The Pennsylvania State University

The Graduate School

Computer Science and Engineering

BINARY-LEVEL TYPE INFERENCE USING DATALOG

A Thesis in

Computer Science

by

Ashley Huhman

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2018

The thesis of Ashley Huhman was reviewed and approved* by the following:

Gang Tan
Associate Professor
Thesis Advisor

Danfeng Zhang
Assistant Professor

Chitaranjan Das
Distinguished Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

The Doop framework is evidence that the Datalog programming language can be used to implement entire non-trivial program analyses while maintaining high precision and scalability. This research focuses on an assembly level graphed-based type inference analysis in Datalog. Datalog provides a straightforward and easy to follow representation of the algorithm used in the analysis originally performed by D. Zeng and G. Tan in the functional programming language Objective Caml. The constraint solving algorithm performs propagation of types on a graph structure for registers and stack slots at the assembly level. The purpose of this algorithm is to help refine a control flow graph that models an entire C program. A more precise control flow graph leads to better predications of dynamic decisions within an application, thus creating a more secure environment. This research aims to allow for a more understandable implementation of the type inference analysis, in particular the constraint solving algorithm. Once the Datalog program has finished computing the results for this algorithm, given the correct input, one can simply query the current type set for any execution point within the C program. The simplicity in creating the logic for the algorithm compared to the implementation in Objective Caml is shown.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank the ONR (Office of Naval Research) for the following grant support for Fall 2017: N00014-17-1-2539: Semantics-Directed Binary Reverse Engineering and Transformation Validation. Disclaimer: The findings and conclusions do not necessarily reflect the view of the funding agency.

Chapter 1

Datalog

Datalog is a "declarative logic language in which each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause" [11]. Datalog guarantees that all queries terminate though the usage of syntax and its implementation. A new and enhanced version of Datalog, PA Datalog, was developed by the company LogicBlox and used for this research [1]. The enhancements of PA Datalog and how other researchers have utilized it for program analysis will be discussed below.

About Datalog

The LogicBlox designers stressed the use of PA Datalog for more practical applications. Important new features included are constructors, stratified negation, use of modular development, entity types, and constraints [1]. These features allow the user to define in detail the kind of relationship needed between elements in the model. For example, a programmer can now state whether they desire a one-to-one relationship between A and B or a one-to-many relationship between A and B. Because of these upgrades, a complete and scalable points-to analysis has been developed for the Java programming language [2].

Datalog has previously been used for modeling low-level [13] [14] [15] and high-level [2] [16] [17] program analyses. Datalog itself is implemented by the high-level programming language Java. With Datalog, the user can more easily model the desired analysis without the trouble of working with low-level implementation details as would be the case for C for example [3]. Like the older version of Datalog, dependency cycles within relations are not allowed in PA

Datalog, thus eliminating the possibilities of non-terminating queries. Therefore, all queries can be guaranteed to execute in polynomial time [3]. All Datalog logic is based on *rules* and *facts*. *Facts* are inputs for the rules, and *rules* are used to recursively solve for a particular set of information. At the beginning of every rule, there is the *head*. The constraints that follow the head are called the *body*. The results for querying a predicate is given in the head and the body contains the constraints that must occur before the head predicate gets populated with information. Solving for the predicate gives way to the use of recursion, which is extremely vital for any program analysis involving computation of nodes with edges i.e. graph structures which are used to represent control flow graphs of programs.

As mentioned before, Datalog has been used in a large-scale analysis by Y. Smaragdakis and G. Balatsouras [2]. They have created a complete, precise, and usable points-to analysis framework for the Java programming language.

Doop Analysis

The points-to analysis for Java programs created by Y. Smaragdakis and G. Balatsouras is known as Doop. In points-to analysis the goal is to compute the following, “check if p can point to q in some execution of the program” [12]. This can be a more difficult computation depending on the indirect branches within a program. Indirect branches are caused by dynamic decisions during the execution of a program i.e. one cannot determine every branches target at compilation time.

