Implementing a Purity and Side Effect Analysis for Java Programs

David A. Graf

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

Winter Semester 2005/06

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

I present an implementation of a new method purity analysis for Java programs, which is described in [1]. A method is pure if it does not mutate any location that exists in the program state right before method invocation. The analysis is built on top of a combined pointer and escape analysis for Java programs and is capable of determining whether methods are pure even if they do heap mutation, provided that the mutation affects only objects created after the beginning of the method.

My implementation is able to parse and analyze methods of Java Bytecode and writes the results into an XML-file that can be parsed and used by tools which need purity information. Additionally, the analyzer is able to export analyzer information of analyzed methods that can be reparsed and reused by later analyses.

I have tested the implementation on several test cases. Apart from some parsing problems, the analyzer runs correctly and in acceptable time. The parsing is based on an external tool which has problems with a few classes of the Java runtime environment. The results of the analyzer show that the analysis effectively recognizes a variety of pure methods, including pure methods that allocate and mutate complex auxiliary data structures.

To test and present my analyzer, I have additionally implemented a GUI which shows all information that is saved and computed by the analyzer.

Contents

1 Introduction **7**

2 Internal Representation **9**

3 Parser **13**
3.1 Bytecode Parsing . . . 13
3.1.1 Bytecode Parsing with the JODE Decompiler . . . 14
3.2 XML Parsing . . . 19

4 Analyzer **21**
4.1 Preparations . . . 21
4.1.1 Inheritance relations . . . 21
4.1.2 Calling relations . . . 21
4.1.3 Control Flow Graph . . . 23
4.1.4 Method Worklist . . . 23
4.1.5 Statement Worklist . . . 23
4.2 Analysis . . . 23
4.2.1 Analyzer Graph . . . 24
4.2.2 Intra-procedural Analysis . . . 25
4.2.3 Inter-procedural Analysis . . . 28

5 Results of the Analysis **31**
5.1 Purity . . . 31
5.2 XML Output . . . 31
5.3 Starting the Analyzer . . . 33
5.3.1 Command Line . . . 33
5.3.2 Imported JAR . . . 34
5.3.3 GUI . . . 34

6 Future Work / Conclusion **37**
6.1 Future Work . . . 37
6.2 Conclusion . . . 37

Chapter 1

Introduction

Methods in object-oriented languages often update the objects that they access, including the "this"/"self" object. Accurately characterizing these updates is important for many tasks. In the Software Component Technology Group[1], many implemented analyzer tools for Java programs need information about purity of each method which will be analyzed. Examples for such projects are the type inference tools [3] and [4]. Currently, the purity information is passed to the analyzers by hand with an XML-file. With my implementation, it will be possible to automate this task.

Commonly, researchers in a variety of fields have identified method purity as a useful concept. For example, pure methods can be used safely in program assertions and specifications. When model checking Java programs, it is important to know that methods are pure because this information allows the model checker to reduce the search space by removing irrelevant interleavings. Examples for using this concept are JML[2] and the Universe Type System[5].

The implementation presented in this report is based on a paper [2] which introduces a new method purity analysis for Java Programs and the corresponding technical report [1]. The analysis is built on top of a combined pointer and escape analysis that accurately extracts a representation of the region of the heap that each method may access. The analysis conservatively tracks object creation, updates to the local variables and updates to the object fields. This information enables the analyzer to distinguish objects allocated within the computation of the method from objects that existed before the method was invoked.

Therefore, the analyzer can check that a method is pure, in the sense that it does not mutate any object that exists in the program state right before the method invocation. This definition allows a method to perform mutation on newly allocated objects (important for iterators).

The analyzer could be divided into three parts (see figure 1.1): a parser, an analyzer and a result part. The parser (see chapter 3) parses Bytecode (.class-files) and transfers them into an internal representation which is explained in chapter 2. The internal representation is analyzed by the implementation of the analyzer, which is presented in the technical report[1]. This analyzer generates for each method an analyzer graph (see section 4.2.1), which represents the combined pointer and escape analysis. With the help of this analyzer graph, it is possible to find out if a method is pure or not not and write this information into an XML-file.

In addition to this 'core' functionality, I have implemented a device, to save analysis information about already analyzed methods in an XML-file and to load analysis information about methods. This device is useful in two ways. First, methods, which are used often and which need a lot of resources to be analyzed (numerous methods from the Java Runtime Environment have this property, such as *System.out.println*) have to be analyzed only once. Second, a lot of Java classes from the Java Runtime Environment contain native methods. Logically, these classes cannot be analyzed (with this analysis). Thus, the user will be able to declare the analyzer graph of native methods by hand and to load this informations over an XML-file into the purity analyzer.

[1] http://sct.ethz.ch

[2] Java Modeling Language http://www.cs.iastate.edu/~leavens/JML/index.shtml

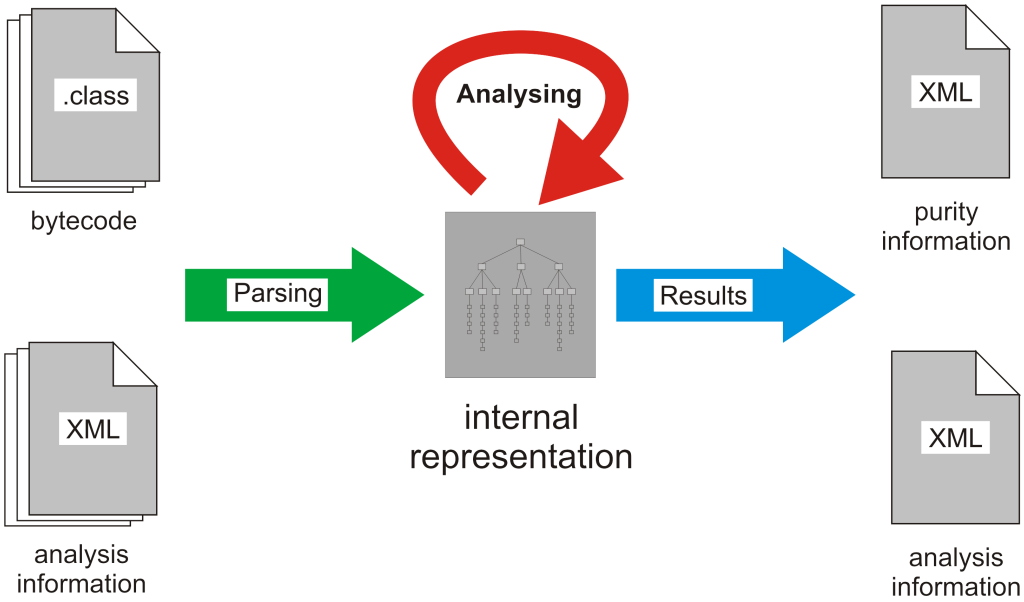

Figure 1.1: overview purity analyzer

The purity analysis can either be started over the command line, a GUI or an imported JAR in a Java project. The GUI shows the internal representation of the analyzer in an effective way. How this three possibilities can be started and what they do exactly is explained in chapter 5. When starting the analyzer, the user can decide if he wants an XML-file with purity information or an XML-file with analyzer information as output.

In the last chapter (chapter 6), I present possible extensions to the purity analyzer (future work) and the conclusions.

Chapter 2

Internal Representation

To analyze .class-files they have to be transformed into an internal representation (IR) which can by analyzed by the purity analyzer. In the following description, I explain in which state the parser has to pass the IR to the analyzer. All classes for building the IR can be found in the package *ch.ethz.inf.sct.purity_analyzer.ir* in my implementation. Before the 'real' analysis can start, the analyzer has to add some other (redundant) information to the IR. This is explained in section 4.1.

The IR is a tree with a root object which is an instance of the class *IRTree*. This root object contains a list of classes (instances of *IRClass*) and interfaces (instances of *IRInterface*). The interface objects contain a list of names of interfaces which they extend. That is all an interface object has to save. The analyzer doesn't need any methods and fields of the interfaces, because the analysis is only built on the program code.

The class objects contain a list of names of interfaces which they implement, and a name of a class which they extend. Like the interfaces, the class objects don't have to save the fields, but logically information about the methods. Each class object contains a list of methods (instances of implementations of *IRMethodable*). The implementations of *IRMethodable* have to save all information which are written in the method head, and have to be able to return analyzer information. To the information of the method signature belongs whether the method is static, abstract, and a constructor and the list of parameters. A parameter is a tuple of type and name of the parameter. The parameters are used to compute the signature of a method. Because of simplicity, the analyzer doesn't use the same signature as Java Bytecode. The signature of the analyzer consists of the name of the method and in brackets and separated with commas the full parameter types (e.g. $public\ void\ test(int\ i,\ Object\ o) \Longrightarrow test(int,\ java.lang.Object)$).

As mentioned above, the implementations of *Methodable* have to be able to return analyzer information. There are two possibilities where this information comes from. First, this information is saved in an XML-file. To save such methods, the analyzer uses the implementation for light methods (*IRMethodLight*). This implementation is only able to save analyzer information. The second implementation (*IRMethod*) saves the program code of the method and will later be analyzed by the analyzer, which means that the analyzer information will be built later.

