An Investigation in Datalog and Static Program Analysis

Laura Thompson
University of Alabama at Birmingham
Birmingham, Alabama, USA
lauratho@uab.edu

Abstract

The declarative logic programming language Datalog is a unique and flexible language that lends itself well to deductive database systems and has particularly promising applications in the field of static program analysis. This purely logical language utilizes Horn clauses to represent relations and is unique in its ability to recursively compute transitive closure problems. The language has been implemented in several ways, but one implementation in particular, Souffle, stands out for its effectiveness in performing static program analyses. Static program analysis deals with the complex problem of modeling the semantics of real programming languages, and as a result, effectively writing program analyses is a difficult task. While the declarative syntax of Datalog lends itself well to this problem, implementations of Datalog are often slow and therefor ineffective. Souffle is a promising implementation because it translates the Datalog code into C++ and then compiles and runs that code making it fast and efficient. The goal of this paper is to serve as an introduction to the Datalog language, its implementations, particularly in Souffle, and its applications with regards to static program analysis. Although program analysis is a complex field with active research, this paper will introduce only simple control and data flow analyses of a small functional programming language, in hopes that this introduction will spark further interest in Datalog and program analysis.

CCS Concepts: • **Program Analysis, Declarative Programming, Data Flow Analysis, Control Flow Analysis**;

Keywords: Datalog, Souffle, Static Program Analysis, Introduction, Data Flow Analysis, Control Flow Analysis

Conference, Date, Location

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Laura Thompson. 2021. An Investigation in Datalog and Static Program Analysis. In *Proceedings of Conference Name (Conference).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Datalog is a declarative logic programming language that expands on the power of a traditional database system by allowing inductive queries on data. The language was originally created as a subset of Prolog in the 1970's although the term Datalog was not officially coined until 1986 by David Maier[4]. Datalog was originally intended to serve as a function-free version of Prolog and a 'middle-ground' between logic programming and database systems. Datalog is useful due to it having a comparatively simple syntax and semantics with respect to Prolog, while maintaining high expressivity [4]. Datalog is intended to be a purely logical language, but many implementations contain extensions that are not purely logical and will be explored later in this paper. Although Datalog has many applications and is used broadly to implement deductive database systems, this paper focuses especially on it applications to static program analysis.

This paper is intended to serve as an introduction to Datalog and its uses in static program analysis. Section 2 will discuss the key concepts of Datalog as a programming language, including Horn-clauses, relational databases, and recursion, as well as how these concepts are implemented using Datalog's syntax. Section 3 will focus on the implementation of Datalog, in particular how the language can be operationalized using relational algebra. This section will also explore some of the advantages and disadvantages that come with particular approaches to implementing Datalog. Section 4 will discuss how Datalog can be used for static program analysis and why static program analysis is such an important application of Datalog. This is of particular interest because Datalog implementations of program analyzers have been found to be significantly more efficient, flexible, and simple than other static program analysis methods[7]. Finally, section 5 will summarize the topics explored in this paper and discuss potential topics for further exploration.

2 Datalog Overview

This part of the paper is meant to focus on the theory of Datalog, and any specific syntax used will be written in Souffle. To begin, the central idea in Datalog is to use logic to deduce new facts using given input data and a set of rules. In this way, Datalog is a lot like other database query languages such as SQL. However, Datalog is more flexible than a language like SQL because its logic oriented syntax allows for recursion which can be used easily compute things SQL cannot, such as transitive closure. Consider the following example of ancestors both in SQL and Datalog.