Doop, compared to previous points-to analysis implementations, is both faster and more precise. It is the first complete end-to-end context-sensitive analysis in Datalog [3]. They have also implemented a modular framework that consists of context-insensitive analysis, object analysis, and call-site sensitive analysis for varying context depths as well. Doop was shown to be

faster than the previous analysis framework known as PADDLE. PADDLE is based on binary decision diagrams (BDDs) context-sensitive interprocedural analysis for the Java programming language [8]. Prior to Doop, PADDLE was regarded as the most complete and scalable points-to analysis [3].

One of the major accomplishments of Doop is the optimization techniques implemented. Y. Smaragdakis and G. Balatsouras mention two main techniques for their optimizations: queries-based optimizations through the use of explicit representation and small algorithm improvements [3]. The extended LogicBlox Datalog engine, PA Datalog, allows for the use of explicit relationships by providing constructors and functions for entity types. This was previously not attainable by the older version of Datalog. Figure 1-1 is an excerpt from the paper by Y. Smaragdakis and G. Balatsouras showing their use of an explicit Java-based variable points to relation for Anderson-style points-to analysis [18].

VarPointsTo(*var*, *heap*) ←
Reachable(*meth*), Alloc(*var*, *heap*, *meth*).

VarPointsTo(*to*, *heap*) ←
Move(*to*, *from*), VarPointsTo(*from*, *heap*).

FldPointsTo(*baseH*, *fld*, *heap*) ←
Store(*base*, *fld*, *from*), VarPointsTo(*from*, *heap*),
VarPointsTo(*base*, *baseH*).

VarPointsTo(*to*, *heap*) ←
Load(*to*, *base*, *fld*), VarPointsTo(*base*, *baseH*),
FldPointsTo(*baseH*, *fld*, *heap*).

Figure 1-1. Excerpt for Doop Anderson-style Analysis.

The *VarPointsTo* relations shown in Figure 1-1 represent the different ways a points-to set can be calculated for the Java language based on Anderson-style analysis [3]. For the first derivation

rule, it reads “if a method is reachable and a variable within that method has been allocated to a heap object, then that variable points to the heap”.

Importance of the Doop Analysis

The Doop framework has been a reference for this research in regards to future optimizations, modularity of multiple analyses implementations, and usage of explicit representation. The optimization techniques given in the papers [4] [5] [6] will be studied for future implementations of this research.

Chapter 2

Type Inference Analysis

This research is based on a recent publication from D. Zeng and G. Tan titled "*From Debugging-Information Based Binary-Level Type Inference to CFG Generation*" [9]. This research uses PA Datalog to implement a specific mechanism found in their framework which computes a set of possible types given by debugging information for registers and stack slots at different execution points in an assembly C program. A brief summary of their work will be provided below.

Purpose of Type Inference

The goal of the type inference analysis provided by D. Zeng and G. Tan is to provide an alternative method for creating a more precise CFG using "compiler-generated meta-information, including symbol tables, relocation information, and debugging information" [9]. Precise CFGs are vital for maintaining the security of an application. For an attacker, a lower-precision CFG can result in more program control without the knowledge of the application manager. Thus, having a high-precision CFG is necessary for a real-world application's security.

Previous CFG construction implementations are either based on binary-level reconstruction or compiler modification. Binary-level analysis does not require source code but yields a low-precision CFG, while the compiler modification method yields a high-precision CFG but requires source code. This type inference method has the ability to construct a high-precision CFG, which scales across versions of compilers and different compilers too [9]. Type inference analysis is used to refine the indirect target branches in the program, thus refining the CFG even

more. Their type inference analysis is implemented through a type inference engine that has the following four mechanisms:

1. Collect types from debugging information.
2. Stack-layout inference algorithm – normalize the representation of stack-slots with respect to a canonical frame address.
3. Constraint generation component – convert instructions and control flows into type constraints.
4. Constraint solving algorithm** – take the generated constraints as input and compute a set of potential types for each execution point in the program.

This research is focused on implementing mechanism 4 in PA Datalog. The goal is to increase readability and ease of finding type set results after Datalog execution has completed.