How the program code has to be saved is introduced in the technical report[1]. The program code has to be transferred into a list of a small subset of Java instructions, called statements. This subset consists of statements which are introduced in figure 2.2. The first section of the statements in the figure comes from the technical report. It was problematic to represent Bytecode with these statements, because the Goto statement is missing; therefore I introduced an additional Goto statement. Because the set of statements contains jump instructions (Goto and If), it must be possible to set labels for the statements. This lead to a further problem: Sometimes the parser (which is explained in chapter 3) knows where to jump, but the statement there is not yet defined. (For example during parsing a while loop, it is not possible to know what the next statement will be after the while loop. But at the condition of the while loop, it must be possible to jump to the next statement after the loop.) To bypass this problem, I introduced a statement that doesn't have a particular purpose except for having a label. This statement is called Nop. The Entry and

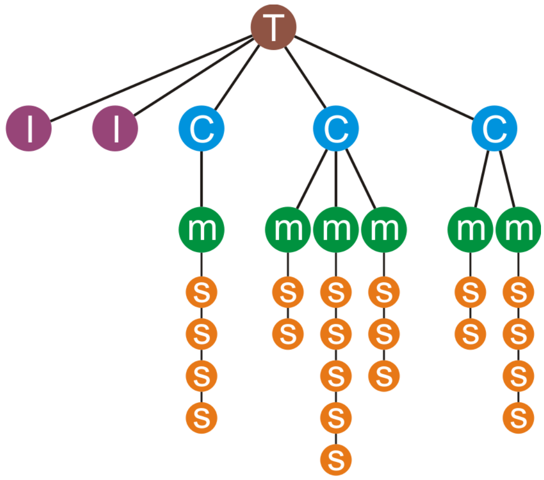

Figure 2.1: Internal representation example: The IR represents a set of classes (C) and interfaces (I). These sets are saved in a root object (T). Classes contain methods (M) and each method a list of statements (S).

Statement	**Statement Format**	**Informal Semantics**
Copy	$v_1 = v_2$	copy one local variable into another
New	$v = new\ C$	create a new object of class C; all fields of the new object are initialized to *null*
NewArray	$v = new\ C[k]$	create an array of k references to objects of class C; all array cells are initialized to *null*
Store	$v_1.f = v_2$	store a reference into an object field
StaticStore	$C.f = v$	store a reference into a static field
ArrayStore	$v_1[i] = v_2$	store a reference into an array cell
Load	$v_1 = v_2.f$	load a reference from an object field
StaticLoad	$v = C.f$	load a reference from a static field
ArrayLoad	$v_2 = v_1[i]$	load a reference from an array cell
If	$if\ (...)\ goto\ a_t$	conditional transfer of control to stmt with label a_t
Call	$v_R = v_0.s(v_1, ..., v_j)$	call method named s of object pointed to by v_0
Return	$return\ v$	return from the currently executed method with the result v
ThreadStart	$start\ v$	start the thread pointed to by v

Helper Stmt	**Statement Format**	**Informal Semantics**
Goto	$goto\ a_t$	transfer of control to stmt with label a_t
Nop		does nothing, helper to set a label
Entry		start statement in the control flow graph
Exit		end statement in the control flow graph

Figure 2.2: Relevant Statements in the Analyzed Program

Exit statements are used for the start and end node of the control flow graph of a method (see subsection 4.1.3).

I have implemented a class for each type of statement, which can be found in the package *ch.ethz.inf.sct.purity_analyzer.ir.stmts*. The types of statements are extended from the class *Statement* which saves all data. This is used by each type, like the label and the list of predecessors and successors (used to build the control flow graph, see subsection 4.1.3). The classes which represent statements have to save the information, which is in figure 2.2 in the column *Stmt Format*, except all information about size of array initialization and index of array store and load. This information is not used by the analysis. For example the Copy statement has to save the names of the source and destination variable, the New statement, the name of the destination and of the class, the If statement the jump address, and so on.

What are these statements for? I have explained in the introduction that the analysis is built on a combined pointer and escape analysis. To perform this analysis, the analyzer generates an analyzer graph, a so called points-to graph (see chapter 4). The technical report[1] introduces rules how the statements make changes to the points-to graph. What happens with the helper statements? For the analysis, the analyzer has to build for each method the control flow graph. The analyzer eliminates the helper statements during building this control flow graph (see subsection 4.1.3).

Chapter 3

Parser

As already mentioned in the introduction, the purity analyzer has two possibilities to parse data. One possibility is to parse Bytecode and the other is to parse analyzer information from already analyzed methods out of XML-files.

All classes for the parsing can be found in the package *ch.ethz.inf.sct.purity_analyzer.parser*. The parser is started by invoking the method *parse* in the class *Parser*. The parameters are an array of package or class names (classes which are imported by Bytecode parsing) and an array of paths of XML-files (classes, which are imported by XML parsing).

3.1 Bytecode Parsing

Parsing Bytecode and transferring the program code of the methods into a list of statements was really a challenge. Because parsing the Bytecode myself would be absolutely too complex, I had to find a tool which parses the Bytecode for me. Tools which parse Java Bytecode in some way exist abundantly, but my Bytecode parse has to be written in Java and work under Java 1.5. Further, it has to present the program code of a method in a way that I can build the statement list which I have presented in chapter 2.

First, I tested special analyzer tools for Java Bytecode. Examples for such analyzer tools are BCEL[1] from the Apache Jakarta Project[2] and ASM[3] from ObjectWeb[4]. The problem with these analyzer tools was always the same. They could not pass the program code in a way that would have been useful for me. They save the program code at the methods with an array of Bytecode instructions which don't contain variable names, because Java Bytecode doesn't contain them. Bytecode may contain information about variable names (normal case), but this information is saved at the end of each method and not in the program code (see in the following Bytecode the variables *this* and *a*).

```
@signature "(I)V"
    public void <init>(int) {
        @line    10
        _L0:     @aload           0
                 @invokespecial   void java.lang.Object.<init>()
        @line    11
                 @aload           0
                 @iload           1
                 @putfield        int Test.a
        @line    12
```

[1] Byte Code Engineering Library http://jakarta.apache.org/bcel/
[2] http://jakarta.apache.org
[3] http://asm.objectweb.org/
[4] http://www.objectweb.org/

```
_L5:    @return
@var    0: this Test [ _L0, _L5]
@var    1: a int [ _L0, _L5]
```

Transferring Bytecode instructions into the instruction of the IR would not be a problem, but I had to find a tool which either transfers the variable information into the Bytecode or introduces new variable names for the unknown ones. Finding such a tool was not very easy, so I asked the writer[5] of the paper[2] what he is using for his implementation. He is using the Flex Compiler Infrastructure[6] which was implemented by the same research group he comes from. But this tool didn't solve my problem, because I never got it running. I asked for documentation, but I never got any and the code of the tool contains nearly no comments.

After these recurring drawbacks, I was a little bit desperate, but then my supervising assistant had a resounding idea. The idea was to use the internal representation of a decompiler to build the IR of the purity analyzer. Thus, I had to select a decompiler. This wasn't really difficult, because I only found one decompiler, which is written in Java and works under Java 1.5, the JODE[7] decompiler.

3.1.1 Bytecode Parsing with the JODE Decompiler

As mentioned above, the Bytecode parser gets an array of packages and classes which the user wants to have analyzed. Important about this classes and packages is that they have to be in the Classpath and that they are written in the normal Java syntax (e.g. *java.lang.Object*). So first, all classes which are in the passed packages have to be found. This is mastered with the help of JODE. JODE provides a class *Classpath* (*net.sf.jode.bytecode.ClassPath*) which has to be instantiated by passing all Classpaths of the current Java program. The instantiated object contains methods like *isPackage(packageName)*, *existsClass(className)* and *listClassesAndPackages(packageName)*. With these methods, it is possible to find all class names.

Then the analyzer has a list of all classes (and interfaces) which it has to analyze. To do this, JODE has to decompile all these classes:

```
// gets classInfo of className (JODE-internal information object of a class)
ClassInfo classInfo = classPath.getClassInfo(className);
// instantiates a class analyzer object with the classInfo and some settings (imports)
ClassAnalyzer classAnalyzer = new ClassAnalyzer(classInfo, imports);
// lets JODE decompile the class, writer = where JODE writes the decompiled Java-File
classAnalyzer.dumpJavaFile(writer);
```

After 'dumping' all classes, the Java-code of all classes is written into the *writer*. But that is only a nice side effect. The major benefit is that the internal representation of JODE is saved in the object *classAnalyzer*.

