NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES
“COMPUTER SYSTEMS: SOFTWARE AND HARDWARE”

MASTER THESIS

Datalog Based Symbolic Program Reasoning for Java

Christos V. Vrachas

Supervisor: Yannis Smaragdakis, Professor NKUA

ATHENS

June 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
“ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συμβολική Συλλογιστική Προγραμμάτων για Java, βασισμένη σε Datalog

Χρίστος Β. Βραχάς

Επιβλέπων: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

Ιούνιος 2019

MASTER THESIS

Datalog Based Symbolic Program Reasoning for Java

Christos V. Vrachas **R.N.:** M1608

SUPERVISOR: **Yannis Smaragdakis,** Professor NKUA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συμβολική Συλλογιστική Προγραμμάτων για Java, βασισμένη σε Datalog

Χρίστος Β. Βραχάς **Α.Μ.:** Μ1608

ΕΠΙΒΛΕΠΩΝ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ

ABSTRACT

abstract english

SUBJECT AREA: Static Program Analysis, Symbolic Reasoning, Propositional Logic

KEYWORDS: flow-sensitivity, path-sensitivity, inference rules and logic proofs

ΠΕΡΙΛΗΨΗ

abstract greek

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατική Ανάλυση Προγραμμάτων, Συμβολική Συλλογιστική, Προτασιακή Λογική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ”ευαισθησία”-ροής, ”ευαισθησία”-μονοπατιού, κανόνες συμπερασμού και αποδείξεις προτασιακής λογικής

This master thesis is dedicated to ...

ACKNOWLEDGEMENTS

I would like to thank ....

June 2019

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SOURCE CODES

1. INTRODUCTION

FooBar

1.1 Thesis Structure

BarBar

2. BACKGROUND

Several program analysis techniques have been proposed in the literature, in order to aid developers and users discover interesting program properties within their software. For example, one might be interested in finding out whether there are memory leaks or whether some piece of code is reachable.

Such techniques come in different flavors, either static or dynamic, and are usually based on strong mathematical concepts. In this section we provide a brief background on these concepts and the frameworks utilized towards the development of program analysis tools.

2.1 Static Program Analysis

Static program analysis is a program analysis technique that aims to reason about a program's behaviors without actually executing it. It has been heavily utilized by optimizing compilers since their early stages and it has also found several other applications, among the areas of softare security, software correctness [11]. The question of whether a program is correct or may terminate for all possible inputs is in general undecidable. However, static analysis techniques are able to tackle undecidability by overapproximating or underapproximating the initial problem, attempting to reason over a simplified version of it.

There is a plethora of different static analysis algorithms in the literature, each one met in several domains. For instance, one may utilize analyses such as *liveness* and/or *pointer* analyses in an optimizing compiler with the intention of eliminating dead code regions or performing a constant propagation/folding optimization. Similarly, a *reachability* analysis that determines whether a specific program point is reachable could be used by a software correctness tool to make sure that an erroneous state is actually never reached.

There is a variety of design choices that may prove essential towards the *scalability* and *precision* of a static program analysis algorithm. We briefly describe some of those choices in the context of this work.

2.1.1 Whole-Program vs. Modular Analyses

Whole-Program analyses need not be confused with *inter-procedural* analyses. An inter-procedural analysis considers multiple functions in order to perform its computations. However, a *whole*-program analysis does not only reason about a program's properties across different functions, but also includes any external dependencies during its computations, such as external libraries or native calls.

On the contrary, a *modular*-analysis reasons about specific parts of a program, disregarding dependencies across different calls and external code. Modular-analysis is fundamentally proportional to *intra*-procedural analyses. In the intra-procedural setting, an analysis would restrict its reasoning within a specific function bound, overlooking the way that function calls or external dependencies affect the computations inside the function. However, someone could also perform a modular-style analysis across a relatively small set of functions, that is an *inter-procedural* analysis.

Whether the analysis would reason about a program's properties in a whole-program or a modular manner is of great importance, because such a decision affects the way that the analysis practitioner would tune its performance - scalability and precision. In the whole-program analysis setting the design tradeoffs between scalability and precision might prove crucial towards the overall analysis reasoning, though a modular-analysis may be able to perform more precise reasoning, due to the relatively restricted area of interest. Whole-program analyses are commonly used in the analysis of languages with complex language features. Those languages, such as the object-oriented ones need to reason about language constructs that may reside on the heap, and thus they may benefit from a whole-program analysis reasoning. Such an analysis may not take into consideration the exact ordering of the instructions of a program, sacrificing precision towards better scalability.

2.1.2 Flow-Sensitivity

The concept of whether an analysis is designed with respect to the instruction ordering is called *flow-sensitivity*. On the contrary, an analysis that is not designed that way is called flow-insensitive. Flow-sensitivity is tightly related with the scalability and precision of static program analyses, due to the fact that a flow-sensitive analysis keeps track of program properties at every point of it, eg. before or after every instruction. Whole-program analyses may usually omit the control-flow constructs during any of their reasoning. However, many whole-program static analyses like the Doop framework's Pointer analysis for Java manage to add flow-sensitivity to their core analysis by slight preprocessing [12].

Several tools utilize state-of-the-art compiler technologies to convert the input source to a lower level *intermediate-representation (IR)*. For instance, there is a plethora of tools that utilize the *LLVM Compiler Infrastructure* [10] for *C*-like languages, or *Soot - A Java optimization framework* [13] - for JVM languages. These tools may further lower the source code in a *Static Single Assignment (SSA)* form. In SSA form, each variable is assigned exactly once, never to be re-assigned again and it is also defined before any of its uses, thus yielding a flow-sensitive representation.

2.1.3 Path-Sensitivity

A path-sensitive analysis is in a way an enhanced version of a flow-sensitive analysis which also considers the path taken up to a specific program point. Such an analysis introduces a path representation usually encoded as the combination of a program's neighboring branch conditional expressions, named *path predicate*. A path predicate is essentially a *Boolean formula*, that is a function of boolean variables whose assignment yields a different control-flow path.

Explicit knowledge of the path taken up to a specific program point allows the analysis to achieve higher precision, ideally eliminating the need for any approximation. However, such knowledge comes at a cost; the number of a program's paths is exponential to the number of branches within the program, thus an analysis that keeps track of all possible paths would not manage to scale. There have been suggested multiple techniques [6], [3] to address such issues in the literature. Some of those techniques manage to achieve scalability by restricting the context of the computations or by introducing loose abstractions of the initial problem. As a static analysis technique a path-sensitive analysis does not eventually execute the program, rather utilizes the path encoding in order to assert properties that hold at specific program points.