Given the a table of parents and children, *Ancestors*, suppose we wanted to generate a new table containing all of the grandparents and grand children. We would do so using the (Inner) JOIN command in SQL, of the table with itself. This would look like:

Parent	Child
Mary	Emily
Emily	Grace
Grace	Clair
Michelle	Beth

Table 1. Example SQL table Ancestors with 'Parent' and 'Child' columns

SELECT T1.Child AS Grandchild,
T2.Parent AS Grandparent
FROM Ancestors T1
JOIN Ancestors T2
ON T1.Parent=T2.Child

which would result in the following table:

GrandParent	GrandChild
Mary	Grace
Emily	Clair

Table 2. Example SQL table Ancestors with 'GrandParent' and 'GrandChild' columns

The JOIN command can be thought of as the generalization of product and intersection. A JOIN clause combines rows from two or more tables (product), based on a related column between them (intersection). This allows us to determine relations one generation back based on the fact that the parent of one entry will be the child of another. In this same way, we could get information about two generations back, great grand parents, by performing another JOIN operation, this time between the table of grand parents and the table of parents.

SELECT T1.Child AS GreatGrandchild,
T2.GrandParent AS GreatGrandparent
FROM Ancestors T1
JOIN Ancestors T2
ON T1.Parent=T2.GrandChild

This will result in only one entry:

GreatGrandParent	GreatGrandChild
Mary	Clair

Table 3. Example SQL table Ancestors with 'GreatGrandParent' and 'GreatGrandChild' columns

Therefor, we have computed the full list of ancestors. In this particular example, all ancestors only required two commands in SQL, and presumably a third command would be needed to combine these tables into a single table of ancestors. However, in a situation involving many generations, it would become tedious to compute each individual generation and then merge the resulting tables. Instead, consider this same example in Datalog. Pure Datalog, that is Datalog implemented only theoretically and without any additional extensions, expresses logic exclusively as *Horn Clauses* that look like this:

GrandParent(x, z) :- Parent(x, y), Parent(y, z)

A Horn Clause is defined as a disjunction of literals with at most one positive literal. This can be written symbolically as $\neg p \vee \neg q \vee \neg v \vee ... \vee u$, or it can be thought of more simply as $p \wedge q \wedge ... \wedge v \rightarrow u$ meaning that the truth of all other facts *implies* the truth of the new fact [4] In the example above, a new fact, *GrandParent(x, y)* is true, and thus exists, if there are some facts *Parent(x, y)* and *Parent(y, z)* that exist for some values x, y, and z. This rule states that if y is the parent of x and z is the parent of y, then z must also be the grandparent of x. This is how Datalog deduces new facts.

One of Datalog's key features is its ability to compute rules recursively. Datalog is unique in that it can recur over a rule repeatedly until a fixed point is reached. This is particularly useful for problems involving computing transitive closure such as the ancestors example above. Given set of facts for

the relationship, *Parent(x, y)* we can generate a new set of facts for the relationship *Ancestor(x, z)*, where *Ancestor(x, z)* will be any previous relative, recursively using the rules:

Ancestor(x, y) :- Parent(x, y)
Ancestor(x, z) :- Ancestor(x, y), Parent(y, z)

In this example, the program finds all ancestors for a given set of *Parent(x, y)* relations by first computing the base case, that a parent is an ancestor, then by computing previous generations by finding the parents of ancestors. This is a problem of transitive closure where the program continues to compute the Ancestor relation on the given parent relation until it no longer produces any new Ancestors To demonstrate Datalog computes this, consider the input from the previous example:

Parent(Mary, Emily)
Parent(Emily, Grace)
Parent(Grace, Clair)
Parent(Michelle, Beth)

Upon its first iteration, the Datalog program will find: Ancestor(Mary, Emily), Ancestor(Emily, Grace), Ancestor(Grace, Clair), Ancestor(Michelle, Beth) which is equivalent to the initial Parent table. The program will then run again and find Ancestor(Mary, Grace) and Ancestor(Emily, Clair), which is the same as the GrandParent table. Then it will find: Ancestor(Mary, Clair), the only entry in the GreatGrandParent table. The program will run over the facts one final time, and no new ancestors will be found, so the program will terminate and return a database of all Ancestors. The database of facts initially given to the program for all values of Parent(x, y) is called the *Extensional* database or *EDB* because it contains facts that are explicitly defined prior to the program running. The facts generated by the Ancestor(x, y) rules are known as the *Intensional* database or *IDB* because the set of facts is defined by rules rather than a predetermined facts in the relation. A summary of key Datalog terms and their syntax in Souffle is given below.