Framework for Type Inference

All the work done by D. Zeng and G. Tan is implemented in Objective Caml (OCaml) language, which is a functional programming language with some object-oriented language features added. OCaml is used for the "expression of symbolic and numeric algorithms" [10]. It also utilizes a parameterized module system.

This research uses the PA Datalog to implement the constraint-solving algorithm for C programs that was originally implemented in OCaml by D. Zeng and G. Tan. Prior to loading in the Datalog program facts, a C program must be compiled into a 32-bit binary. Once the executable is ready, the file is then sent through OCaml mechanisms 1 through 3. The output for mechanism 3 is the constraint generation information and a dataflow and instruction-based graph. This graph information is then converted to Datalog facts that meet the specifications stated in the Datalog logic file. Datalog will then use the fact's dataflow and instruction information as part of the constraint solving which performs type propagation. The final Datalog output for computing type set predicates will be compared to the OCaml results.

Constraint Generation Algorithm

Now will begin the discussion on constraint generation (mechanism 3). The constraint-generation output serves as input to the algorithm that is being implemented in Datalog for this research. The constraint generation output contains all the 'facts' that are used as input to the derivation logic in Datalog. Constraint generation includes gathering any types from debugging information and attaching the information to the associated storage locations. Also, the instructions are observed to generate the type of control flow that occurred between subsequent instructions in the program. Constraint generation is implemented on all lines of an x86 C program assembly code. The output for constraint generation is a graph-based structure whose nodes represent execution points in the program and are identified by the storage location name and the address of the assembly language program. If a node was assigned types by debugging information, they become decorated type variables.

Each node will have potential edges connecting to them to other nodes. These edges are labeled with two important pieces of information. The first edge label states the type of control-flow constraint that was observed from the instruction. There are three types of control-flow constraints given in the constraint-generation output:

1. **Value-flow constraint** – label 1

   An example of an instruction causing this is $x \coloneqq y$. This states that node x will now have a type set that includes all of node y's types.

2. **Dereference-flow constraints** – label 2

   An instance causing this is $x \coloneqq memory(y + offset)$. This states the type at memory location $(y + offset)$ will be read and assigned to x's type set.

3. **Address-flow constraints** – label 3

Example for this type of flow constraint is $x \coloneqq lea(y + offset)$. This will load the address for the type at storage location $(y + offset)$ and assigned this address type to x's type set.

The second edge label is the offset information which can be seen above. When the offset label is not 0, then a complex type such as a structure is present. The offset can then be used to gather the specific element's type within the structure at an offset such as $(y + offset)$. Therefore, when propagating types, the type sets for nodes can become more accurate than assigning the entire complex type. This is directly helpful for refining the CFG.

All of the information mentioned above is converted to facts and used as direct input for the Datalog rules. Hence, the derivations must handle all the different possibilities for debugging type information and control flow labeled edges.

Constraint Solving Algorithm

The constraint-solving algorithm this research implements in Datalog follows the algorithm described in [9] with the use of explicit representation, a feature mentioned in the Doop analysis. The algorithm takes the constraint generated information from the previously mentioned mechanism 3 and computes type sets for each execution point, or node, in the program being analyzed. The Datalog logic was tested on a subset of the same SPEC2006 benchmarks used in [9]. The logic behind the constraint-solving algorithm as well as all derivation rules in PA Datalog will now be discussed.

Before describing the logic, the facts must be introduced. As was mentioned in the Datalog information section, Datalog consists of facts and rules where facts serve as input to the derivation rules. For this work, there are three types of facts that can be loaded directly from the OCaml constraint generation information. Once the OCaml constraint generation information is

loaded in, a graph is generated in Datalog using these facts. The direct facts can be seen in Figure 2-2 below. Here, the *nodeDBGtype* predicates represent an instance where debugging information was given for a node or execution point in the program. There are two entity types, $Node$ and $Type$ can be seen in Figure 2-1. $Node$ and $Type$ are created as entities because each node and type instance should be uniquely identified by their corresponding name given, $nodeName$ and $typeName$. This eliminates duplicate and incorrect information when solving for the final type sets of nodes since these names act as constructors for $Node$ and $Type$. Constructors establish a one-to-many relationship.

$$\boldsymbol{Node}(t), Node\colon nodeName(t\colon s) \rightarrow string(s).$$
$$\boldsymbol{Type}(t), Type\colon typeName(t\colon s) \rightarrow string(s).$$

Figure 2-1: Two entity types given for the Datalog program as reference-mode predicates.