One of the adverse effects of pure Static Analysis techniques is that they result a large amount of *false positives* by trading off precision for scalability. In the context of such techniques, a false positive is said to be an erroneous report of a static analysis tool that a property violation has been discovered within the analyzed program, though such violation does not eventually exist. There have been proposed several techniques of either (semi-) dynamic nature that execute directly the program via code instrumentation, or static solutions that attempt to simulate the execution of a program.

2.2 Symbolic Execution

Symbolic Execution is yet another program analysis technique that may be of either static or dynamic flavor and mainly attempts to answer whether certain properties of a program could potentially be violated by a piece of code [1]. At the same time it also results an automated way to generate test cases for programs under testing.

In contrast to the aforementioned static analysis techniques that do not directly execute a program, Symbolic (or *Concolic*) Execution mainly attempts to simulate the execution of a program by considering symbolic (non-deterministic) values for its input. Usually, during the concrete execution of a program a single execution path is cosidered for any computation, whilst a Symbolic Execution engine manages to explore a plethora of paths thanks to the symbolic values assigned to its input variables. As such, a concrete execution is said to under-approximate the desired analysis. *Concolic Execution* is a mixture of symbolic and concrete execution that considers concrete input values when possible, thus making symbolic execution feasible in practice. The latter is usually described as Dynamic Symbolic Execution (DSE). Several tools have been proposed in the literature such as SAGE [7] and KLEE [2], that are considered to be the tools of choice when it comes to binary analysis and/or systems testing.

In the symbolic setting, execution is typically driven by a *symbolic execution engine* which at its core maintains some state for every explored path. The core data-structures of such an engine are the *Boolean formulas* that encode an execution path by considering the satisfied conditions taken for that exact path, and a *symbolic store* that keeps a mapping between the variables met within the path and their equivalent symbolic expressions or values. The former are constructed on each branch execution, while the latter is updated after an assignment to the corresponding variable. Established by those two structures for a path, the engine utilizes an *automated theorem prover* whose purpose is to check whether there is any violation of the property under analysis that would lead to a failed execution, besides verifying path feasibility. A path is said to be feasible in such scenario, if there exists an assignment of concrete values to its symbolic encoding that would yield the satisfiability of its boolean formula. Along with *path-explosion* due to the exponential state space exploration, non-deterministic inputs and several other vulnerabilities, symbolic execution efficiency also relies to the use of efficient external theorem provers. Ongoing research attempts to face those threats in order to provide better tools that aid developers and help discover bugs in programs.

2.3 Automated Theorem Proving

Automated theorem proving, namely the ability to automatically prove a mathematical theorem has been the insatiable desire of the scientific community for an extensive period of time. Theorem proving's establishment is essentially mathematical logic formulas and modelling. On the other hand, *Computer Science* foundations lie within mathematics and logic. Many times software developers want to prove that a specific property is satisfied for a program and its given input values, like its termination. Computer programs can also be encoded as a logical formula and so formal theorems may be constructed in the programming setting too.

The purpose of an automated theorem prover (ATP) is ideally to answer whether a theorem can be proven or not, providing at the same time a formal proof or a counterexample. Automated theorem proving was initially considered to be part of Artificial Intelligence (AI). However, human assistance would be essential in many cases, leading at he same time to the recognition of unable to be proven theorems (such as termination).

Automated theorem proving has lead to the creation of several tools following different approaches to solve the same problem: proof construction. Of particular interest have been satisfiability modulo theories (SMT) provers (or solvers) that are the main tools employed in symbolic execution. SMT is a generalization of the traditional boolean satisfiability problem (SAT) that utilizes underlying decidable mathematical theories in order to encode and give semantical meaning to function predicates instead of boolean (binary) variables. Such theories are the theory of linear arithmetic, bitvectors and arrays and provide a powerful expressiveness to the related solvers which could also be considered as constraint satisfaction engines. Provided a predicate formula encoding, a prover aims to check whether the formula is satisfiable; there is an assignment of values to the compounded predicates. The state of the art tools for theorem proving have been the Z3 Theorem Prover [4], and the Coq Proof Assistant [5], an interactive theorem prover

2.4 Datalog and Doop

2.4.1 Datalog

Datalog is a declarative programming language, essentially a subset of the *Prolog* programming language due to the lack of function symbols, the cut operator among other features. It is also usually considered a query language and has been described as pure SQL enhanhanced with recursion.

Clause ordering does not matter in Datalog, and thus any order of computation is guaranteed to yield the same results, unlike Prolog. Computation in Datalog is established by declaring *input facts* along with *reasoning rules*. Based on those, anything that can be inferred is computed by the underlying engine reaching a convergence state due to the fact that only new facts, that is asserted truth, can be produced by the inference rules. Such rules are called *monotonic* and the associated computation is the *fixed-point*: rules produce new facts until a convergence state is reached.

The main statements of datalog are the inference rules, usually consisting of the *head* and the *body*, or formally `rule`. Both the head and the body may consist of conjunctions (expressed with the comma operator ,) or disjunctions (expressed with the semicolon operator) of predicates, equivalently called relations. Sharing many similarities with databases, a relation can be thought as a database table, whilst input facts and inferred knowledge can be thought as the database rows. The comma operator along with the variable constraints that may form along the operand relations describes what is essentially known as the join operator in the database community.

Datalog has been extensively used both in the academic world and the industry. Several prominent knowledge databases have been designed with Datalog being the underlying engine (i.e. LogicBlox). Datalog also finds use cases in the construction of network specification files, distributed systems, security analysis of smart contracts [8] and static program analysis. One of the static analysis tools that have successfully embodied Datalog to express its analyses is the Doop framework.

2.4.2 The Doop framework

The Doop framework is a static program analysis tool particularly performing pointer analysis for the languages of the Java Virtual Machine (JVM) ecosystem. Late work has also included analysis of Python, and especially TensorFlow programs.

Doop analyses are expressed in the Datalog language in a purely declarative setting and the framework itself is known to be the first to introduce full, end-to-end context-sensitive analyses for JVM based languages. The use of Datalog has lead to the generation of new algorithms for pointer analysis that manage to scale, yet provide accurate results.