From the *classAnalyzer* object, the analyzer reads if the class is an interface and the names of super classes and super interfaces. The object also contains a list of *methodAnalyzer* objects (of course not for interfaces). These objects contain the internal representation of the methods. It is rather intricate to get these method analyzers:

```
// iterates through all methodInfos of the classInfo
for (MethodInfo methodInfo : classInfo.getMethods()) {
  // gets type of the methodInfo
  MethodType methodType = Type.tMethod(classPath, methodInfo.getType());
  // gets methodAnalyzer with the name and type of the method
  MethodAnalyzer methodAnalyzer = classAnalyzer.getMethod(methodInfo
                                  .getName(), methodType);
```

[5] Alexandru Sălcianu http://www.mit.edu/~salcianu
[6] http://www.flex-compiler.lcs.mit.edu/
[7] Java Optimize and Decompile Environment from Jochen Hoenicke http://jode.sourceforge.net/

```
}
```

The method analyzer object contains all information the purity analyzer needs to analyze the method, that is, whether the method is abstract, static, all parameters (name and type) and the program code in a special form.

The program code is stored as a tree with blocks and expressions. A Block correlates with a block in Java code (like an if-block, loop-block, try-block, etc.) and an expression correlates with an expression in Java code (like $a = b + c$, $b + c$, etc.). You see that the IR of JODE is very similar to Java code. On the one hand, it is an advantage, because now all variables can be found out easily, on the other hand everything is nested. A block can contain other blocks and expressions and an expression can contain other expressions (e.g. the expression '$a = b + c$' contains the two subexpressions 'a' and '$b + c$)'.

To transfer the block-expression-tree into my internal representation, the analyzer walks through all these blocks and expressions and builds the statement list for them. In JODE, the blocks can be found in the package *net.sf.jode.flow* and the expressions in *net.sf.jode.flow.expr*.

To get the statement list of a block, the analyzer contains for each type a method which transfers the specific block into a statement list. These methods are in the class *ParserBlocks*. To invoke such a method, I have implemented the following method in the class *Parser* that builds the method name with the help of the block type and invokes the corresponding method by reflection.

```
void parseBlock(StructuredBlock block, IRMethod irMethod,
                IRStmtList stmtList, String breakLabel, String continueLabel) {
  // creates method name with the class of the block
  String methodName = "parse" + block.getClass().getSimpleName();
  Class partypes[] = { block.getClass(), this.getClass(), irMethod.getClass(),
                       stmtList.getClass(), String.class, String.class };
  Object arglist [] = { block, this, irMethod, stmtList, breakLabel,
                        continueLabel };
  // gets the class of the method
  Class cls = Class.forName(this.getClass().getPackage().getName()
              + ".ParserBlock");

  // gets the method
  Method method = cls.getDeclaredMethod(methodName, partypes);
  // invoke the method
  method.invoke(null, arglist );
```

StructuredBlock is the super type of all types of blocks, *irMethod* is the current method, *stmtList* is where the invoked method adds the new statements (at the end) and the break and continue labels are used for break and continue instructions.

The following table explains, how each block type from JODE is transferred into the statement list *stmtList* for the internal representation.

Block Name	**Additions to the statement list**
BreakBlock *break* | *break label*	If break has no label, then add a Goto statement that jumps to the break label (parameter). If break has a label, then add a Goto statement that jumps to the label *"%end%" + label*. This works, because the parse methods for loop and switch blocks which are labeled add a Nop with this kind of label at the end.
CaseBlock *case in a switch block*	Parse each subblock.

Continued on next page

Block Name	Additions to the statement list
CatchBlock	Right now, only all subblocks of this block are parsed. Theoretically, the exception which is passed to the block should also be observed. But because of simplicity reasons, this is not implemented yet (see 6.2).
ConditionalBlock *if-then-block with an empty then-part and no else-part*	Parse the condition-expression.
ContinueBlock *continue* | *continue label.*	If continue has no label, then add a Goto statement that jumps to the continue label (parameter). If continue has a label, then add a Goto statement that jumps to the label *"%begin%" + label.* This works, because the parse methods for loop blocks which are labeled add a Nop with this kind of label at the end.
EmptyBlock	empty → does nothing
FinallyBlock	Parse each subblock. This structure is never used in Java 1.5, because the compiler adds the code of the final block at the end of the try-and of each catch-block.
IfThenElseBlock	Parse the condition-expression and the then- and else-block. Adds the stmtLists from the condition, the 'then' and the 'else' with additional Goto's, If's and Nop's to the stmtList, that the inserted statements represent the code from the if-then-else block (first the stmts from the condition, then an if, then the stmts from the 'then', etc.).
InstructionBlock	An instruction block contains an expression. It parses this expression and adds the statements to the stmtList.
LoopBlock	Parses the condition-expression and the body (which is a block). Adds the resulting stmtList from the condition and the body with additional Goto's, If's and Nop's to the stmtList that the inserted statements represent the code from the loop block. If the loop has a label, then adds a Nop with the label *"%begin%" + label* at the beginning and a Nop with the label *"%end%" + label* at the end. If the loop has no label, then add Nop's with newly created labels at the beginning and the end. This has to be done because of possible break and continue labels in the body of the loop. The names are passed to the parsing of the body by passing the name of the 'begin Nop' as break label and the name of the 'end Nop' as continue label.
ReturnBlock	Parses the expression in the return instruction and adds a new Return statement to the stmtList.
SequentialBlock	Parse each subblock.
SwitchBlock	Parse the condition-expression, each case-block and the default-block. Adds the stmtLists from the condition, the cases and the default with additional goto's, if's and Nop's to the stmtList that the inserted statements represent the code of the switch block. A switch block might also contain break labels. To be able to handle them, the analyzer has to insert a Nop before the default block and to pass the name of the Nop as break label.
SynchronizedBlock	Parse each subblock.
ThrowBlock	In the ThrowBlock, the analyzer only analysis the thrown expression. That the thrown object should be handled specially is not implemented yet (belongs to the not yet solved problem in the Catch block, see 6.2).

Continued on next page

Block Name	Additions to the statement list
TryBlock	The TryBlock is handled like as Switch statement without a condition. The try and catch blocks are like 'cases' and the finally block like 'default'. That is not a correct transformation of the code, but the analysis of a switch block and a try block is the same.

The same has been implemented for the expressions. The methods for parsing an expression are in the class *ParserExpression*. To invoke the methods, I have implemented a similar method as for the blocks.

```
String parseExpression(Expression expr, IRMethod irMethod,
                       IRStmtList stmtList, String destVar) {
  // creates method name with the class of the block
  String methodName = "parse" + expr.getClass().getSimpleName();
  Class partypes[] = { expr.getClass(), this.getClass(), irMethod.getClass(),
                       stmtList.getClass(), String.class };
  Object arglist [] = { expr, this, irMethod, stmtList, destVar };

  // gets the class of the method
  Class cls = Class.forName(this.getClass().getPackage().getName()
            + ".ParserExpression");

  // gets the method
  Method method = cls.getDeclaredMethod(methodName, partypes);
  // invoke the method
  return (String)method.invoke(null, arglist );
```

Expression is the super type of all types of expression, *irMethod* is the current method, *stmtList* is where the invoked method adds the new statements (at the end). *destVar* is a special construct. A lot of expressions are expressions, which save something in a variable. If such an expression should save its result in a known variable, then the name of the variable can be passed over the *destVar*. If such an expression gets *null* as *destVar*, then it saves its result in a newly created variable and returns the name of this variable.

The following table explains, how each expression type from JODE is transferred into the statement list stmtList for the internal representation.

Expression Name	Addings to the statement list
ArrayLength *subexpr.length*	Array length not significant for the analysis. Only parse subexpression (subexpr).
ArrayLoad *a[i]*	Parses subexpression *a*. Creates new ArrayLoad Statement. The name of the source is the returned string of the parsing of the subexpression and the name of the destination comes from the passed parameter destVar. If destVar is *null*, creates a new destVar (invoke method *getNameForAdditionalVariable* in the passed object irMethod). Returns destVar.
Binary Operations e.g. *a + b*	Operations with base types are ignored. Only the subexpressions are parsed (a and b).
CheckCast *(Type)subexpr*	Parses subexpression.

Continued on next page

Expression Name	Addings to the statement list
ClassField *ClassName.class*	Create a new StaticLoad statement with ClassName as source class,'class' as source field and passed destVar as destination if not *null*, else create new destVar.
CompareBinary $a < b$	Ignored. Parses only subexpression (a and b).
CompareToInt $a < 1$	Ignored. Parses subexpression (a).
CompareUnary $a < 1.2$	Ignored. Parses subexpression (a).
ConstantArray *A[] a* = {*x,y,z*}	Parses all subexpressions (x, y, z). Creates a NewArray Statement with A as type and the passed destVar as destination if destVar not *null*, else creates a new destVar. Afterwards, creates for each variable x,y and z an ArrayStore statement with destVar as array name and the variable name (x,y or z) as source.
Const *a constant*	Only significant, if the constant is *null*, because constants are always base types or *null*. Then create a new Copy statement with source "null" and destination destVar and return destVar. Else return *null*.
Convert *(Type)subexpr*	Ignored. Parses subexpression.
GetField *static (Type.x) or non-static (subexpr.x) field access*	If non-static, parse subexpression and creates new Load statement. If static, creates new StaticLoad statement.
IfThenElse *(a ? b : c)*	Handeled like an IfThenElse block.
IInc *subexpr++*	Ignored. Parses subexpression.
InstanceOf *subexpr instanceof type*	Ignored. Parses subexpression.
InvokeOperator $subexpr.m(v_1, ..., v_2)$ or $Type.m(v_1, ..., v_2)$	Parses subexpression, if non-static method invocation and argument expressions $(v_1, ..., v_2)$. Creates parameter list and new Call statement. The parameter of the parameter list have as type the parameter type of the invoked method, and not the static type of the passed arguments (except for the 'this' argument in non static method, because this undecidable)! That is like this, because a call statement must be able to compute the signature of the invoked method. If the method invocation invokes the method *start0* of the class *java.lang.Thread*, creates a new ThreadStart statement, because this method starts a new thread.
LocalLoad	LocalLoad represents a read access to a local variable. If the variable is a base type, returns null. If the passed destVar is *null*, return the name of the variable, else create a new Copy statement with the destination destVar and the variable name as source.
LocalStore	LocalStore represents a write access to a local variable. Returns the name of the local variable.
MonitorEnter	Ignored. Parses subexpression (object, which is entered to the monitor).
MonitorExit	Does nothing.
NewArray *destVar = new Type[]*	Creates a new NewArray statement.