Relation: A named group of attributes
Syntax: .decl R(x:type, y:type, ..., n:type)
Fact: A clause that holds true, also an instance of a relation
Syntax: .input RelationName
RelationName(xval, yval).
This searches the directory for a tab-separated file called RelationName.facts which it will use to generate a set of facts based on the relation RelationName(x, y) where x and y are tab separated entries in RelationName.facts of data type 'type'.
Rule: Expression representing a Horn Clause, used to find new facts.

Syntax: NewRelation(x, y) :- R1(x, "value"), R2(_, y).
The body of a rule can contain one or more facts, but must end in a period, and all facts must hold true for a new NewRelation fact to be generated. In Souffle, when a rule is used to find new facts, the values in the body of the rule can take several common forms. The first, is a variable such as "x" or "z" in this case which means that it can be any value already defined by some other fact of that relation. The second is a value, which is explicitly specified such as the string "Value" or the number 1234. These can be any value provided they are the same type as specified by the relation. The third type is _ which is a placeholder for any value, although it may be safer to use a variable to ensure the value is actually in the fact and relation. More information on the exact syntax of writing rules in Souffle can be found in the language documentation.

Although there is significantly more to the Datalog language than the examples covered in this section of the paper, it should provide a sufficient understanding of the basic functionality of Datalog, particularly as it pertains to the logic of writing facts, relations, and rules. Additionally, this section put a special emphasis on recursion because it is an integral and unique part of the language, and will continue to be used throughout this paper.

3 Implementation of Datalog

The previous section of this paper focused primarily on general concepts and syntax. This section will expand on that by covering some of the caveats that come with actually implementing Datalog. The first subsection will deal with how a rule in Datalog may actually be evaluated using relational algebra operations and also develop an experimental demonstration of this evaluation. This will serve as a transition into the second subsection which will discuss the stratification of Datalog rules in the Souffle implementation, which is a means of improving the efficiency of rule evaluation. The third subsection will cover an implementation of Datalog that is different than Souffle but also highly efficient. Finally, the paper will discuss some common extensions of the pure Datalog that are often implemented such as negation [2].

3.1 Datalog as Relational Algebra

The generation of new facts can most easily be thought of in terms of relational algebra. Before exploring how relational algebra operations can be used to represent logical expressions, this paper will briefly define four relational algebra operations that are fundamental to Datalog.

1. *Join* ($\bowtie$): The natural join takes two relations $R_1 \bowtie R_2$ and returns a third relation which includes tuples that are merged from R_1 and R_2 where their corresponding attributes have equal values.

2. *Union (∪)*: The union of two relations is a new relation that contains the union of the sets of tuples in each relation.

3. *Project(Π)*: $\Pi_(a1, a2, ..., ak)$(R) is the projection of relation R and indicates a new relation where elements a1 through ak have been removed.

4. *Rename(ρ)*: $\rho_(a \rightarrow a')$(R) is the rename operation on attribute a of relation R to rename the attribute as a'.

Given these four operations, the simple example

.decl R_1*(r1val1, r1val2)*
.decl R_2*(r2val1, r2val2)*
.decl R_3*(r3val1, r3val2)*
R_1*(a,c) :-* R_2*(a,b),* R_3*(b,c).*

can be computed as follows:

$t_1 = \rho_{r2val2 \rightarrow r3val1}$
$t_2 = t_1 \bowtie R_3$
$t_3 = \Pi_{r3val1}(t_2)$
$t_4 = \rho_{r2val1 \rightarrow r1val1}(t_3)$
$t_5 = \rho_{r3val2 \rightarrow r1val2}(t_4)$
$R_1 = t_5$

This process first renames the feature at location b in R_2 to be the same as b in R_3. Next t_1, which is R_2 with its second attribute renamed, is joined with R_3 along b, creating a new tuple containing *(r2val1, r3val1, r3val2). r3val1* is projected out from the tuple. Finally, the attributes of the remaining tuple are renamed to match those of R_1.