Doop at first utilizes the Soot framework to convert the initial program, usually compressed as a Java ARchive (JAR) into Java bytecode. Based on the bytecode version of the program, Doop generates the input facts for the analysis, introducing several relations that can be pre-computed. The framework then includes the Datalog based Souffle tool for static analysis [9] in order to perform any reasoning starting from the provided input facts. That is, Souffle is the tool in which any reasoning happens. Doop analyses have been expressed by a bynch of Datalog rules in Souffle Datalog. Even though several context-sensitive analyses have been expressed, there is not any path-sensitive analysis within the framework, thus it relies on the flow-sensitivity introduced by Soot.

3. STATIC DECLARATIVE SYMBOLIC REASONING

In this chapter we present our approach towards static declarative symbolic reasoning. The main core of the reasoning is encoded as a bunch of Souffle Datalog rules, along with the related input facts. Doop includes a whole of preprocessing, expressive analyses and postprocessing pieces that can be leveraged to enhance the core of our reasoning at relatively swift speed.

3.1 Schema Relations and Types

3.1.1 Input Facts and Types

Doop includes a set of fact generators that provide us with an extensive set of input facts that may be leveraged for our reasoning. Those generators either target different input languages or different intermediate representations (IR). As an example, there are generators that target the Java language, Android technology or Python. Those generators may further utilize different tools that would take care the translation of the input program into the corresponding IR; the current tools being used by Doop are namely Soot and WALA. In the case of Java, the generators ought to produce the same set of input facts that Doop consumes for its main analyses, and thus there is a set of files that define the common predicates-facts that should be produced by the generators. The existing set of generated facts is quite extensive and it is constantly further enriched, yet it does not provide us with a set of facts crucial for the reasoning we introduce. Hosting a rich whole of various pointer analyses, Doop mainly needs knowledge that is related to pointers, or in the Java setting reference types. As such, Doop lacks presence of multiple facts in regard to primitive types or values and in the case that they exist they are relatively non-adequate. In that notion we decided to modify the common fact generator in a way that enhances Doop with facts to be consumed by our symbolic reasoning approach, or modifies several existing ones. At the same time those facts may also come handy for Doop itself in the future. In order to be able to construct a fine-grained analysis we modified a certain amount of facts as follows in figure 1.

```
_AssignOperFrom(instruction, position, from).
_IfVar(instruction, position, var).
```

Figure 1: Modified Doop facts

_AssignOperFrom introduces a new position variable that keeps track of each subsequent operand of an instruction’s right-hand side (rhs), whilst the same also holds for _IfVar too. Next, we provide a glimpse of all the newly introduced generated facts in figure 2.

```
_OperatorAt(instruction, operator).
_DummyIfVar(instruction, var).
_AssignOperFromConstant(instruction, position, from).
_IfConstant(instruction, position, constant).
```

Figure 2: New Doop facts

Those input facts share many similarities in regards to the associated variables - relation columns - though each one encodes crucial knowledge. The _OperatorAt relation holds facts that encode the operation being present at a given instruction. _AssignOperFromConstant is an alternate rule of _AssignOperFrom, whilst _IfConstant is equivalent to _IfVar. A constant type is different to a variable type in the context of the framework, thus the need for a more detailed differentiation between the two. Such differentiations have extensively been made during the development of our approach, but in many cases will be omitted in the context of this text. Lastly, the _DummyIfVar relation has been introduced in order to encode the conditional expressions at a given branch instruction, that is quite necessary to our implementation and its need is evoked due to the IR provide by Soot. It is also worth mentioning that the facts and the associated relations as described above are provided by the generators written in Java. Mere preprocessing takes place before importing the final facts into Doop's knowledge base in order to follow its conventions. That preprocessing leads to the following renames as described in figure 3.

```
AssignOper_From(instruction, position, from).
AssignOper_FromConstant(instruction, position, from).
If_Var(instruction, position, var).
If_Constant(instruction, position, constant).
DummyIf_Var(instruction, var).
Operator_At(instruction, operator).
```

Figure 3: Imported facts

The relations described thus far are only a small set of the input to our reasoning, as we have made use of the various relations and rules that were already existed in Doop. Here we have only presented the brand new input relations or the ones that we modified. However we can not provide an extensive presentation of all, thus we will briefly describe them in the following sections, in the context of the rules that they are used.

Our symbolic reasoning approach namely reasons about a program's possible expressions. That said, there is a need for the representation of a program's expression, either primitive ones or not. Doop does not include such an encoding, due to the fact that its analyses mainly reason about pointers. In order to address that and also lay the foundations of our approach, we have introduced the following types as described in figure 4.

```
.type SymbolicInput = Var | MethodInvocation | NumConstant
.type Operator = symbol
.type ExpressionType = PrimitiveType | ReferenceType
.type Base = SymbolicInput | Operator
.type Expr = [
  base: Base ,
  type: ExpressionType ,
  left: Expr ,
  right: Expr
]
```

Figure 4: Expression Type

The types that are defined using a (|) symbol are said to inherit from the associated types, whilst the latter definition introduces a new record type. The expression type manages to represent both unary, binary and constant expressions at the same time, by distinguishing each case with the help of the *base* field. If base is of type *SymbolicInput* then the latter *Expr* fields are nil, whilst in the case of base being of *Operator* at least the first of the Expr fields is indeed an expression. For each distinct expression we also preserve its type, which can either be *PrimitiveType* or *ReferenceType*. Those types are the building blocks for the construction of the program's expression. It may come natural to the reader that the encoding leads to complex *expression trees*. This expression type is essentially what powers our declarative symbolic reasoning approach, encoding both the primitive expressions of the input program and its control flow constructs at the same time.