Continued on next page

Expression Name	Addings to the statement list
New *destVar = new Type()*	Creates a new New statement.
Nop	Does nothing.
OuterLocal	Special load local expression. Handled like LoadLocal.
PrePostFix	Some special binary operator. Ignored. Parses subexpressions.
PutField	Ignored because the PutField expression should be catches at the parsing of the expression Store. Parses subexpressions, because it happens in some special cases that the parser comes to this expression.
Store *a = b*	Parses subexpression if necessary. Creates, dependent of the two sub expressions a StaticStore, a Store, an ArrayStore statement or nothing.
StringAdd *a = "a" + "b"*	Ignored. Parses subexpressions.
This	If destVar is *null*, then returns "this". Else add a new Copy statement with source this and destination destVar.
Unary	Ignored. Parses subexpressions.

In the descriptions above, I have often mentioned 'base types'. Base types are types, which aren't significant for the analysis. Theoretically, a type is not significant to the analysis, if it is known that the type doesn't contain impure methods. In my implementation, I defined all Java primitive types as base types and String. String, because it contains only pure methods and is used very often.

3.2 XML Parsing

Parsing the exported IR's is done with XMLBeans[8]. The XML Schema file which defines the XMLBeans generated runtime JAR[9] is in the folder *xml* and is called *annotations.xsd*. The Schema contains also annotations for the purity output (see chapter 5). The generated runtime JAR is in the folder *lib* and is called *xmlTypes.jar*.

After parsing the XML-file with xmlBeans, the analyzer transfers the structure from XMLBeans into the its IR. It is noteworthy that this analysis builds only 'light' methods (*IRMethodLight*), because the XML-file saves for each method the analyzer information. Further, the XML parser doesn't overwrite methods, which are already inserted into the IR from the Bytecode parser. The XML parser adds only the methods which haven't been parsed by the Bytecode parser.

[8] http://xmlbeans.apache.org/
[9] See XMLBeans Schema Compilation http://xmlbeans.webappshosting.com/schemaToolsV103/compile.do

Chapter 4

Analyzer

The analyzer part is more or less a transformation of the description in the paper [1] into code with little extension.

The analyzer is divided into two parts. First, it makes some preparations to the IR. Afterwards, it does the 'real' analysis.

4.1 Preparations

During the preparation phase, the following items have to be added to the internal representation: inheritance relations, calling relations, control flow graph of each method and the order in which the methods and statements have to be analyzed (method worklist, statement workslist).

4.1.1 Inheritance relations

The classes in the IR have to save the super class, if they have one and all sub classes and super interfaces. The interfaces have to save all super interfaces, sub interfaces and sub classes.

The parser saves only the names of the super interfaces and sub class in each class object, and the super interfaces in each interface object. To find the classes and interfaces of these names, the classes and interfaces are saved in maps with their names as key in the root of the IR tree.
Algorithm to get the super class, list of super interfaces and list of sub classes for classes and list of super and sub interfaces for interfaces:

- For each interface (*curInt*):
  - For each superinterface name of *curInt*:
    - Gets the interface (*superInt*) of the superinterface name.
    - Adds *superInt* to the super interfaces of *curInt*.
    - Adds *curInt* to the sub interfaces of *superInt*.
- For each class (*curClass*):
  - For each superinterface name of *curClass*:
    - Gets the interface (*superInt*) of the superinterface name.
    - Adds *superInt* to the super interfaces of *curClass*.
    - Adds *curClass* to the sub classes of *superInt*.
  - If the *curClass* has a superclass name:
    - Gets the class (*superClass*) of the name.
    - Adds *superClass* to the super classes of *curClass*.
    - Adds *curClass* to the sub classes of *superClass*.

4.1.2 Calling relations

To build the worklist of methods, the analyzer needs to know the callers and callees of each method. Further, the call statements in the statement list of each method have to know all possible methods

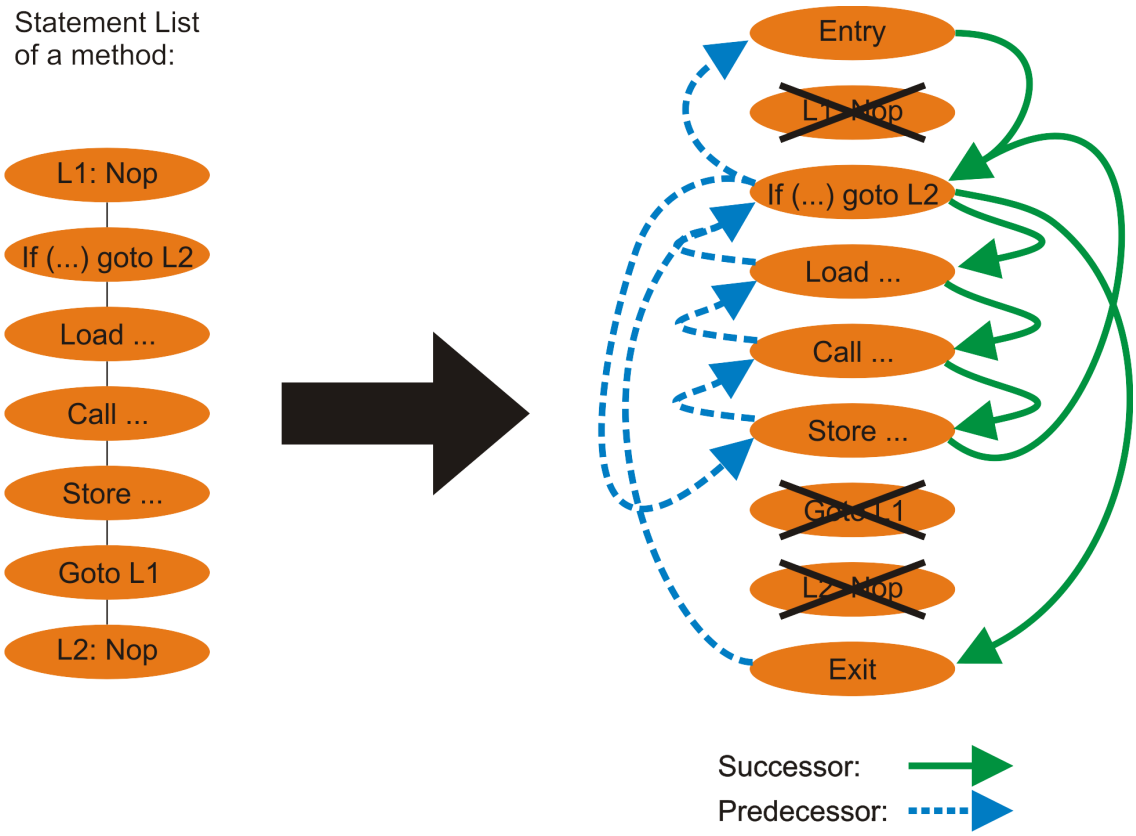

Figure 4.1: example of control flow creation

which the invocation could invoke, which means that each call statement has to know is possible callees.

To find out these relations, the analyzer has to iterate through all Call statements and to search in all possible dynamic types in which the method could be invoked, the method with the same signature as the Call statement. Searching all possible dynamic types works like this:

- If the static type of the method invocation is an interface:
  - Searches in all direct and indirect sub classes of the interface for methods with the same signature as the method invocation.
- If the invoked method is a constructor, 'static', or 'super':
  - Searches in the static type the method with the same signature as the method invocation.
- Else (normal invocation on an object of a class):
  - Searches in the static type the method with the same signature as the method invocation.
  - If the method cannot be found in the static type, it searches in the next super class. If it cannot be found in the super class, then it searches in the super class of the super class, and so on.
  - Searches in all direct and indirect sub classes of the static type for a method with the same signature as the method invocation.