These same operations can be applied to a similar example involving recursion. In this case, we will use the problem of computing transitive closure. Given an extensional database of edges consider these two sets of rules for computing transitive closure:

path(a, b) :- edge(a, b)
path(a, c) :- path(a, b), path(b, c)

path(a, b) :- edge(a, b)
path(a, c) :- edge(a, b), path(b, c)

Intuitively, it would seem like the first set of rules is more effective than the second set of rules because it can compute all possible edges in fewer steps than the second set of rules. The second set of rules will have to iterate n-1 times where n is the number of nodes, so the graph can compute all paths of edge length 2, then all paths of length 3, and so on until a single path of length n is generated. The first set of rules takes log(n) iterations because it can compute all paths of length 2, then all paths of length 4, 3, and 2, then all paths of length 8, 7, 6, 5... and so on over each single step. Although this is seemingly more efficient, we can show using relational algebra how this takes up more computational space than calculating one path length at a time as in the second set of rules.

Souffle uses an evaluation strategy known as semi-naive evaluation to more efficiently compute recursion. Consider the fact *path* that is being evaluated. In naive evaluation, the entirety of *path* would have to be recomputed on each iteration, making it a slow process. Semi-naive evaluation divides up *path* into three sub-rules:

path_new - Facts generated in the current step
path_delta - Facts that were generated one step previous
path_full - All facts that have been generated thus far

Using these three rules we can think of the recursive rule

path(a, c) :- edge(a, b), path(b, c)

as follows:

1. Join *edge(a,b)* and *path_delta(b,c)* to create *path_new(a,c)*
2. Add *path_new* to *path_full*
3. Make *path_new* into *path_delta* and *clear path_new*

In this way, rather than entirely recomputing the set of paths, all paths of length 1 can be used to compute all paths of length 2, and then all paths of length 2 can be used to compute all paths of length 3, and so on, but paths of smaller lengths will not be recomputed. However, when computing the recursive rule

path(a, c) :- path(a, b), path(b, c)

rather than computing *path_new(a,c) :- edge(a,b)* $\bowtie$ *path_delta(b,c)*
the program must compute *path_new(a,c) :-*
path_full $\bowtie$ *path_full* $\cup$
path_full $\bowtie$ *path_delta* $\cup$
path_delta $\bowtie$ *path_full* $\cup$
path_delta $\bowtie$ *path_delta*
which results in significantly more redundant computation. In an experiment with a graph as small as 100 nodes and 99 edges with each edge connecting one node to the next, this recursive rule ran about 100 times slower than the recursive rule that computed one edge new edge length at a time. Therefor, it is important to keep the operations used to compute Datalog logic in mind when writing Datalog code.

3.2 Stratification of Graphs in Souffle

Another feature that Souffle uses to increase the efficiency of its recursive computations is the stratification of graphs. [6] A strata is a subgraph within the connectivity graph for a Datalog Program that contains no cyclic dependencies on

any other parts of the graph. This means that rules within a strata do not depend on any other rules outside of that strata. This allows the program to increase its efficiency when evaluating a program by avoiding redundant computations that may arise without prior understanding of the structure of the program. Strata can also be used to compute stratified negation, a form of negation sometimes used in Datalog that states the head of a rule is true when some part of the body is not true. Strata allow the program to ensure the body of a rule is fully computed before computing the head, to avoid having to remove facts that were found to be true and then later negated. [7]

3.3 Datalog Using Binary Decision Diagrams

Although this paper focuses on the Souffle implementation of Datalog, a better understanding of how Datalog works can be obtained by looking at other ways of implementing Datalog. One particularly efficient way to implement Datalog was proposed in the paper *Using Datalog with Binary Decision Diagrams for Program Analysis*[7]. This paper describes an an implementation of Datalog created by the authors known as **bddbddb** which stands for Binary Decision Diagram (BDD)-Based Deductive DataBase. This implementation of Datalog utilizes binary decision trees to represent large relations, which results in fast execution time because the operations take time proportional to the size of the data structure rather than the number of tuples themselves.