In Figure 2-2, the different types of edge predicate information are given in the form of successors and predecessors from the OCaml output. Each edge will connect two nodes and will contain edge information relating to the control flow between the two connecting nodes.

$$\boldsymbol{nodeDBGtype}(n, t) \rightarrow Node(n), Type(t).$$
$$\boldsymbol{val\_flow\_edge}(pred, succ) \rightarrow Node(pred),\ Node(succ).$$
$$\boldsymbol{val\_flow\_Struct\_edge}(pred, off, succ) \rightarrow Node(pred),\ Node(succ),\ int[32]$$
$$\boldsymbol{deref\_flow\_edge}(pred, off, succ) \rightarrow Node(pred), Node(succ), int[32](off).$$
$$\boldsymbol{addr\_flow\_edge}(pred, off, succ) \rightarrow Node(pred), Node(succ), int[32](off).$$
$$\boldsymbol{Type}(t), \boldsymbol{Type}\colon \boldsymbol{typeName}(t\colon s) \rightarrow string(s).$$

Figure 2-2: OCaml output information converted into Datalog facts.

From the $nodeBDGtype$ and all the edge predicates, all other node's type set can be initialized to indicate that the node has no predecessor. The entity element $Type$ represents all

different debug information types. Figure 2-3 shows the initial type assignment derivations for the nodes void of debug information and when debugging information is present.

$$\boldsymbol{dbg\_assign}(n) \leftarrow nodeDBGtype(n,\_).$$
$$\boldsymbol{no\_pred}(n) \leftarrow !\,(val\_flow\_edge(\_,curr\_node); val\_flow\_Struct\_edge(\_,\_,curr\_node)$$
$$; deref\_flow\_edge(\_,\_,curr\_node); addr\_flow\_edge(\_,\_,curr\_node).$$
$$\boldsymbol{nodeHasType}(n,\text{"<}T\text{>:}Top - source\text{"}) \leftarrow no\_pred(n).$$
$$\boldsymbol{nodeHasType}(n,t) \leftarrow nodeDBGtype(n,t).$$

Figure 2-3: Type initialization rules for all nodes.

There are three cases for type initialization. Case 1: If a node has no predecessor and no debugging type information, then assign it type "$T: Top - source$". This type indicates that a node has no type information available. This syntax matched the OCaml type assignment for this case. Case 2: If a node has been assigned type information from the debugger, then initialize it with that type. Case 3: Else, a node's type set does not get populated. Any node that is assigned debugging information does not require any further solving since debugging information is assumed to be always true.

After all nodes have been initialized with either the debugging type information or the assigned types "$T: Top - source$, then the calculation for computing the type sets can begin. All the initialization information is stored directly in the predicate $nodeHasType$ which is the predicate that will contain all the propagated type information for each node.

The predicate that contains all type set information for each node is $nodeHasType(n,t) \rightarrow Node(n), Type(t)$ where n is an instance of $nodeName$ and t is an instance of $typeName$. Once the constraint solving algorithm concludes, a node can have multiple entries in its $nodeHasType$ predicate. Figure 2-4 shows a few derivation examples from

the Datalog program for calculating a node's $nodeHasType$ predicate in an x86 assembly C program.

Value-flow constraint: Edge label is 0.

1. $nodeHasType(n,t) \leftarrow val\_flow\_edge(pred,n),$
   $!\,dbg\_assign(n),$
   $hasNodeType(pred,t),$
   $t! = \text{"<}T\text{>:}Top - source\text{"}.$

Dereference-flow constraint: Edge label is 1.

2. $nodeHasType(n,t) \leftarrow deref\_flow\_edge(pred,0,n),$
   $nodeHasType(pred,t),$
   $!\,dbg\_assign(n),$
   $t! = \text{"<}T\text{>:}Top - source\text{"}.$

Address-flow constraint: Edge label is 2.