4.1.3 Control Flow Graph

For each method, the list of statements has to be converted into a control flow graph. This is done by saving at each statement the statement's successors and predecessors in the control flow. During this process, the two helper statements (Goto, Nop) are deleted respectively replaced with links and an Entry and an Exit statement are introduced, that the analyzer knows where to start and where to end (see figure 4.1). The Entry statement has the first statement of the method as successor, the Exit statement the last and all Return statement as predecessors.

4.1.4 Method Worklist

The methods should be analyzed in an optimal order: The analysis of a method is dependent on all its callees. Because of this, before a method is analyzed, all callees should have been analyzed. Logically, this is not always possible, because of strongly connected methods (methods, which mutually call each other).

The analyzer builds a worklist of methods that as few methods as possible are analyzed before their callees. Algorithm:

- Builds a list *curMethodList* with all methods of the IR.
- As long as *curMethodList* is not emtpy, it gets a *method* form the list and calls *deepFirstStrategy(method)*

deepFirstStrategy(Method method):

- Marks *method* as visited and deletes it in the *curMethodList*.
- For each *callee* of *method* that is not visited
  - calls *deepFirstStrategy(callee)*
- Inserts *method* at last position in the *methodWorklist*.

4.1.5 Statement Worklist

The reason for having a statement worklist at each method is the same as for the method worklist. The analysis of a statement is dependent on all the statement's predecessors. This is also not always possible, because of loops.

The analyzer builds a worklist of statements for each method, that as few as possible statements are analyzed before their predecessors. Algorithm:

- Builds a list *curStmtList* with all statements of a method.
- As long as *curStmtList* is not empty, it gets a *stmt* form the list and calls *deepFirstStrategy(stmt)*

deepFirstStrategy(Statement stmt):

- Marks *stmt* as visited and deletes it in the *curStmtList*.
- For each *successor* of *stmt* that is not visited
  - calls *deepFirstStrategy(successor)*
- Inserts stmt at first position in the *methodWorklist*.

4.2 Analysis

The analysis, which is described in this section, builds for each statement an analyzer graph (see subsection 4.2.1). The analyzer graph of a method is equivalent to the analyzer graph of the last statement (Exit statement) which is used to find out if a method is pure or not (see chapter 5).

The analyzer iterates through the method worklist (see subsection 4.1.4) and analyzes each method. After analyzing a method, the analyzer checks if the analysis of the method has made any changes to the analyzer graph of the method. If yes, the analyzer analyzes again all callers

$$
\begin{aligned}
n &\in Node = INode \uplus PNode \uplus LNode \uplus n_{GBL} \\
\\
n^I_{lb} &\in INode(\text{inside nodes}) \\
n^P_{m,i} &\in PNode(\text{parameter nodes}) \\
n^L_{lb} &\in LNode(\text{load nodes}) \\
\\
\langle n, f \rangle &\in AField = Node \times Field(\text{abstract fields}) \\
\\
I &\in IEdges = \mathcal{P}(Node \times Field \times Node) \\
O &\in OEdges = \mathcal{P}(Node \times Field \times LNode) \\
L &\in LocVar = Var \rightarrow \mathcal{P}(Node) \\
\\
G &\in PTGraph = IEdges \times OEdges \times LocVar \times \mathcal{P}(Node)
\end{aligned}
$$

Figure 4.2: Sets and notations of the Points-To Graph

of the method, which already have been analyzed. This is done, because the analysis of a method is dependent on the analyzer graphs of all callees. When a callee's analyzer graph changes, the method has to be analyzed again.

What happens when the analyzer analyze a method? It iterates through the statement worklist (see subsection 4.1.5) and analyzes each statement. After analyzing a statement, the analyzer checks if the analysis of the statement has changed the analyzer graph of the statement. If yes, the analyzer analyzes again all successors of the statement which already have been analyzed. This is done, because the analysis of a statement is dependent on the analyzer graphs of all predecessors. In the technical report[1], the analysis of a statement is called intra-procedural analysis. The intra-procedural analysis is explained in the subsection 4.2.2, after the analyzer graph.

For method invocations which are represented by the Call statement, the analyzer has to map the analyzer information of the invoked methods to the Call statement. This operation is called inter-procedural analysis and is explaind in the subsection 4.2.3.

4.2.1 Analyzer Graph

As I have written in the introduction, the whole analysis is built on top of a combined pointer and escape analysis. To represent the pointer analysis, the analyzer computes for each statement an analyzer graph that models the heap and a set W_m of mutated, externally visible (from outside the method to which the statement belongs to) abstract fields, after executing that statement (changed fields). In the technical report[1], the graph is called Points-to Graph (see sets and notations of the graph in figure 4.2).

The nodes represent objects and the arrows references on the heap. Parameter nodes represent objects, which have been passed as parameter, load nodes objects, which are referenced from a node, which is visible from the outside and inside nodes objects, which have been newly created. The graph contains two different types of references. Outside edges represent references, which have existed before method invocation and inside edges references, which have been newly created from the current method. Local variable (LocVar) represents all possible objects to which a local variable might point to.

Additional to the elements which are given by the technical report, I have added some elements to the Points-to Graph:

- A set of abstract fields to represent the mutated fields W_m (changed fields).
- A linked list of the parameter nodes (in the order in which they are passed). This list is useful for the inter-procedural analysis (subsection 4.2.3).
- A set of globally escaped nodes. That are nodes which are referenced from outside the method.
- A set of return nodes (represents all possible return objects).
- A set of all local variables which have been used.

The creation of new nodes is rather special. If a statement asks for a new parameter node, it always gets one with another unique name. If a statement asks for a new load or inside node, it always gets the same (unique) one. The special handling with inside and load nodes is because of possible loops. A statement creates usually one new node (except Entry statement). If it always gets another one, the statement would change the analyzer graph in a loop forever. The parameter nodes are not handled like this, because they are only created in the Entry statement.

The changed fields are represented by a tuple of a node and a field. As I have written before, the node represents an object. What happens if a statement mutates a static field? In this case, the analyzer contains one instance of the class GlobalNode. This instance is used to create a changed field for a write access to a static field.

4.2.2 Intra-procedural Analysis

The paper[1] names the transformation that each statement causes to the analyzer graph, intra-procedural analysis. To start the intra-procedural analysis of a statement, the analyzer has to create the analyzer graph which represents the heap right before statement execution. This graph is built by merging all analyzer graphs from the predecessors.

The following table explains the changes to the analyzer graph from each statement type. The reasons for doing all these things are mentioned in the technical report[1] in the section 5.1.

Stmt Name	**Transformations to the *Analyzer Graph***
Copy $v_1 = v_2$	• Gets $LocVars$ of v_1 and v_2 • Removes all nodes from $LocVar_{v_1}$ • Adds all nodes of $LocVar_{v_2}$ to $LocVar_{v_1}$
New $v = new\ C$	• Gets $LocVar$ of v • Removes all nodes from $LocVar_v$ • Adds a new inside node to $LocVar_v$
NewArray $v = new\ C[k]$	• Gets $LocVar$ of v • Removes all nodes from $LocVar_v$ • Adds a new inside node to $LocVar_v$

Continued on next page

Stmt Name	Transformations to the *Analyzer Graph*
Store $v_1.f = v_2$	• Gets *LocVar* of v_1 • Adds all not *Internal Nodes* of $LocVar_{v_1}$ with the field f to *Changed Fields*. • If v_2 is not null (not a basetype): – Gets *LocVar* of v_2 – Introduces a new *Internal Edges* from all nodes of $LocVar_{v_1}$ to all nodes of $LocVar_{v_2}$ with field f
StaticStore $C.f = v$	• If v is not *null* (not a base type): – Gets *LocVar* of v – Adds all nodes of $LocVar_v$ to the list of global nodes • Adds field f with node *Global Node* to *Changed Fields*
ArrayStore $v_1[i] = v_2$	• Gets *LocVar* of v_1 • Adds all not *Internal Nodes* of $LocVar_{v_1}$ with the field f to *Changed Fields*. • If v_2 is not *null* (not a base type): – Gets *LocVar* of v_2 – Introduces new *Internal Edges* from all nodes of $LocVar_{v_1}$ to all nodes of $LocVar_{v_2}$ with field *[]*
Load $v_1 = v_2.f$	Special handling: invokes method processLoad ($processLoad(v_1, v_2, f)$) which is described after the table.
StaticLoad $v = C.f$	• Gets *LocVar* of v • Removes all nodes from $LocVar_v$ • Adds *Global Node* to $LocVar_v$
ArrayLoad $v_2 = v_1[i]$	Special handling: invokes method processLoad ($processLoad(v_1, v_2, "[]")$) which is described after the table.
If $if\ (...)\ goto\ a_t$	modify nothing (statement is only needed for creating the control flow)