A binary decision tree is a type of data structure that can encode relations in terms of a series of true/false decisions rather than viewing an entire relation as once. The tree encodes whether or not a new fact can be generated from the given ones by encoding the entire relation as the binary decision tree. At each level of the tree, starting with the root node, if an input value is true the program will follow one branch, if false, the program will follow another. Remember that a Horn Clause can be thought of logically as $p \wedge q \wedge ... \wedge v \rightarrow u$ so to generate a new fact, the program can follow the true-branch repeatedly, otherwise, a false or non-existent fact will result in the program following the false branch and a new fact will not be generated. An example of how binary decision trees work, given in Whaley et al. is provided below:

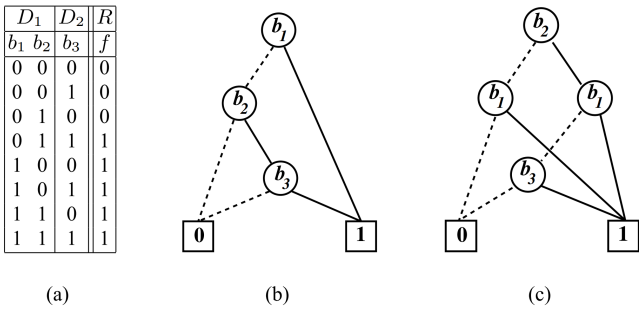

Figure 1. Example of two possible binary decision diagrams for the same data. [6, Fig. 2]

One clear caveat of the BDD's that is illustrated in this image is the fact that different trees may be generated from the same set of facts depending on how variables are ordered. **bddbddb** overcomes this problem by generating trees at random and then using learning techniques to remember more effective tree-ordering. Although this is effective, it is not much different than randomly guessing the order of the tree and lacks a proper methodology. Although it is important to recognize the setbacks that a given implementation of Datalog may have, the paper successfully demonstrated the effectiveness of **bddbddb** by creating several fast points-to program analysis and compared the execution times to those of other logic programming implementations as well as those same points-to analysis written in C. They found **bddbddb** to not only be faster, but required fewer lines of code to produce the points-to analysis when compared with other methods. The next section of this paper will cover the actual program analysis in Datalog more thoroughly.

3.4 Extensions of Datalog

Pure Datalog consists only of the Horn Clause-based rules described at the beginning of this paper.[2] However, it is common for implementations of Datalog to include an extended set of operations that make the implementation easier to use. This section will cover three common extensions: Negation, Comparison Operators, and Disjunction.

3.4.1 Negation. Negation is a non-trivial extension of Datalog, that can mean two different things: general negation, and negation of a single fact. General negation is the idea that *¬Head(a,b) :- Rule(a,b).* That is, the head of the rule is false when the body is true. This is generally not computationally compatible with Datalog and is usually not included as an extension. However, negation of a single rule is a commonly included extension. This is presented in the form *Head(a,c) :- Rule1(a,b), ¬Rule2(b,c).* Negation of a fact is difficult to compute because it requires prior knowledge of all facts to determine which facts are not included. One way of overcoming this is stratified negation, as implemented in [7]. This method determines negation rules once the entire strata has been computed in order to avoid having to remove facts when new ones are discovered. Another possible issue with negation is circular definitions such as *A(x) :- ! B(x).*
B(x) :- ! A(x).
which cannot be overcome even by stratification because it cannot be determined if anything belongs to the relation "A" without determining if it belongs to relation "B". But to determine if it is a "B" one needs to determine if the item

belongs to "A". Hence, the strata cannot be determined before negation. Negation is a useful tool, but it should be used with care due to complications it may cause.

3.4.2 Comparison Operators. The comparison operators $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$ are frequently used between variables and constants appearing in the body of a rule, making these a useful tool for performing queries in Datalog. In Souffle, the syntax for using these operations is applied as if it were another fact in the body of the rule:
Head(a, c) :- Rule1(a, b), Rule2(b, c), a < c.