3. $nodeHasType(n,t) \leftarrow addr\_flow\_edge(pred,0,n),$
   $!\,dbg\_assign(n),$
   $nodeHasType(pred,type),$
   $t = string\text{:}\,add[\text{“PTR”},type],$
   $t! = \text{"<}T\text{>:}Top - source\text{"}.$
4. $nodeHasType(n,ptrfield) \leftarrow addr\_flow\_edge(pred,off,n),$
   $!\,dbg\_assign(n),$
   $nodeHasType(pred,complex_t),$
   $hasFieldType(complex\_t,off,field),$
   $ptrfield = string\text{:}\,add[\text{"PTR"},complex\_t].$

Figure 2-4: Snippets for $nodeHasType$ predicate.

Derivation 1 for $nodeHasType$ represents a value-flow constraint. Instructions that invoke this control flow assignment are simple move instructions from $storage\ location_1$ to $storage\ location_2$. There is a value-flow constraint automatically generated that is not seen in the OCaml implementation but does exist in the Datalog implementation. These edge labels represent condensed information in the OCaml graph output. If there are subsequent instructions

where nothing interesting related to control flow or the assembly instructions is happening, the OCaml does not gives these subsequent nodes a $nodeHasType$. However, one of the goals in Datalog is to have the ability to query all execution point's type sets and thus is needed to properly perform the type inference constraint solving. Figure 2-5 shows what this information is loaded in as.

$$extra\_exePoints(n) \rightarrow Node(n).$$
$$val\_flow\_edge(pred, n).$$

Figure 2-5: Extra edges from Datalog implementation.

The deference-flow constraint with label 1 in Figure 2-4 can also include logic for when offsets are not equal to 0. If offsets are not equal to 0, then the type assignment is referencing a more complex $storage\ location$ such as a structure or an element in a fixed sized array. Assuming Datalog has the correct fact information for these types, like the fields within a structure, their offsets, and types, then Datalog will utilize the $hasFieldType$ predicate. This predicate can also be seen in the $nodeHasType$ derivation for address-flow constraint of label 2. The $hasFieldType$ predicate can be seen in Figure 2-5.

$$hasFieldType[complex\_t, offset] = complex\_t \rightarrow int[32](offset),$$
$$Type(complex\_t),$$
$$Type(field).$$

Figure 2-6: $hasFieldType$ predicate for creating more precise type propagation.

The predicate $hasFieldType$ is an example of a functional predicate in Datalog where the key-space inside the brackets uniquely identify what comes after the equal sign. So here, each

combination of $complex\_t, offset$ will uniquely identify to a single $Type$ or more specifically a $typeName$ instance. If the same key-space combination is used for identifying to any other $Type$ or $typeName$, the Datalog engine will produce an error.

To see all derivation rules and logic within the Datalog program, refer to Appendix A. One assumption for all derivations is that the necessary facts are provided automatically in some form via OCaml output.

Chapter 3

Results

The Datalog program was ran against three SPEC2006 benchmarks used in the type inference analysis [9]. Table 3-1 shows the number of nodes, debug types, edges, and more for LIBQUATUM, LBM, and SPECRAND. Comparisons between the Datalog output and OCaml output for constraint solving were done by creating a Datalog fact *propagated* that directly loads the final results for the OCaml constraint solving output. Figure 3-1 shows the Datalog derivation for calculating the difference set between *nodeHasType* and *propagated*.

$$difference\_set(n,t) \rightarrow Node(n), Type(t).$$
$$difference\_set(n,t) \leftarrow (!\left(nodeHasType(n,t), propagated(n,t)\right),$$
$$(nodeHasType(n,t); propagated(n,t))),$$
$$!\,extra\_exePoints(n), t! = \text{"<}T\text{>:}\,Top - source\text{"}.$$

Figure 3-1: Predicate used for calculating the difference between node*HasType* and *propagated*.

