){
                int a = 0;
                int i = 0;
                int x = //Read from stdin;
                while(i<x){
                        try{
                                a = fact(i);
                        }
                        catch (RuntimeException f){
                                a = a +1;
                        }
                        i= i + 1;
                }
        }
}
```

```
class BenchmarkPL{
        int fact(int i){
                if(i < 1 ) {
                        throw new FactException()
                }
                return i * fact(i-1)
        }
        void main(){
                int a = 0
                int i = 0
                int x = //Read from stdin
                while(i < x){
                        try{
                                a = fact(i)
                        }
                        catch(FactException f){
                                a = a + 1
                        }
                        i = i +1
                }
        }
}
```

In the following table we presents the results we had gathered. All the results are in milliseconds on the Oracle JRE 1.8.25.

x value	Java Stack unwind	PL Stack unwind	Java memory access	PL memory access	Java entire application	PL entire application
1	0,018	0,043	0,002	0,03	125	138
50	0,44	1,28	0,011	0,67	138	153
100	1,06	2,78	0,028	1,15	143	168
250	8,38	11,03	0,045	1,80	144	217
500	22,01	27,15	0,065	2,74	148	360
700	47,75	39,15	0,104	3,39	186	653

The results are interesting: as expected NERL introduces some overhead during stack traversal and memory access. However the native exception handler procedure is extremely slow with a large call stack. This is probably because it's optimised for a little call stack and because the JVM exception policy always generates a stack trace when an exception is raised.

As expected NERL's real problem can be found during memory access. Our NERL implementation needs to place a barrier between every memory access because it needs to know the current state of the machine, since we don't know when an exception is raised. We registered a 30 times increment compared to the native application during memory access. Obviously there are differences between how the Java compiler and the PL compiler optimise the bytecode for this application, and this kind of results aren't unexpected since we knew that handling the memory is the most critical part of

our project performance-wise. In the next sections we present some strategies that can drastically improve the performance of our memory manager.

Because of the overhead during memory accesses, the entire PanzyLang application is almost 3 times slower than the native application. This might be seen as an excessive degradation, however our benchmarks was purposely designed for underlying a worst-case scenario. Because of that, we think our solution is good enough for little languages that greatly benefit from a quick development cycle, like DSLs.

5.2 Future improvement

NEL is already in an usable state: with it, we developed a small language in which we can add exceptions support to a language modularly and independently from the rest of it. However there are some performance degradation and there is a lot of room for improvements. In this section we present some possibile improvements to our implementation that should make NEL a viable solution for real-world languages and DSLs.

NERL possesses a single, central call stack and can't work in a multithread environment. NERL doesn't support the concept of threads in its current form and NEL doesn't have any way to coordinate the accesses from different threads. A good idea could be utilising a separate stack for each thread the application possess but this can be a limitation for certain exception procedures.

Another area that should needs heavy improvements is memory management: since NERL must handle the memory used by the application, in our solution we put a barrier before every memory access, introducing a significative overhead to an already slow operation. An interesting improvement could be the creation of a hierarchical system with a cache for reducing the number of memory barrier operations, condensing NERL internal updates and therefore reducing the overall overhead of our layer.

Another interesting improvement could be updating the NERL stack concurrently: in this way we can probably further improve the memory barriers efficiency.

Finally, we can improve our architecture for improving the support to more complex exception handling algorithms. NEL permits to write an exception handler policy as we saw in Chapter 2, however it provides only basic methods for controlling the call stack and the machine's state.

5.3 Conclusions

At the start of this thesis we wanted to find a way for seeing the exception handling as an independent feature in order to being used in Neverland2. We have presented our solution in the form of an exception handling library to better abstract this feature and make it independent from both the language and the execution hardware. We implemented our library on the Java Virtual Machine, created a small object-oriented language lacking exceptions and we finally developed, using our NEL library, a set of modules that extend our language with two different types of exception. The results

demonstrate that we have created a solution with a great extensibility but we need more tests and different languages for understanding if NEL can be considered a good solution for every type of language. Also, while it is in a usable state right now, our library needs some improvements before it can be used in the creation of real-world languages and DSLs, like the supports for multithreading. However, with this thesis we have showed that we can write an exception handling mechanism independent from the rest of the language, making the exception handling an independent and portable feature.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, First Edition*. Pearson Education, 1986.

[2] John B. Goodenough. *Exception handling: issues and a proposed notation*. Communications of the ACM Volume 18, December 1975

[3] Marjan Mernik, Jan Heering, Anthony M. Sloane. *When and How to Develop Domain-Specific Languages*. ACM Computing Surveys (CSUR) Volume 37, December 2005

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[5] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Oracle, 2013

[6] Cazzola Walter. *Domain-Specific Languages in Few Steps: The Neverlang Approach*. SC12, June 2012

[7] Cazzola Walter, Vacchi Edoardo. *Neverlang2: Componentised Language Development for the JVM*. SC13, June 2013