Continued on next page

Stmt Name	Transformations to the *Analyzer Graph*
Call $v_R = v_0.s(v_1, ..., v_j)$	• If the Call statement contains callees (target methods of the invocation), the analyzer does the inter-procedural analysis (see subsection 4.2.3). Theoretically, this is not correct, because the analyzer relies on the fact that either a Call statement contains all possible invoked methods or none. This might no be true, but it is not definite. For example, the analyzer cannot detect if the user has forgotten to pass a class which contains a possible callee. • If the Call statement contains no callees (in the IR tree): – Adds node *Global Node* with field *%native%* to changed fields (because each native call must raise impurity) – Gets *LocVar* of v_R and remove all nodes – Adds *Global Node* to $LocVar_{v_R}$ – For each parameter v_i: ∗ Gets *LocVar* of v_i ∗ Adds all nodes from $LocVar_{v_i}$ to the list of global nodes
Return *return v*	• If v not *null* or *void* (not a return without a variable) – Gets *LocVar* of v – Adds all nodes from $LocVar_v$ to the list of return nodes
ThreadStart *start v*	• Gets *LocVar* of v • Adds all nodes form $LocVar_v$ to the list of global nodes
Entry	Initializes the analyzer graph: • Creates the parameter list: Adds for each parameter of the method a new parameter node to the parameter list if the type of the parameter is not a base type. If the parameter is a base type, adds *null* to the parameter list (has to be done, because the position of a parameter node must match with the index of the corresponding parameter). • Adds for each (non base type) parameter a LocVar to the analyzer graph and adds the corresponding parameter node.
Exit	Does nothing except holding the final analyzer graph, the analyzer graph of the method.

processLoad(String destVar, String srcVar, String srcField):

- Gets LocVars for destVar and srcVar.
- Creates a set of nodes, which are pointed by an internal edge from nodes of $LocVar_{src} \rightarrow srcNodes$.
- Creates a set of nodes from $LocVar_{src}$, which are escaped (reachable form from outside the method) $\rightarrow srcEscaped$.
- Removes all nodes from $LocVar_{dest}$ and adds $srcNodes$.
- If $srcEscaped$ not empty:
  - Creates a new load node and adds it to $LocVar_{dest}$.
  - Creates for each node in $srcEscapted$ an Outside Edge to the new Load Node with field $srcField$.

4.2.3 Inter-procedural Analysis

At a Call statement with possible callees, the analyzer maps all analyzer graphs from the callees into the analyzer graph before the Call execution: The analyzer saves the analyzer graph before Call execution and makes each mapping on this saved graph. Afterwards, the analyzer merges all resulting graphs. The result of the merging is the result of the execution of the Call statement.

The mapping is divided into four steps: Construction of the Node Mapping, Combination of the Analyzer Graphs, Analyzer Graph Simplification, and Modification of the Changed Fields.

Construction of the Node Mapping:

The mapping starts by computing the 'core' mapping μ that disambiguates as many parameter and load nodes from the callee as possible. Example: The mapping $\mu(LNode1) = \{INode2, LNode3\}$ has the following meaning: The node $LNode1$ from the callee is equivalent to two nodes from the caller, $INode2$ and $LNode3$. The technical report[1] introduces the rules for the mapping μ with the following formulas:

$$L(v_i) \subseteq \mu(n^P_{callee,i}),\ \forall i \in \{0, 1, ..., j\} \tag{4.1}$$

$$\frac{\langle n_1, f, n_2\rangle \in O_{callee},\ \ \langle n_3, f, n_4\rangle \in I,\ \ n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} \tag{4.2}$$

$$\frac{\begin{array}{c}\langle n_1, f, n_2\rangle \in O_{callee},\ \ \langle n_3, f, n_4\rangle \in I_{callee},\\ (\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\}) \neq \emptyset,\\ (n_1 \neq n_3) \vee (n_1 \in LNode)\end{array}}{\mu(n_4) \cup (\{n_4\} \setminus PNode) \subseteq \mu(n_2)} \tag{4.3}$$

Constraint 1 maps each parameter node from the callee to the nodes pointed to by the passed variables ($L(v_i)$ = Nodes from the LocVar of v_i).

The handling of **constraints 2** and **3** is a bit more complex. They are executed as a fixpoint iteration as long as any mapping changes:

- **Constraint 2:**
  Handles the case when the callee reads references created by the Call statement. It matches an outside edge ($\langle n_1, f, n_2\rangle$) from the callee against an inside edge ($\langle n_3, f, n_4\rangle$) from the Call statement in the case when n_1 might represent n_3, i.e., $n_3 \in \mu(n_1)$. In this situation, the analyzer adds n_4 to the mappings of n_2, i.e., $n_4 \in \mu(n_2)$.
  Algorithm in the analyzer:
  - Iterates through all outside edges of the callee.
  - Iterates through all inside edges of the Call statement.
  - If the startnode of a inside edge has a mapping to the startnode of the outside edge, it adds the end node of the inside edge to the mappings of the endnode of the outside edge.
- **Constraint 3:**
  Matches an outside edge against an inside edge from the callee. This constraint deals with the aliasing present in the calling context. Consider an outside edge ($\langle n_1, f, n_2\rangle$) and an inside edge ($\langle n_3, f, n_4\rangle$) from the callee. If the start node of the outside edge and the startnode of the inside edge might represent the same node in the callee (if the mappings of n_1 and n_3 have nodes in common, $\mu(n_1) \cap (\mu(n_3) \neq \emptyset)$, then n_2 might be n_4 (i.e. $n_4 \in \mu(n_2)$). Therefore, $n_4 \in \mu(n_2)$, if n_4 is not a parameter node, because the analyzer already knows all the nodes that the parameter nodes stand for. Also, as n_4 is a node from the callee, it might be a node placeholder that represents some other nodes. Therefore, node n_2 might represent not only n_4, it might represent also $\mu(n_4)$. The same reasoning is valid, if n_1 might represent n_3. Therefore, the Constraint has the condition $(\mu(n_1) \cup n_1) \cap (\mu(n_3) \cup n_3) \neq \emptyset$.

The third part of the precondition ($(n_1 \neq n_3) \vee (n_1 \in LNode)$) reduces the applicability of Constraint 3 and avoids fake mappings.
Algorithm in the analyzer:
- Iterates through all outside edges of the callee.
- Iterates through all inside edges of the callee.
- If all the conditions of the Constraint 3 are true, it adds all mappings of the end node of the inside edge and the end node itself, if it is not a parameter node to the mappings of the end node of the outside edge.

The analyzer computes the final mapping μ' by extending the "core" mapping μ with a mapping from each non-parameter node to itself ($\forall n, \mu'(n) = \mu(n) \cup (n \backslash PNode)$. μ' maps nodes form the analyzer from the callee to nodes that appear in the analyzer graph after the Call statement. Inside nodes from the callee are added to their mapping, because when a callee creates an object on the heap, the object exists also for the caller.

Unlike parameter nodes, load nodes are generally not fully disambiguated. Each load node is a placeholder for the nodes that a specific Load instruction loads from an escaped node. That escaped node might remain an escaped node even in the analyzer graph after the Call statement.

Combining the Analyzer Graphs:

After the analyzer obtains the node mapping μ', it uses it to combine the analyzer graph before the execution of the Call statement with the analyzer graphs of the callees. Formally, it is defined with the following equations:

$$IEdges_{CallStmt} = IEdges_{CallStmt} \cup \bigcup_{\langle n_1, f, n_2 \rangle \in IEdges_{callee}} \mu'(n_1) \times \{f\} \times \mu'(n_2)$$

$$OEdges_{CallStmt} = OEdges_{CallStmt} \cup \bigcup_{\langle n, f, n^L \rangle \in OEdges_{callee}} \mu'(n) \times \{f\} \times \{n^L\}$$

$$Nodes(CallStmtLocVar_{v_R}) = \mu'(Nodes(CalleeLocVar_{v_{ret}}))$$

$$EscapedNodes_{CallStmt} = EscapedNodes_{CallStmt} \cup \mu'(EscapedNodes_{Callee})$$

The equations above require some explanation. The heap references that existed before the call might also exists after the call. Hence, all edges persist. In addition, if the callee created a heap edge $\langle n_1, f, n_2 \rangle$ where n_1 may be any node from $\mu'(n_1)$, and n_2 may be any node from $\mu'(n_2)$, then the callee might have created any of the inside edges from the set $\mu'(n_1) \times f \times \mu'(n_2)$. All these edges appear in the analyzer graph after the Call statement.

The outside edges are a bit different. Because the end node of an outside edge is always a load node, the end node could never be newly created in the callee. Because of this, it doesn't have to be mapped. Only the start node of an outside edge has to be mapped.

The LocVar of the variable to which the Call statement passes the return variable contains after the Call Statement only the mappings of the nodes which are saved as return nodes in the analyzer graph of the callee. Finally, the set of globally escaped nodes is the union of the set of the globally escaped nodes of the analyzer graph before the Call Statement and the mappings of the globally escaped nodes of the analyzer graph from the callee.