3.4.3 Disjunction. Because Datalog rules are written as $p \wedge q \wedge ... \wedge v \rightarrow u$, disjuntion allows for the inclusion of inclusion of the logical 'or' operator within a rule in the body. Thus, the program could be extended logically to include clauses of the form $p \wedge (q \vee r) \wedge ... \wedge v \rightarrow u$. In Souffle the example syntax for this extension is:
LivesAt(person, building) :-
Owner(owner, building),
(person=owner ; Housemate(owner, person)).
which expresses the rule that a person lives in a building if they are the owner, or a housemate of the owner. Thus the conditions person=owner and Housemate(owner, person) are joined by ; to indicate that either must hold.

4 Datalog for Static Program Analysis

Program analysis is a difficult problem due to the complexity of real programming languages. Static program analysis in particular seeks to analyze programs by modeling their semantics and uncovering information about the program prior to run-time. Analyses of this type written in traditional imperative languages often require thousands of lines of code to be precise, while those written declaratively in various implementations of Datalog are often slow for large amounts of data and difficult to scale or customize [1, 3, 6]. The particular implementation of Datalog used in this paper, Souffle, seeks to remedy this problem by translating code written in Datalog to C++ and then compiling and running the resulting C++ code. As a result, Souffle allows for the speed and scalability of using an imperative language while retaining all of the advantages of writing program analysis declaratively in Datalog [3, 6]. It is for this reason that Souffle is used to explore Datalog in this paper.

There are several approaches to static program analysis, as well as different things that can be analyzed. More information on approaches to static analysis can be found in [5]. However, the remainder of this paper will focus specifically on introducing Data Flow and Control Flow analyses, as well as how to write a simple analysis. Data Flow analysis investigates how far and to where in a program data will reach. Control Flow analysis deals with how different parts of the program are reached from within the rest of the program. The two are similar in that they analyze the connectivity of a program, and therefor may be done together. The basics of Data and Control Flow analysis will be demonstrated using the following simple functional programming language:

$$
\begin{aligned}
e ::= \ & \texttt{(lambda (}x\texttt{) }e\texttt{)} \\
& \texttt{(app }e\ e\texttt{)} \\
& \texttt{(if }e\ e\ e\texttt{)} \\
& \texttt{(let ([}x\ e\texttt{]) }e\texttt{)} \\
& \texttt{(op }e\ e\texttt{)} \\
& \texttt{(const }v\texttt{)} \\
& \texttt{(ref }x\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
op ::= \ & + \\
& - \\
& * \\
& / \\
& < \\
& > \\
& = \\
& \textit{and} \\
& \textit{or}
\end{aligned}
$$

A small example program in this language might look like:

```
(app
   (let ([y (const 5)])
      (lambda (x) (+ (ref x) (ref y))))
   (const 6))
```

This is a simple application of the function (+ x y) where y is defined as 5 and applied on x which is 6 in this case. We can see intuitively that the program will reach the let statement and evaluate its body, then return the body lambda function to be applied on the value 6. However, a more precise understanding of when and how each part of the program is reached can be accomplished through a formal program analysis.

For the sake of this analysis, we will assume that the language has already been parsed into a set of nodes describing the cases above. The declaration of these nodes is shown as the set of relations below, where "id" is a unique value associated with that expression in the program, and any other ids in a relation are those of the sub-expressions within that expression.

SynLambda(id, arg, body_id)
SynApp(id, efunc_id, earg_id)
SynLet(id, var, rhs_id, body_id)
SynIf(id, guard_id, true_id, false_id)

SynPrimBin(id, op, arg1_id, arg2_id)

SynRef(id, val)
SynConstInt(id, val)
SynConstFloat(id, val)
SynConstBool(id, val)
SynConstString(id, val)

SynProg(start)

The first five relations represent first five supported expression forms in the syntax while the next five represent the breaking up of all "const" values in the program into the types int, float, bool, and string. SynProg is a relation that will be used to indicate to the analysis where the program starts, although for the case of our simple functional programming language, this will always be the outermost-leftmost expression. In the case of this analysis, all supported primitive binary operators, "op", will use the same node, but different symbols representing the operators themselves. The resulting nodes for our example program are:

SynConstInt(3, 5).
SynRef(6, "x").
SynRef(7, "y").
SynPrimBin(5,"+", 6, 7).
SynLambda(4, "x", 5).
SynLet(2, "y", 3, 4).
SynConstInt(8, 6).
SynApp(1, 2, 8).
SynProg(1).