When calculating $difference\_set$, it is necessary to exclude extra nodes included in the Datalog implementation and not included in the OCaml implementation. As mentioned earlier, the Datalog program calculates the type set information for every execution point of the program, whereas OCaml uses the information internally but does not include the nodes as output for calculating the type set. If these nodes were not excluded here, the actual difference between the two method's outputs, $difference\_set$ and $propagated$, would not be seen. It would seem as if the two programs gave very different results when that is not the case.

Due to the discrepancies between information available to OCaml and the information available to Datalog i.e. the OCaml output for constraint generation, some Datalog and OCaml

outputs differ for nodes, when in reality both contain correct the information. The OCaml framework has access to the header C files and can observe them to narrow down type information even further during constraint solving. However, Datalog only has access to the OCaml constraint generation output given. This does not include any header information. Observe Figure 3-2 for an example in benchmark LIBQUANTUM for the "quantum_reg_struct" structure within the "quantum.h" file.

$$[39]EAX_{aa8}, [0]ELEM[0](int < 4 >)$$
$$[39]EAX_{aa8}, [62]ELEM[0]\big(typedef(quantum\ reg{:}\ STRUCT\ quantum\ reg\ struct < 20$$

Figure 3-2: Example of output difference for a node, when both types are correct.

The first $difference\_set$ entry in Figure 3-2 corresponds to a $propagated$ result for the LIBQUANTUM benchmark. The second $difference\_set$ entry corresponds to the $nodeHasType$ result. Although these outputs look nothing alike, they are indeed both correct. The $nodeHasType$ predicate gets this type from a debugging type assignment, while $propagated$ does as well. However, there is some logic implemented inside of OCaml not corresponding to the constraint solving algorithm. OCaml uses quality guessing to gather that the type propagated should be the first element within the structure "quantum_reg_struct" based on an OCaml syntax. To ensure these statements are correct, it can be seen inside the header file "quantum.h" for LIBQUANTUM that the first element inside the structure is an $(int < 4 >)$ field. These correct but different results will be propagated for all subsequent nodes causing different set information. This is the cause for many differences in the two method's results. So, although the two outputs are not identical, neither are incorrect. One is simply more precise. This can be changed if Datalog had complete fact information from the header files as well.

The following table, Table 3-1 contains all the results for the benchmarks LIBQUATUM, LBM, and SPECRAND.

Table 3-1: Information for benchmarks LIBQUANTUM, LBM, and SPECRAND.

	LIBQUANTUM	LBM	SPECRAND
Total *Node* entities	174794	21639	982
Total *val_flow_edge* facts	194139	22873	1017
Total *val_flow_Struct_edge* facts	69	14	4
Total *deref_flow_edge* facts	332	44	9
Total *addr_flow_edge* facts	43	14	1
Nodes assigned types by debugging information	3839	608	53
Execution time for Datalog (sec)	53.969	1.778	0.72
Execution time for OCaml (sec)	4.474	1.094	0.344
Average number of types per node after constraint solving	~1.07	~1.208	~1.063
Number of (*Node*,*Type*) tuples in *difference_set*	11485	1075	17

In the above table, the number of type sets seen in $difference\_set$ reflects the raw results, including the OCaml header information that Datalog does not have access to at this point. This information accounts for many of the $difference\_set$ entries. However, there is no way to easily include this header information without manually entering it into Datalog at this point in time, so it is included in the overall calculation of $difference\_set$. Beyond those differences due to intern header information retrieval within OCaml, the remaining differences are unexplained. Until all the information available for OCaml to use during constraint solving is available to Datalog as facts to the Datalog program, a correctness result of 100% cannot be

achieved. This is the limitation of using a high-level program such as Datalog for program analysis, all the information must be given in Datalog fact form.

Datalog file predicates were used to load in large amounts of data for the nodes with debugging type assignments, the extra nodes included in the type propagation for Datalog, all edge information including *val_flow_edge*, *val_flow_Struct_edge*, *deref_flow_edge*, and *addr_flow_edge*. These were all loaded in separately. The execution time listed in Table 3-1 does not include the loading time for the Datalog facts. Datalog execution is prolonged by loading in file facts separately since Datalog populates predicates as it loads in the data. Thus, excess execution may be occurring since there are repeats for edge information given to by OCaml constraint generation information. File predicates are necessary due to the Java runtime limitations of loading in large facts without the use of file predicates. Optimizations were not performed for this research, but Doop has shown with their algorithm enhancements a speedup of 1000 times can be achieved compared to un-optimized logic.