Analyzer Graph Simplification:

The analyzer simplifies the analyzer graph for the program point after the Call statement by removing all captured (not reachable from the outside, not reachable from a parameter node or globally escaped node) load nodes (together with all adjacent edges), as well as all outside edges that start in a captured node. These rules are very clear because the definition of a load node is that a it represents an object from outside the method. If such an object is captured, it is a case

for the garbage collector! The same for an outside edge that starts in a captured node. An outside edge is a reference from an escaped (opposite of captured) node. So if the node is captured, the reference is for nothing.
Algorithm in the purity analyzer:

- Gets all escaped node from the analyzer graph (all parameters, all nodes which are reachable from parameters over outside edges and all globally escaped nodes).
- Iterate through all outside edges and delete all edges which have a start node that is not in the list of globally escaped nodes and/or an end node that is a load node and not in the list of globally escaped nodes.
- Iterate through all inside edges and delete all edges which have a start and/or a node load which is not in the list of globally escaped nodes.
- Deletes all captured load nodes in local variables.
- Deletes changed fields with captured load nodes.
- Deletes all captured load nodes from the list of return nodes.

Modifiy Changed Fields:

Finally, the most important update, the mapping of the changed fields (W_m). This is done by adding the following set to W_m of the Call Statement:

$$\bigcup_{\langle n,f\rangle} ((\mu'(n) \backslash InsideNode) \cap N) \times \{f\}$$

N is the set of nodes that appear in the simplified analyzer graph. The analyzer uses the mapping μ' to project each node modified by the callee. As usual, it ignores inside nodes. We also use the set intersection "$\cap N$" to ignore nodes that have been removed by the analyzer graph simplification.

Chapter 5

Results of the Analysis

The goal of the purity analyzer is to find out, if a method is pure or not. In the next sections, I explain how the analyzer detects if a method is pure or not and in what form it saves this information in an XML-file. Finally, I describe the three ways of starting the analyzer

5.1 Purity

A method is pure, if its analyzer graph satisfies the following conditions:

- A parameter node or a node which is reachable from parameter nodes over outside edges must not escape globally.
- A parameter node or a node which is reachable from parameter nodes over outside edges must not be changed, meaning, must not be a node in the list of changed fields.
- Notes:
  - A node escapes globally if it is reachable from the set of globally escaped nodes or the global node (the node that represents all static fields).
  - For constructors, the analyzer follows the JML[1] convention of allowing a pure constructor to mutate fields of the ”this” object: it suffices to ignore all changed fields for the parameter node that models the ”this” object.

5.2 XML Output

The purity analyzer has two possible output types. One type is only for writing purity information into an XML-file and the other is to export the internal representation in a simplified form into an XML-file that can be imported by a later analysis (see section 3.2). An example for the first type is the following XML-file:

```
<?xml version=”1.0” encoding=”UTF−8”?>
<ann:annotations xmlns:ann=”http://sct.inf.ethz.ch/annotations”>
    <ann:class name=”ch.ethz.inf.sct.purity_analyzer.test.speech.Data”>
        <ann:method name=”<init>” modifier=”pure”/>
    </ann:class>
    <ann:class name=”ch.ethz.inf.sct.purity_analyzer.test.speech.Iter”>
        <ann:method name=”next” modifier=””/>
        <ann:method name=”<init>” modifier=”pure”/>
        <ann:method name=”remove” modifier=”pure”/>
        <ann:method name=”hasNext” modifier=”pure”/>
    </ann:class>
</ann:annotations>
```

[1] Java Modeling Language http://www.cs.iastate.edu/~leavens/JML/index.shtml

Like the parsing, the output is made with XMLBeans and need no further explanations. The schema for output was not introduced by me; it was introduced for another project that needed purity information as input. The analyzer saves the following elements in the (purity) XML-file:

- classes
  - name
  - methods
    - name
    - all parameters with name, type and index
    - modifier pure if method is pure

For the second output type, I had to make some extensions to the schema. As I said, the analyzer exports a simplified form of the internal representation. This simplified form contains the following elements:

- *classes*
  - *name*
  - is interface
  - array of names of super interfaces
  - name of super class
  - *methods*
    - *name*
    - *all parameters with name, type and index*
    - is constructor
    - is static
    - is abstract
    - *modifier pure if method is pure* (saved here that this file can also be used as purity input)
    - analyzer graph
      - array of inside edges
      - array of outside edges
      - array of local variables
      - array of globals
      - array of changed fields
      - array of return nodes
      - array of parameter nodes

Notes: if a class represents an interface, it doesn't contain methods, and if a method is abstract, it doesn't contain an analyzer graph.

Example for a simplified IR:

```
<?xml version="1.0" encoding="UTF-8"?>
<ann:annotations xmlns:ann="http://sct.inf.ethz.ch/annotations">
  <ann:class name="ch.ethz.inf.sct.purity_analyzer.test.paper.Iterator" isInterface ="true"/>
  <ann:class name="ch.ethz.inf.sct.purity_analyzer.test.paper.Element">
    <ann:method isConstructor="true" isStatic="false" name="&lt;init&gt;" isAbstract="false"
      modifier="pure">
      <ann:parameter name="d" type="java.lang.Object" index="0"/>
      <ann:parameter name="n" type="ch.ethz.inf.sct.purity_analyzer.test.paper.Element"
        index="1"/>
      <ann:ptgraph>
        <ann:edgeinside startnode="P0" field="data" endnode="P1"/>
        <ann:edgeinside startnode="P0" field="next" endnode="P2"/>
        <ann:varlocal varname="d">
          <nodename>P1</nodename>
        </ann:varlocal>
```

```
            <ann:varlocal varname="n">
               <nodename>P2</nodename>
            </ann:varlocal>
            <ann:varlocal varname="this">
               <nodename>P0</nodename>
            </ann:varlocal>
            <changedField field="next" nodename="P0"/>
            <changedField field="data" nodename="P0"/>
            <nodeparam nodename="P0" index="0"/>
            <nodeparam nodename="P1" index="1"/>
            <nodeparam nodename="P2" index="2"/>
         </ann:ptgraph>
      </ann:method>
      <superclass>java.lang.Object</superclass>
   </ann:class>
</ann:annotations>
```

5.3 Starting the Analyzer

All files of the purity analyzer are saved on the 'waldorf.inf.ethz.ch'-CVS-Server in the folder 'dietlw/people/projects/purity-analyzer', which is an Eclipse[2] project. If you check out the project, you must take care that all Eclipse variables are correctly set. The Eclipse variables are used because the purity analyzer contains SWT[3] elements and they are imported over build-path variables. The problem with these variables is that they have different names in each Eclipse version .

I have also created two JARs in the folder 'jars'. The difference between the two files is that the light version doesn't contain the packages which are necessary for the GUI, because they are quite big. Unfortunately, the GUI in the full version works only under Windows, because for starting the GUI, the 'swt-win32-3212.dll'-file must be in the same folder as the JAR-file and this is a Windows library. SWT needs this library, because the most of the toolkit is native. It has the advantage that it is faster and the rendering works better, but the disadvantage is that it is operating system dependent.

The analyzer can be started, respectively the result can be generated in three different ways: Over command line, an imported JAR in a Java project or with a GUI. The three ways have in common that they can only analyze classes/packages which are in the Java Classpath and the names of the classes/packages have to be passed in Java form, e.g. 'java.lang.Object'.

5.3.1 Command Line

There are three possible targets to start the analyzer over a command line:

- The class *ch.ethz.inf.sct.purity_analyzer.Main* with all Classpathes. Because this is cumbersome, I have written a small batch-file which already contains the Classpathes.

  ```
  command.bat <input>
  ```

- One of the two JARs.

  ```
  java -jar purity-analyzer_full.jar <input>
  ```

The input has the following form: '-fulloutput|-normaloutput -classes:{class1,package1} -irs:{address1,address2}'. *-fulloutput* causes an XML-output of the internal representation and

[2] http://www.eclipse.org/
[3] Standard Widget Toolkit http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html

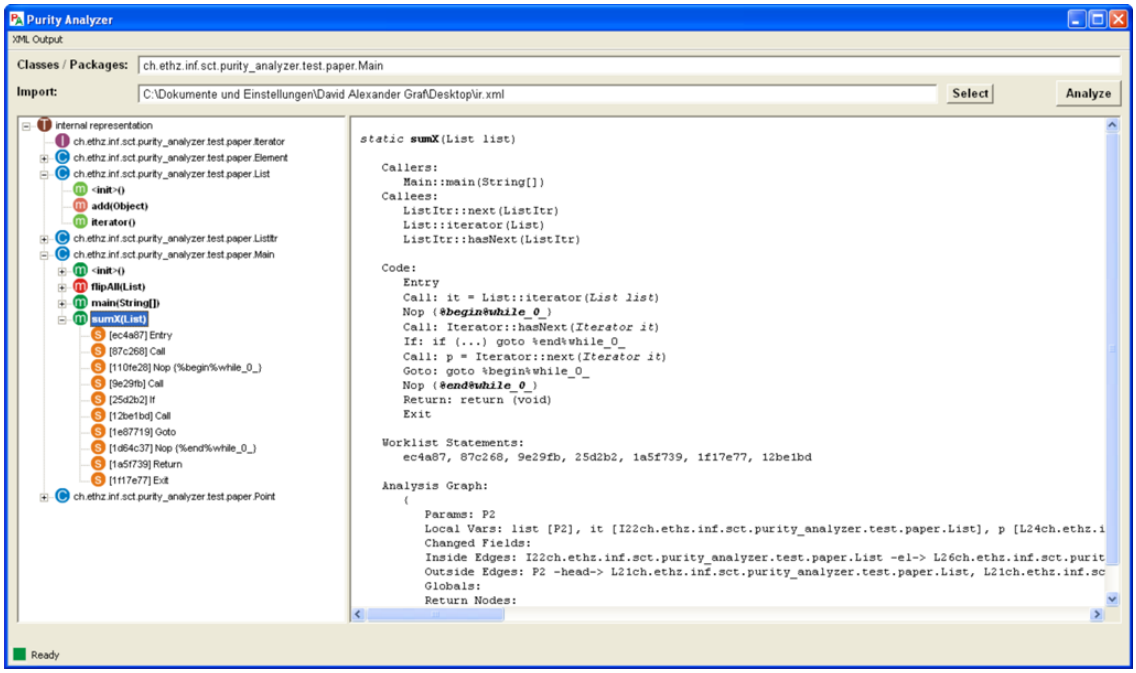

Figure 5.1: GUI

-normaloutput a normal purity XML-output. *-classes* are the classes and packages, that the analyzer analyzes and *-irs* all internal representations in XML-form, which the analyzer imports.