3.1.2 Relations

We here provide a brief presentation of the *main* declared relations that make our approach work in figure 5.

```
.decl  ResolveExpr(meth: Method, var: symbol, expr: Expr)
.decl  isExpr(expr: Expr)
.decl  isArithmeticExpr(expr: Expr)
.decl  isReferenceExpr(expr: Expr)
.decl  isBooleanExpr(expr: Expr)
.decl  isBooleanExprLeftRight(exprOther: Expr, exprX: Expr, exprY: Expr,
                               op: Operator)
.decl  isBooleanExprLeftRightInMethod(exprOther: Expr, exprX: Expr, exprY: Expr,
                                       op: Operator, meth: Method)
.decl  BuildPathExprBefore(meth:Method, prev:Instruction, exprBase: Expr,
                            insn:Instruction)
.decl  PathExpressionBefore(meth:Method, insn: Instruction, pathExpr: Expr)
.decl  PathExpressionAfterTrue(meth: Method, insn: Instruction, pathExpr: Expr)
.decl  PathExpressionAfterFalse(meth: Method, insn: Instruction, pathExpr: Expr)
.decl  ExprImpliesOther(expr: Expr, exprOther: Expr)
.decl  ExprIsAlwaysTrue(expr: Expr)
.decl  ExprIsAlwaysFalse(expr: Expr)
.decl  ExprIsNegationOfOther(expr: Expr, exprOther: Expr)
```

Figure 5: Main Symbolic Reasoning relations

Those relations and their associated rules are going to be thoroughly described in the following section where we will be introducing the main parts of *declarative symbolic reasonig*. *ResolveExpr* is the main relation that constructs the program's expressions that yield from the input statements and expressions, whilst the *PathExpression** relations are responsible for the construction of the control flow expressions, that is essentially encoding the branches taken up to as specific program point. Lastly, our declarative symbolic reasoner is powered by the rules of *ExprImpliesOther*, *ExprIsAlwaysTrue*, *ExprIsAlwaysFalse* and *ExprIsNegationOfOther* relations, attempting to prove *implications* between the expressions and assert those that hold *true* or *false* equivalently, if possible.

3.2 Program Expression Trees

Introducing a Souffle type that represents a program's expressions is only one step towards symbolic reasoning of a program. For our reasoner to be able to provide us with meaningful results we have to generate its world of expressions. To that end, we have introduced several rules that build expression trees from the given program's input relations which essentially encode the expressions of a program in a symbolic manner. The first set of those rules originate from the *ResolveExpr* relation that was declared in figure 5 of the previous section.

The ResolveExpr rules are divided into two sets of rules. The ones that encode the base, *symbolic input* expressions and those that lead to the construction of the complex composite expressions. Symbolic input expressions namely represent the symbolic inputs of a method. We consider (a) method parameters, (b) method invocation results, (c) instance and static fields loading, (d) numeric constant assignments and (e) phi assignments as symbolic inputs. Those input cases form the basis of our expressions, and it also worts mentioning that our reasoner mainly asserts any new knowledge in an intra-procedural fashion. The rules for the enumerated symbolic input expressions are presented in the following figures.

```
ResolveExpr(?meth, ?var, ?expr) :-
isAssignLocal_Insn(?insn),
AssignLocal_From(?insn, ?param),
FormalParam(_, ?meth, ?param),
Var_Type(?param, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
AssignInstruction_To(?insn, ?var),
Instruction_Method(?insn, ?meth),
?expr = [?param, ?type, nil, nil] .
```

Figure 6: (a) method parameter

```
ResolveExpr(?meth, ?var, ?expr) :-
isMethodInvocation(?insn),
MethodInvocation_Method(?insn, ?sig),
Method_ReturnType(?sig, _, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
Instruction_Method(?insn, ?meth),
AssignReturnValue(?insn, ?var),
?expr = [?param, ?type, nil, nil] .
```

Figure 7: (b) method invocation results

```
ResolveExpr(?meth, ?var, ?expr) :-
isLoadInstanceField_Insn(?insn),
LoadInstanceField_To(?insn, ?var),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
Instruction_Method(?insn, ?meth),
?expr = [?var, ?type, nil, nil] .
```

Figure 8: (c) load instance fields

```
ResolveExpr(?meth, ?var, ?expr) :-
isLoadInstanceField_Insn(?insn),
LoadInstanceField_To(?insn, ?var),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
Instruction_Method(?insn, ?meth),
?expr = [?var, ?type, nil, nil] .
```

Figure 9: (d) load static fields

The various relations used in the rules bodies are quite self-explanatory which proves the expressiveness and high quality of Doop's code. In the following figures we present the recursive rules that apply for the cases of (a) local variable assignments, (b) unary assignments, (c) binary expression assignments and (d) if conditions.

For simplicity, several details have been omitted from the rules as defined in figure 13 and they have been hidden under the *FUNCTION* relation. This function relation is demonstration purposed only and it is responsible to emit the appropriate variables and generate

```
ResolveExpr(?meth, ?var, ?expr) :-
isAssignNumConstant_Insn(?insn),
AssignInstruction_To(?insn, ?var),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
Instruction_Method(?insn, ?meth),
AssignNumConstant_Id(?insn, ?const),
?expr = [?const, ?type, nil, nil] .
```

Figure 10: (e) numeric assignments

```
ResolveExpr(?meth, ?var, ?expr) :-
Instruction_Method(?insn, ?meth)
isAssignLocal_Insn(?insn),
AssignInstruction_To(?insn, ?var),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
c = count: AssignInstruction_To(_, ?var),
c > 1, ?expr = [?var, ?type, nil, nil] .
```

Figure 11: (f) phi assignments

```
ResolveExpr(?meth, ?var, ?expr) :-
Instruction_Method(?insn, ?meth)
isAssignLocal_Insn(?insn),
AssignInstruction_To(?insn, ?var),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
c = count: AssignInstruction_To(_, ?var),
c = 1, AssignLocal_From(?insn, ?from),
ResolveExpr(?meth, ?from, ?expr).
```

```
ResolveExpr(?meth, ?var, ?expr) :-
isAssignUnop_Insn(?insn),
AssignInstruction_To(?insn, ?var),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
Instruction_Method(?insn, ?meth),
AssignOper_From(?insn, _, ?right),
_OperatorAt(?insn, ?op),
ResolveExpr(?meth, ?right, ?rExpr),
?expr = [?op, ?type, ?rExpr, nil] .
```

Figure 12: (a) local variable and (b) unary assignments

```
ResolveExpr(?meth, ?var, ?expr) :-
FUNCTION(?var, ?right1, ?right2, ?op),
Var_Type(?var, ?type),
(isPrimitiveType(?type) ;
isReferenceType(?type)),
ResolveExpr(?meth, ?right1, ?lExpr),
ResolveExpr(?meth, ?right2, ?rExpr),
?expr = [?op, ?type, ?lExpr, ?rExpr] .
```

```
ResolveExpr(?meth, ?var, ?expr) :-
FUNCTION(?var, ?right1, ?right2, ?op),
ResolveExpr(?meth, ?right1, ?lExpr),
ResolveExpr(?meth, ?right2, ?rExpr),
?expr = [?op, "boolean", ?lExpr,
?rExpr] .
```

Figure 13: (c) binary assignments and (d) if conditions

their corresponding expressions in either case of binary expression assignment or if conditions for the rules to work. It essentially attempts to generate the appropriate combination of operands and expressions based on whether an operand is a constant or a variable.