Once last measurement considered is lines of code for the OCaml and Datalog implementations. It's expected that the LOC for Datalog would be significantly less than the OCaml LOC. The following table shows the LOC for all files used in both implementations.

Table 3-2: Lines of code for Datalog and OCaml implementation.

	Total LOC
Datalog Implementation	39
OCaml Implementation (including external function calls)	~119

In the appendix, the Datalog program can be seen that was used to implement the type propagation. The OCaml LOC are an approximation because there were function calls to an

external file and it is possible not all function calls were noticed when observing the lines of code. Also, comments were not included in these LOC calculations.

Future Work

Since Datalog has been proven to perform at optimal levels for complex analysis, optimization methods used in the Doop framework can be utilized to not only extend the current work done in Datalog for this research but make execution time decrease drastically. This will be necessary for scalability when dealing with large benchmarks such as GCC in the SPEC2006 benchmark suit. Another improvement that can be made is to make information that is available to OCaml available to Datalog as Datalog facts. Without the same information set, it requires manual lookups which is both time consuming and unrealistic. Once these improvements are made, this Datalog type inference analysis can be extended to include even more for analyses implementations, including D. Zeng and G. Tan final control flow graph construction that uses the type inference information. This would be a natural conversion into Datalog due to the graph structure.

Appendix A

Datalog Program

The following is the entire Datalog program for this research. It includes all derivation and facts used. The spacing and indentation was changed to fit the format of this paper and appears differently in the actual program used in implementation.

```
/* Entity Declarations */
Node(n), Node:nodeName(n:s) → string(s).
lang:physical:capacity[`Node] = 1000000000000000000.

Type(t), Type:typeName(t:s) → string(s).
lang:physical:capacity[`Type] = 1000000000000000000.
hasFieldType[structure,off] = field_type → int[32](off), Type(structure), Type(field_type).

extra_exePoint(a) → Node(a).

/* Predicates used by nodeHasType */
addr_flow_edge(a,b,c) → Node(a), int[32](b), Node(c).
deref_flow_edge(a,b,c) →Node(a), int[32](b), Node(c).
val_flow_edge(a,c) →Node(a), Node(c).
val_flow_Struct_edge(a,b,c) →Node(a), int[32](b), Node(c).

nodeDBGtype(n,t) → Node(n), Type(t).

/* Type propagation constraint solving */
propagated(n,t) → Type(t), Node(n).

dbg_assign(n) → Node(n).
dbg_assign(n) ← nodeDBGtype(n,_).
no_pred(n) → Node(n).
no_pred(curr_node) ← !(val_flow_edge(_,curr_node) ;
                       val_flow_Struct_edge(_,_,curr_node) ;
                       deref_flow_edge(_,_,curr_node) ;
                       addr_flow_edge(_,_,curr_node)),
                       !dbg_assign(curr_node), Node(curr_node).

nodeHasType(n,t) → Node(n), Type(t).
nodeHasType(n,t) ← nodeDBGtype(n,t).
nodeHasType(n,"<T>:Top-source") ← no_pred(n).
```

```
nodeHasType(n,t) ← dbg_assign(n), nodeHasType(n,t),
                   (val_flow_Struct_edge(_,_,n) ; val_flow_edge(_,n) ;
                    addr_flow_edge(_,_,n) ; deref_flow_edge(_,_,n)).

// value-flow (0:0)
nodeHasType(n,t) ← !dbg_assign(n), val_flow_edge(pred,n), nodeHasType(pred,t),
                   t!="<T>:Top-source".
// value-flow (0:#)
nodeHasType(n,t) ← val_flow_Struct_edge(pred,_,n), !dbg_assign(n),
                    nodeHasType(pred,t), t!="<T>:Top-source".

// dereference-flow (1:0)
nodeHasType(n,t) ← deref_flow_edge(pred,0,n), nodeHasType(pred,t),
                   !dbg_assign(n), t!="<T>:Top-source".

// dereference-flow (1:#)
nodeHasType(n,field) ← deref_flow_edge(pred,off,n), !dbg_assign(n),
                       off!=0, nodeHasType(pred,structure),
                       hasFieldType(structure,off,field).

nodeHasType(n,t) ← deref_flow_edge(pred,off,n), off!=0,
                   !dbg_assign(n), propagated(pred,t).

// address-flow (2:0)
nodeHasType(n,t) ←  addr_flow_edge(pred,0,n), !dbg_assign(n),
                    nodeHasType(pred,type), t=string:add[t1,")"],
                    t1=string:add["PTR(",type].
// address-flow (2:#)
nodeHasType(n,ptrfield) ← addr_flow_edge(pred,off,n),
                          !dbg_assign(n), nodeHasType(pred,structure),
                          hasFieldType(structure,off,field),
                          ptrfield=string:add[ptrfield1,")"],
                          ptrfield1=string:add["PTR(",field].

difference_set(n,t) →Node(n), Type(t).
difference_set(n,t) ← (!(nodeHasType(n,t),propagated(n,t)),
                       (nodeHasType(n,t);propagated(n,t)) ),
                        !extra_exePoint(n), !dbg_assign(n), t!="<T>:Top-source".

// number of types per node
sum[n] = x ← agg<<x = count()>> nodeHasType(n,_).
```