If the analyzer is started over a JAR, all result files are saves in the same folder as the JAR otherwise they are saves in the folder 'print' in the project.

5.3.2 Imported JAR

If one of the JARs is imported to a Java project, the project can start the analyzer by invoking the method 'analyze(String[] classesPackages, String[] imports, boolean normalOutput, String fileAddress)' in the class *ch.ethz.inf.sct.purity_analyzer*. *classesPackages* are the classes and packages that the analyzer analyzes and *imports* all internal representations in XML-form which the analyzer imports. If *normalOutput* is true, the analyzer writes only purity information in the file with the name *fileAddress*, else the simplified internal representation.

5.3.3 GUI

To test and present my purity analyzer, I have implemented a GUI which shows the internal representation. The GUI can be started in Eclipse with the main class *ch.ethz.inf.sct.purity_analyzer.gui.Gui* (make sure that you run it as a SWT Application and not as a Java Application!), with a double click on 'purity-analyzer_full.jar', or with a double click on 'purity-analyzer_light.jar', if all SWT libraries are in the Classpath.

The analysis can be started by writing all classes and packages in the classes/packages field and one IR-XML-file in the import field and click analyze (it is only possible to parse one IR-XML-file, but thats enough for testing and presenting).

The left sub window represents the tree of the internal representation. The elements are **T**ree root (brown), **C**lasses (blue), **I**nterfaces (purple), **m**ethods (green/red) and **s**tatements (orange). The green methods are pure and the reds are not. You may notice that some method signs have a bleached color. These methods are parsed from an XML-file and the other ones from Bytecode.

The numbers that are written next to the statement signs are the hashcode of the objects which represent a statement.

When you click with the mouse on an element, the right window shows some information about this element:

- Tree Root
  - method worklist
- Interface
  - name
  - super interfaces, sub interfaces, super classes
- Class
  - name
  - super interfaces, super class, sub classes
- Bytecode Method
  - name
  - callees and callers
  - code
  - statement worklist (the statements are identified by the hashcode of the representing objects)
  - analyzer graph
- XML-IR Method (light method)
  - name
  - analyzer graph
- Statement
  - signature
  - predecessors and successors (identified by the hashcode of the representing objects)
  - analyzer graph

Chapter 6

Future Work / Conclusion

6.1 Future Work

Right now, it is only possible to feed the analyzer with Bytecode (.class-files) and exported internal representation as XML-file. The analyzer is able to write purity information or simplified, parseable internal representation as output.

Analogous to the Bytecode, it should be possible to feed the analyzer with Java code (.java-files). With a suitable Java code parser, it should be no problem to implement this update.

At the moment, the analyzer builds the analyzer graph only to detect if a method is pure or not. That is fine, but because the analysis extracts a precise representation of the region of the heap that each method may access, it is able to provide useful information even for methods with externally visible side effects:

- Write Effects:
  For each method a regular expression that describes the changes that the method has done to the heap state before method invocation (see section 6.2 in the technical report[1]).
- Read-Only Parameters:
  A parameter p_i is read-only if none of the locations transitively reachable (over outside edges) from p_i is mutated (section 6.3 in the technical report).
- Safe Parameters:
  A parameter is safe if it is read-only and its method does not create any new externally visible heap paths to an object transitively reachable from the parameter (section 6.4 in the technical report).

Additional to this technical extensions, it would be useful to have a graphical view of the analyzer graph. In the current version, it is quite cumbersome to check if an analyzer graph is correct. With pointed nodes and edges, it would be better.

Unfortunately, there is also some future work to do in the current implementation: The problem is that the transformation of the throw instruction from the internal representation of JODE to the IR of the purity analyzer is not correctly handled. Right now, a throw instruction is ignored. Theoretically, a throw instruction is like a method invocation of a catch structure somewhere higher in the call graph. However that is very difficult to handle. Furthermore, the technical report doesn't mention this problem.

6.2 Conclusion

Recognizing method purity is important for a variety of program analysis and understanding task. I present the implementation of the first purity analyzer for Java that is capable of recognizing pure methods that mutate newly allocated objects, including encapsulated objects that do not escape their creating method.

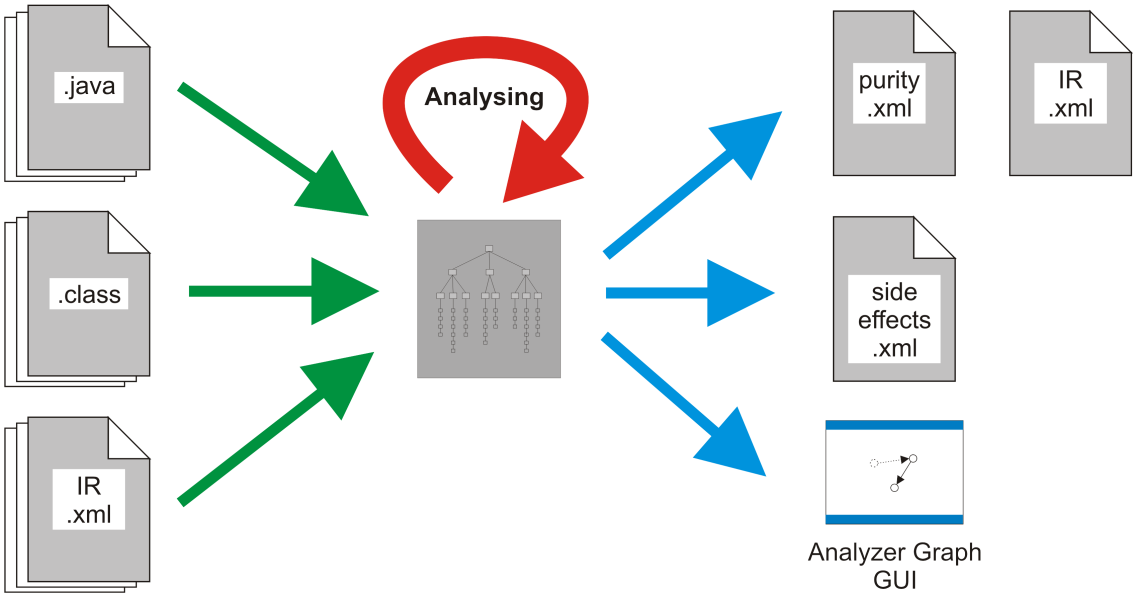

Figure 6.1: Future Work

My purity analyzer is able to analyze each internal representation which comes correctly from JODE in acceptable time (analysis of the analyzer itself lasts 5 seconds). JODE has for example problems with a few classes of the Java runtime environment. When JODE has problems with a class, the class is not inserted into the internal representation, meaning that this class will no be analyzed.

The technical report[1] provides the following about the most important future work: *The most important future work direction concerns making the analysis better suited to the analysis of incomplete programs and libraries, to make this possible, one should have a specification for the missing parts of the program.* With my implementation, it is possible to add specifications for missing parts over the imported IR-XML. It is rather cumbersome to write an analyzer graph into an XML-file by hand, but it is possible.

I have written around forty test classes for my analyzer which are all analyzed correctly. Some interesting cases (like all examples of the technical report[1], examples to check each statement, loops, strongly connected methods, etc.) are located in the package *ch.ethz.inf.sct.purity_analyzer.test*.

Acknowledgement

I would like to thank my supervisor Werner Dietl and Jochen Hoenicke (creator of JODE) for their assistance.

Bibliography

[1] Martin Rinard Alexandru Sălcianu. A combined pointer and purity analysis for java programs. Technical report, Massachusetts Institute of Technology, 2004.

[2] Martin Rinard Alexandru Sălcianu. *A Combined Pointer and Purity Analysis for Java Programs.* PhD thesis, Massachusetts Institute of Technology, 2004.

[3] N. Kellenberger. Static Universe type inference. Master's thesis, ETH Zurich, 2005.

[4] F. Lyner. Runtime Universe type inference. Master's thesis, ETH Zurich, 2005.

[5] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.