As previously mentioned, the rules presented in this section constitute one of the most fundamental parts of our reasoner. They construct the most primitive, yet essential expressions of the anlyzed program. However, the expressions are guarded in a sense that if unrolled, they would all lead to complex expressions between symbolic expressions. We shall mention here that any reasoning is purely symbolic, and thus there is not any concrete evaluation of the expressions in the sense of constant folding. One could also notice that even though the lack of boolean expressions up to this point. That is due to the SSA IR upon which any reasoning happens. The various branches, however complex they are are split into multiple simplified ones and thus we have to kee track of these. In the following section we provide an overview of the algorithm responsible for generating those boolean expressions that encode the control flow construcs of the methods of the input program.

3.3 Path Expressions

Any Doop reasoning happens on facts produced at a lower level IR called Jimple, Soot's IR. Jimple omits several complex constructs, such as boolean expressions. In our case though, we need to have an exact representation of those high level control flow constructs in order to be able to reason about a program and answer several useful questions about it. In order to further facilitate and enhance our core symbolic reasoning technique we have introduced several algoritms that are responsible for the reconstruction of those expressions, that namely represent the program's branch conditions or complete path predicates.

Even though the analyzed program is quite simplified and the various complex control flow structures have been lowered to way simpler forms, the reconstruction of the desired predicates is not trivial at all. Complex expressions between operands that contain operators such as $\&\&$, $||$ and ! have been simplified to three-address code (TAC) that completely trim those operators in favour of if statements with simplified boolean conditions. Even though we could modify the code transformer that lowers the initial program in a way that would avoid those transformations, we decided to preserve any external framework functionality as it is, implementing any predicate reformation in pure Datalog. Following we introduce the algorithm that implements the desired funcionality in two parts. For the shake of pure logic formality we consider $\wedge$, $\vee$ and $\neg$ the equivalents of *and*, *or* and *negation* operators.

At first we provide a high level overview of the algorithms responsible for the restoration of the boolean predicates. The basis of each predicate is the first boolean expression met in the program, that is usual the first branch instrucion. The algorithm keeps track of the path predicates at any program point before and after each instruction. Each branch instruction splits two distinct path predicates, whilst any other instruction does not produce any new path predicate but rather emits the predicate that holds at the program point before it. However, an instruction may have more than a single predecessors that lead the execution path up to the program point before it. The decision to consider only a single predecessor would lead to inaccurate results as several program paths would have been lost. In order for our algorithm to yield sound path predicates we join the expressions that hold after each predecessor at the program point before such an instruction. Recall that any reasoning is symbolic and there is not any cocrete evaluation, thus we can not be sure of what path predicates should be discarded. This brief introduction reveals that our algorithm needs to implement the following two functionalities: (a) the enumeration of all possible predecessors of an instruction and (b) the making of the path predicates for every program point.

In the following figures we are going to introduce the algorithmic parts responsible for the instruction predecessors enumeration. The whole logic is implemented in pure Datalog rules like the rest of our work. Doop contains a set of rules populating relations that represent various control flow constructs, such as the set of basic blocks in each function which we made use of. The simplest case of this enumeration is the single predecessor case. We present the relations and the associated rules for this one, along with the *helper* rules that populate the relations which keep track of the first and last predecessor of an instruction in the following figures.

The above rules introduce the handling of the least complicated cases. With just a few rules we manage to identify the one and only predecessor of an instruction. In figure 14 we present the rule that populates the relation keeping track of the single predecessors of an instruction which is powered by the rules of FirstPredOfInsnInMethod and LastPre-

```
SinglePredOfInsnInMethod(?pred, ?insn, ?meth) :-
   Instruction_Method(?insn, ?meth),
   FirstPredOfInsnInMethod(?meth, ?pred, ?insn),
   LastPredOfInsnInMethod(?meth, ?pred, ?insn).
```

Figure 14: Single predecessor rules.

```
FirstPredOfInsnInMethodOrdinal(?meth, ?firstOrd, ?insn) :-
   Instruction_Method(?insn, ?meth),
   ?firstOrd = min ord(?prev): MayPredecessorModuloThrow(?prev, ?insn).

FirstPredOfInsnInMethod(?meth, ?first, ?insn) :-
   Instruction_Method(?insn, ?meth),
   MayPredecessorModuloThrow(?first, ?insn),
   ?min_ord = ord(?first),
   FirstPredOfInsnInMethodOrdinal(?meth, ?min_ord, ?insn).
```

Figure 15: First predecessor of instruction in method

```
LastPredOfInsnInMethodOrdinal(?meth, ?lastOrd, ?insn) :-
   Instruction_Method(?insn, ?meth),
   ?lastOrd = max ord(?prev): MayPredecessorModuloThrow(?prev, ?insn).

LastPredOfInsnInMethod(?meth, ?last, ?insn) :-
   Instruction_Method(?insn, ?meth),
   MayPredecessorModuloThrow(?last, ?insn),
   ?max_ord = ord(?last),
   LastPredOfInsnInMethodOrdinal(?meth, ?max_ord, ?insn).
```

Figure 16: Last predecessor of instruction in method

dOfInsnInMethod relations. In those two presented in figures 15 and 16 we leverage Doop's control-flow graph (CFG) analysis by using the MayPredecessorModuloThrow relation that provides us with the predecessors of an instrucion. We further aggregate with the help of min and max aggregation functions of Souffle in order to distinguish the first and last predecessors of an instruction accordingly. Those rules are also utilized during the enumeration of the multiple predecessors of a single instruction, but they are wrapped into complex rules that separate the multiple to the single predecessor case. We present those wrappers in figure 17.

```
FirstOfMultiplePredsOfInsnInMethod(?pred, ?insn, ?meth) :-
   Instruction_Method(?insn, ?meth),
   FirstPredOfInsnInMethod(?meth, ?pred, ?insn),
   !LastPredOfInsnInMethod(?meth, ?pred, ?insn).
LastOfMultiplePredsOfInsnInMethod(?pred, ?insn, ?meth) :-
   Instruction_Method(?insn, ?meth),
   !FirstPredOfInsnInMethod(?meth, ?pred, ?insn),
   LastPredOfInsnInMethod(?meth, ?pred, ?insn).
```