References

[1] Green, Todd J., et al. “LogicBlox, Platform and Language: A Tutorial.” *Lecture Notes in Computer Science Datalog in Academia and Industry*, 2012, pp. 1–8., doi:10.1007/978-3-642-32925-8_1.

[2] Smaragdakis, Yannis, and George Balatsouras. “Pointer Analysis.” *Foundations and Trends R in Programming Languages*, vol. 2, no. 1, 2015, pp. 1–69., doi:10.1561/2500000014.

[3] Smaragdakis, Yannis, and Martin Bravenboer. “Using Datalog for Fast and Easy Program Analysis.” *Datalog Reloaded Lecture Notes in Computer Science*, 2011, pp. 245–251., doi:10.1007/978-3-642-24206-9_14.

[4] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In L. Dillon, editor, ISSTA ’09: Proceedings of the 2009 International Symposium on Software Testing and Analysis, New York, NY, USA, July 2009.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In OOPSLA ’09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 2009. ACM.

[6] Y. Smaragdakis, M. Bravenboer, and O. Lhot´ak. Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In POPL ’11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, 2011. ACM.

[7] Gottlob, Georg, and Christos Papadimitriou. “On the complexity of single-Rule datalog queries.” *Information and Computation*, vol. 183, no. 1, 2003, pp. 104–122., doi:10.1016/s0890-5401(03)00012-9.

[8] *Paddle: BDD-Based context-Sensitive interprocedural analysis of Java*, www.sable.mcgill.ca/paddle/.

[9] From Debugging-Information Based Binary-Level Type Inference to CFG Generation. D. Zeng and G. Tan. *In 8th ACM Conference on Data and Application Security and Privacy* (CODASPY), 2018.

[10] Chailloux, Emmanuel, et al. Developing Applications With Objective Caml. *O'REILLY & Associates*, 2000.

[11] "Datalog User Manual." *Datalog User Manual, The MITRE Corporation*, 2004, www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html.

[12] Rayside, Derek. "Points-To Analysis." 14 Nov. 2005.

[13] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In PODS '05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–12, New York, NY, USA, 2005. ACM.

[14] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, Applications of Logic Databases, pages 163–196. Kluwer Academic Publishers, 1994.

[15] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation, pages 131–144, New York, NY, USA, 2004. ACM.

[16] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In ICSE '08: Proc. of the 30th int. conf. on Software engineering, pages 391–400, New York, NY, USA, 2008. ACM.

[17] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. In Proc. European Conf. on Object-Oriented Programming (ECOOP), pages 2–27. Spinger, 2006.

[18] Lars O. Andersen. Program Analysis and Specialization of the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994.