To begin the analysis, we will need to declare several relations indicating what the analysis will find. These can be broken up into Data Flow analysis, Control Flow analysis, and relations that assist both analyses. To begin, we will declare some relations that will help both analysis as well as allow us to further understand the program. These relations will be

.decl ProducerExp(e:number)
.decl AtomicExp(e:number)

ProducerExp and *AtomicExp* are very similar. *ProducerExp* represents all forms that return some value to the program, while *AtomicExp* represents all forms that immediately return. These are subtly different, and this will become more clear as we continue with the program. The next nodes we need to declare will be for Data Flow analysis.

.decl ExpReachesVar(expid:number, var)
.decl ExpReachesRet(exp_id:number, ret_id:number)

ExpReachesVar indicates that some expression or data in the program has reached a variable assignment, which will then allow that data to be propagated as its associated variable. *ExpReachesRet* indicates that some data or expression reaches the return point of some other expression. The final nodes to declare are those for Control Flow analysis as follows:

.decl ReachableExp(exp_id:number)
.decl ReachableRet(exp_id:number)

ReachableExp will be used to determine which expressions are reachable in the program, and *ReachableRet* will tell which expressions can be returned from once reached. Now that these relations have been declared, we can begin to analyze the program. First, the start of the program must be reachable.

ReachableExp(id) :- SynProg(id).

Therefor, the initial App node, 1, will be reachable in our example program. Next, we need to indicate that all constants, pimitive binary operations, and lambdas are producers. This is because they all return some value that can be immediately used by the program. Thus for *SynPrimBin*, *SynLambda*, and *SynConst...* we will have a rule such that

ProducerExp(id) :- SynConstInt(id, _).

Additionally, as mentioned above, we have a separate relation for *AtomicExp* which expands on *ProducerExp* by including all expressions that immediately return, rather than only those that return some value. Therefor we can conclude that

AtomicExp(id) :- ProducerExp(id).
AtomicExp(id) :- SynRef(id, _).

Because all producer expressions will immediately return, and all references to variables will also immediately return, even if the value associated with that variable has not yet been evaluated. For our analysis the resulting producer expressions are 3, 4, 5, and 8 corresponding to the two constant integers in the program, the primitive binary operation, '+', and the lambda. The atomic expressions expand to include the variable reference nodes 6 and 7. There are two final rules that establish the groundwork for the analysis:

ExpReachesRet(id, id) :- ProducerExp(id), ReachableRet(id).
ExpReachesRet(p_id, ref_id) :- SynRef(ref_id, var),
ExpReachesVar(p_id, var).
ReachableRet(id) :- AtomicExp(id), ReachableExp(id).

The first rule applies to both Data flow and Control flow and

says that if an expression is both a producer expression and the expression can be returned from (control flow), then that expression reaches the return point of itself (data flow). The second rule specifies that if some variable is referenced in the program, and that variable is associated with a value, then the value reaches the return point of the expression in which the variable appears. This is key for understanding where variables are being referenced within a program. The third rule specifies the base case for control flow and states that if an expression is atomic, meaning it returns immediately, and it is reachable, then the return point of that expression is also reachable. At this point we have laid the ground work to begin the actual analysis of the program. As shown in the syntax, there are six cases where the program does something to an expression: *SynLet*, *SynIf*, *SynApp*, *SynPrimBin*, and *SynLambda*. In this case, *SynLambda* is a special case compared to the others because it is a producer expression and therefor returns immediately, but the subexpression in the body of the lambda will also need to be stepped through when analyzing the program. For the other five cases, we will break up their analysis based on the expression type, and then further by whether Control Flow or Data Flow analysis is occurring. We will begin with the Let case, although it is important to note that all cases will be similar. For data flow analysis we establish the following rules:

```
ExpReachesRet(p_id, let_id) :- SynLet(let_id, _, _, body_id),
                               ExpReachesRet(p_id, body_id).
ExpReachesVar(p_id, var) :- SynLet(_, var, rhs_id, _),
                            ExpReachesRet(p_id, rhs_id).
```