Figure 17: First and last of multiple predecessors wrappers

The negation of LastPredOfInsnInMethod in the first rule is required so that the FirstOfMultiplePredsOfInsnInMethod relation considers instructions with multiple predecessors. Otherwise it would be equivalent to the SinglePredOfInsnInMethod relation, whilst the same also holds for the LastOfMultiplePredsOfInsnInMethod. Having introduced the first and last predecessors rules for either case, there is the need of introducing the rules of the relations associated with yielding an ordering of each distinct predecessor. Datalog does not come with any explicit iteration syntax, thus we need to come with a way to enumerate those possible predecessors in a recursive style. We present the steps towards the multiple predecessors enumeration in figure 18.

```
NotNextPredOfInsnInMethod(?meth, ?prev, ?next, ?insn) :-
   Instruction_Method(?insn, ?meth),
   MayPredecessorModuloThrow(?prev, ?insn),
   MayPredecessorModuloThrow(?next, ?insn),
   MayPredecessorModuloThrow(?nextPossible, ?insn),
   ord(?prev) < ord(?next),
   ord(?prev) < ord(?nextPossible),
   ord(?nextPossible) < ord(?next).

NextPredOfInsnInMethod(?meth, ?prev, ?next, ?insn) :-
   Instruction_Method(?insn, ?meth),
   MayPredecessorModuloThrow(?prev, ?insn),
   MayPredecessorModuloThrow(?next, ?insn),
   ord(?prev) < ord(?next),
   !NotNextPredOfInsnInMethod(?meth, ?prev, ?next, ?insn).

NextOfMultiplePredsOfInsnInMethod(?next, ?prev, ?insn, ?meth) :-
   Instruction_Method(?insn, ?meth),
   !LastPredOfInsnInMethod(?meth, ?prev, ?insn),
   NextPredOfInsnInMethod(?meth, ?prev, ?next, ?insn).
```

Figure 18: Predecessors ordering rules

The last rule in figure 18 is yet again a wrapper which is used for the predecessors enumeration. It populates the associated relation excluding the last predecessor. Even though there are many different ways to tackle this problem, we decided to provide an interim relation which when negated yields the next predecessor in the enumeration. The rule feeding that relation, named NotNextPredOfInsnInMethod, considers the previous and next possible predecessor of an instruction and asserts that in case there is another one between them the rule shall fail. Thus, the negation of this relation yields the actual next possible predecessors which are then used to proceed with the enumeration.

After introducing the part responsible for the predecessors enumeration, we may proceed to the main algorithm that creates the path predicates at every single program point. A high level overview of the algorithm has been provided earlier, and we now present the core rules of this part in the following figures. We have already introduced the notion of *before* and *after* program points in the context of a distinct instruction and at first we describe the rules responsible for the construction of the path predicates after an instruction via the PathExpressionAfterTrue and PathExpressionAfterFalse relations.

The path expression predicates are split into two dsjoint sets of *true* and *false* conjuncts due to the separation of a program path after a branch instruction. A branch instruction leads to different execution paths based on whether the condition succeeds or not. In order to differentiate between the two, we have introduced the NegationOfConditionAtIfInsn relation that yields the negated condition of a branch condition that would be used to gen-

```
PathExpressionAfterTrue(?meth, ?insn, ?cond),
PathExpressionAfterFalse(?meth, ?insn, ?negatedCond) :-
   FirstIfInsnInMethod(?insn, ?meth),
   NegationOfConditionAtIfInsn(?cond, ?negatedCond, ?insn).

PathExpressionAfterTrue(?meth, ?insn, ?pathExpr1),
PathExpressionAfterFalse(?meth, ?insn, ?pathExpr2) :-
   PathExpressionBefore(?meth, ?insn, ?pathExpr),
   NegationOfConditionAtIfInsn(?cond, ?negatedCond, ?insn),
   ?cond = [?op, ?type, ?exprLeft, ?exprRight] ,
   ?pathExpr1 = ["&&", ?type, ?cond, ?pathExpr] ,
   ?pathExpr2 = ["&&", ?type, ?negatedCond, ?pathExpr] .

PathExpressionAfterTrue(?meth, ?insn, ?pathExpr) :-
   PathExpressionBefore(?meth, ?insn, ?pathExpr),
   !isIf_Insn(?insn).
```

Figure 19: Path expressions after an instruction

erate the complement of a path predicate. The first rule in figure 19 represents the basis of any path predicate formation, whist the second one further splits each path predicate into two subsequent expressions. The last rule represents the path predicate after any statement that is not a branch instruction. However, those rules heavily depend on the existence of path predicates on the program point right before each instruction. In figures 20 we present the rules of the relations that yield the path predicates at the program point before each instruction for the single predecessor case.

```
PathExpressionBefore(?meth, ?insn, ?expr) :-
   PathExpressionAfterTrue(?meth, ?pred, ?expr),
   SinglePredOfInsnInMethod(?pred, ?insn, ?meth),
   isIf_Insn(?pred), isJumpTarget(?insn).

PathExpressionBefore(?meth, ?insn, ?expr) :-
   PathExpressionAfterFalse(?meth, ?pred, ?expr),
   SinglePredOfInsnInMethod(?pred, ?insn, ?meth),
   isIf_Insn(?pred), !isJumpTarget(?insn).

PathExpressionBefore(?meth, ?insn, ?expr) :-
   PathExpressionAfterTrue(?meth, ?pred, ?expr),
   SinglePredOfInsnInMethod(?pred, ?insn, ?meth),
   !isIf_Insn(?pred).
```

Figure 20: Path expressions before an instruction - single predecessor

The complicated case of the construction of the before-path predicates is essentially when an instruction has multiple predecessors that flow up to it. In order to facilitate the rules of the relation PathExpressionBefore, we introduced the BuildPathExprBefore relation, responsible for constructing a disjunct of all the predecessor path expressions that meet at the program point right before the instruction. The former relation rules utilize the predecessor enumeration relations introduced before, by implicitly enumerating all over the predecessors. Lastly, we present the logic of the disjunct formation in figure 21.

```
PathExpressionBefore(?meth, ?insn, ?pathExpr) :-
   LastOfMultiplePredsOfInsnInMethod(?pred, ?insn, ?meth),
   BuildPathExprBefore(?meth, ?pred, ?pathExpr, ?insn).
BuildPathExprBefore(?meth, ?pred, ?expr, ?insn) :-
   FirstOfMultiplePredsOfInsnInMethod(?pred, ?insn, ?meth),
   isIf_Insn(?pred),
   IsJumpTarget(?insn),
   PathExpressionAfterTrue(?meth, ?pred, ?expr).
BuildPathExprBefore(?meth, ?pred, ?expr, ?insn) :-
   FirstOfMultiplePredsOfInsnInMethod(?pred, ?insn, ?meth),
   isIf_Insn(?pred),
   !IsJumpTarget(?insn),
   PathExpressionAfterFalse(?meth, ?pred, ?expr).
BuildPathExprBefore(?meth, ?pred, ?expr, ?insn) :-
   FirstOfMultiplePredsOfInsnInMethod(?pred, ?insn, ?meth),
   !isIf_Insn(?pred),
   PathExpressionAfterTrue(?meth, ?pred, ?expr).
BuildPathExprBefore(?meth, ?next, ?pathExpr, ?insn) :-
   BuildPathExprBefore(?meth, ?prev, ?expr, ?insn),
   NextOfMultiplePredsOfInsnInMethod(?next, ?prev, ?insn, ?meth),
   !isIf_Insn(?next),
   PathExpressionAfterTrue(?meth, ?next, ?exprPrev),
   ?pathExpr = ["||", "boolean", ?expr, ?exprPrev] .
BuildPathExprBefore(?meth, ?next, ?pathExpr, ?insn) :-
   BuildPathExprBefore(?meth, ?prev, ?expr, ?insn),
   NextOfMultiplePredsOfInsnInMethod(?next, ?prev, ?insn, ?meth),
   isIf_Insn(?next),
   isJumpTarget(?insn),
   PathExpressionAfterTrue(?meth, ?next, ?exprPrev),
   ?pathExpr = ["||", "boolean", ?expr, ?exprPrev] .
BuildPathExprBefore(?meth, ?next, ?pathExpr, ?insn) :-
   NextOfMultiplePredsOfInsnInMethod(?next, ?prev, ?insn, ?meth),
   BuildPathExprBefore(?meth, ?prev, ?expr, ?insn),
   isIf_Insn(?next),
   !isJumpTarget(?insn),
   PathExpressionAfterTrue(?meth, ?next, ?exprPrev),
   ?pathExpr = ["||", "boolean", ?expr, ?exprPrev] .
```

Figure 21: Path expression before multiple predecessors

All of the rules introduced in this section generate the boolean logic formulas, disjuncts and conjuncts. Those formulas are modeled with the help of the expression type introduced during the first sections of this thesis. We may now proceed to the description of the core algorithm of our analysis, static declarative symbolic reasoning.

3.4 Boolean Symbolic Reasoning

A symbolic reasoner mainly deduces knowledge about the expressions of a program and may get as complex as one wants to. Such tool is essentially a theorem prover that given a friendly input representation and a set of guiding facts tries to reason about a program by constructing formal proofs with the help of a set of inference rules. In the context of this work we have introduced a fundamental symbolic reasoner that mainly deduces knowledge about the boolean expressions of a program in a purely symbolic manner. That is, our reasoner contains pure propositional logic formulas and it does not concretely evaluate any expression on the fly, but rather constantly emits new knowledge until a fixpoint is reached: there can be no more knowledge produced by the set of inference rules. In this chapter we exhibit the core implementation of our reasoner by describing the set of propositional logic axioms and inference rules that bring into life our technique. The whole of those axioms and rules are implemented as pure Datalog rules, without the help of any external tool. The main work of the reasoner is to ask the question of *"What are the program expressions that may be implied by another program expression?"*.

The main relation of the core reasoning algorithm is the ExprImpliesOther relation. Formally, this relation represents the following logical statement:

$$P \rightarrow Q \tag{3.1}$$

where P and Q represent any logical (boolean) formula.

3.4.1 Propositional Logic Axioms

We begin with the enumeration of the axioms that constitute the basis of our reasoner. In the following figures we present the Datalog rules describing those axioms.

```
ExprImpliesOther(?expr, ?expr) :-
   isBooleanExpr(?expr).
```

Figure 22: Self implication

```
ExprImpliesOther(?expr, ?exprLeft),
ExprImpliesOther(?expr, ?exprRight) :-
   isBooleanExprLeftRightInMethod(?expr, ?exprLeft, ?exprRight, "&&", ?meth).
```

Figure 23: $A \wedge B \rightarrow A$ **and** $A \wedge B \rightarrow B$

```
ExprImpliesOther(?exprLeft),
ExprImpliesOther(?expr, ?exprRight) :-
   isBooleanExprLeftRightInMethod(?expr, ?exprLeft, ?exprRight, "&&", ?meth).
```

Figure 24: $A \rightarrow A \vee B$ **and** $B \rightarrow A \vee B$

```
ExprImpliesOther(?expr, ?exprOther) :-
   isBooleanExprLeftRightInMethod(?expr, ?exprLeft, ?exprRight, "&&", ?meth),
   isBooleanExprLeftRightInMethod(?exprOther, ?exprRight, ?exprLeft, "&&", ?meth).
```

Figure 25: $A \wedge B \rightarrow B \wedge A$

```
ExprImpliesOther(?expr, ?exprOther) :-
   isBooleanExprLeftRightInMethod(?expr, ?exprLeft, ?exprRight, "||", ?meth),
   isBooleanExprLeftRightInMethod(?exprOther, ?exprRight, ?exprLeft, "||", ?meth).
```

Figure 26: $A \lor B \rightarrow B \lor A$

```
ExprImpliesOther(?expr, ?exprOther) :-
   isBooleanExprLeftRightInMethod(?expr, ?exprLeft, ?exprRight, "&&", ?meth),
   ?exprOr = ["||", "boolean", ?exprB, ?exprC] ,
   ?exprLeft = ["&&", "boolean", ?exprA, ?exprB] ,
   ?exprRight = ["&&", "boolean", ?exprA, ?exprC] ,
   ?exprOther = ["||", "boolean", ?exprLeft, ?exprRight] .
```

Figure 27: $A \land (B \lor C) \rightarrow (A \land B) \lor (A \land C)$

```
ExprImpliesOther(?expr, ?exprOther) :-
   isBooleanExprLeftRightInMethod(?expr, ?exprLeft, ?exprRight, "&&", ?meth),
   ?exprLeft = ["&&", "boolean", ?exprA, ?exprB] ,
   ?exprRight = ["&&", "boolean", ?exprA, ?exprC] ,
   ?exprNRight = ["||", "boolean", ?exprB, ?exprC] ,
   ?exprOther = ["&&", "boolean", ?exprA, ?exprNRight] .
```

Figure 28: $(A \land B) \lor (A \land C) \rightarrow A \land (B \lor C)$

```
ExprImpliesOther(?expr, ?exprOther) :-
   ?exprBC = ["&&", "boolean", ?exprB, ?exprC] ,
   isBooleanExprLeftRightInMethod(?expr, ?exprA, ?exprAB, "||", ?meth),
   ?exprLeft = ["||", "boolean", ?exprA, ?exprB] ,
   ?exprRight = ["||", "boolean", ?exprA, ?exprC] ,
   isBooleanExprLeftRightInMethod(?exprOther, ?exprLeft, ?exprRight, "&&", ?meth).
```

Figure 29: $A \lor (B \land C) \rightarrow (A \lor B) \land (A \lor C)$

```
ExprImpliesOther(?expr, ?exprOther) :-
   ?exprAB = ["||", "boolean", ?exprA, ?exprB] ,
   ?exprBC = ["||", "boolean", ?exprB, ?exprC] ,
   isBooleanExprLeftRightInMethod(?expr, ?exprAB, ?exprBC, "&&", ?meth),
   ?exprRight = ["&&", "boolean", ?exprB, ?exprC] ,
   isBooleanExprLeftRightInMethod(?exprOther, ?exprA, ?exprRight, "&&", ?meth).
```

Figure 30: $(A \lor B) \land (A \lor C) \rightarrow A \lor (B \land C)$

3.4.2 Propositional Logic Inference Rules

The axioms described before form the ground truth for the reasoner. However, for the reasoner to infer new knowledge we have to define a set of inference rules. An inference rule is the actual means of proving. Based on any number of logical premises an inference rule infers one or maybe even more conclusions.

The formal definition of a logical inference rule that takes only two premises is the following:

$$P, P \rightarrow Q \vdash Q \tag{3.2}$$

where P and Q are metavariables which are essentially logical sentences that may consist of multiple logical statements.

An equivalent definition of the equation 3.2, also known as the *modus ponens* is:

$$\frac{P, P \rightarrow Q}{Q} \tag{3.3}$$

Based on the definitions above we introduce a set of inference rules that are the ones responsible to power up our analysis.

```
ExprImpliesOther(?expr, ?exprOther) :-
   ExprImpliesOther(?expr, ?exprInter),
   ExprImpliesOther(?exprInter, ?exprOther).
```

Figure 31: $A \rightarrow B$ **and** $B \rightarrow C \vdash A \rightarrow C$

```
ExprImpliesOther(?exprA, ?exprOther) :-
   isBooleanExprLeftRightInMethod(?exprOther, ?exprB, ?exprC, "&&", ?meth).
   ExprImpliesOther(?exprA, ?exprB),
   ExprImpliesOther(?exprA, ?exprC).
```

Figure 32: $A \rightarrow B$ **and** $A \rightarrow C \vdash A \rightarrow B \wedge C$

```
ExprImpliesOther(?exprLeft, ?exprA) :-
   isBooleanExprLeftRightInMethod(?exprLeft, ?exprB, ?exprC, "||", ?meth).
   ExprImpliesOther(?exprB, ?exprA),
   ExprImpliesOther(?exprC, ?exprA).
```

Figure 33: $B \rightarrow A$ **and** $C \rightarrow A \vdash B \vee C \rightarrow A$

```
ExprImpliesOther(?exprBNeg, ?exprANeg) :-
   ExprIsNegationOfOther(?exprA, ?exprANeg),
   ExprIsNegationOfOther(?exprB, ?exprBNeg).
```

Figure 34: $A \rightarrow B$ **and** $\neg B$ **and** $\neg A \vdash \neg B \rightarrow \neg A$

```
ExprImpliesOther(?expr, ?exprOther) :-
   ExprIsAlwaysFalse(?expr),
   isBooleanExprLeftRightInMethod(?expr, _, _, _, ?meth),
   isBooleanExprLeftRightInMethod(?exprOther, _, _, _, ?meth).
```

Figure 35: $False \vdash False \rightarrow A$

```
ExprImpliesOther(?exprOther, ?expr) :-
   ExprIsAlwaysTrue(?expr),
   isBooleanExprLeftRightInMethod(?expr, _, _, _, ?meth),
   isBooleanExprLeftRightInMethod(?exprOther, _, _, _, ?meth).
```

Figure 36: $True \vdash A \rightarrow True$

```
ExprImpliesOther(?expr, ?exprOther) :-
   ExprImpliesOther(?expr, ?exprCompOther),
   ?exprCompOther = ["||", ?type, ?exprOther, ?exprFalse] ,
   ExprIsAlwaysFalse(?exprFalse).
```

Figure 37: $A \rightarrow (B \vee False) \vdash A \rightarrow B$

```
ExprImpliesOther(?expr, ?exprOther) :-
  ExprImpliesOther(?exprCompOther, ?exprOther),
  ?exprCompOther = ["&&", ?type, ?expr, ?exprTrue] ,
  ExprIsAlwaysTrue(?exprTrue).
```

Figure 38: $A \wedge True \rightarrow B \vdash A \rightarrow B$

4. EVALUATIONS

foolala

5. RELATED WORK

foorelated

6. CONCLUSIONS AND FUTURE WORK

FooConclusions

ABBREVIATIONS - ACRONYMS

IR	Intermediate Representation
JVM	Java Virtual Machine

REFERENCES

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[3] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices*, volume 37, pages 57–68. ACM, 2002.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[6] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, volume 43, pages 270–280. ACM, 2008.

[7] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[8] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018.

[9] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[10] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO ’04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[12] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[13] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON ’99, pages 13–. IBM Press, 1999.