The first rule establishes that if an expression reaches the return point of the body of the Let then it will also return from the Let expression as a whole. The second rule establishes that the expression associated with the variable in the Let expression reaches that variable. For control flow analysis we have the rules:

```
ReachableExp(rhs_id) :- SynLet(let_id, _, rhs_id, _),
                        ReachableExp(let_id).
ReachableExp(body_id) :- SynLet(_, _, rhs_id, body_id),
                         ReachableRet(rhs_id).
ReachableRet(let_id) :- SynLet(let_id, _, _, body_id),
                        ReachableRet(body_id).
```

The first rule states that the right-hand side associated with the variable in the Let expression can be reached to evaluate if the Let expression itself is reachable in the program. The next rule says that the body is reachable if the right-hand side can be returned from, and then final rule says that the whole Let expression can be returned from if the body expression can be returned from. We can see from this structure that control flow analysis deals with this idea of "stepping" into one expression, checking that it returns, and if it does so, stepping into the next expression, and, if all sub-expression can be reached and returned in a certain evaluation order, the whole expression can be returned. This will be the case for a majority of the control flow analysis for this language. Therefor, the control flow rules for If are:

```
ReachableExp(guard_id) :- SynIf(if_id, guard_id, _, _),
                          ReachableExp(if_id).
ReachableExp(true_id) :- SynIf(_, guard_id, true_id, _),
                         ReachableRet(guard_id).
ReachableRet(if_id) :- SynIf(if_id, _, true_id, _),
                       ReachableRet(true_id).
```

The last second and third rule must also have a counterpart for the False branch. For data flow analysis, there are only two rules, which state that if an expression returns from either the true or false branch, then it must also return from the If expression. In this case, this is an oversimplification because we do not know based on the guard expression whether the true or false branch will be evaluated without further analysis, so although this analysis is still accurate, it is less precise. The rule for the True branch is shown below:

```
ExpReachesRet(p_id, if_id) :- SynIf(if_id, _, true_id, _),
                              ExpReachesRet(p_id, true_id).
```

The next two expression forms are App and PrimBin which are similar in that they take two expressions and apply some change to those expressions. The control flow for App gets a little bit tricky. The first two rules are nearly identical to rules we have already covered and state first that the function expression is reachable if the entire App expression is reachable, and that the argument expression is reachable if the function expression returns. We then introduce a new relation AppStep to indicate both that an application can be fully stepped through and that somewhere in the function expression there is a lambda function that is being applied on whatever is in the argument expression. This rule applies to both control and data flow because it checks that a lambda expression is being returned from the function expression in App.

```
AppStep(app_id, func_id, arg_id, lam_id, var, body_id) :-
                              SynApp(app_id, func_id, arg_id),
                              ReachableRet(arg_id),
                              ExpReachesRet(lam_id, func_id),
                              SynLambda(lam_id, var, body_id).
```

Initially, we had not evaluated what was inside a Lambda expression, only confirmed that it returned some value. Now that we are able to actually apply that Lambda on something, we check what is actually inside the body of the Lambda to confirm that it can return. This is expressed by the following rules which illustrate that we can now reach the body of

the Lambda and that if that body returns, then the entire application can return.

```
ReachableExp(body_id) :- AppStep(_, _, _, _, _, body_id).
ReachableRet(app_id) :- AppStep(app_id, _, _, _, _, body_id),
                        ReachableRet(body_id).
```

Additionally, we can use the new AppStep relation to analyze the data flow by ensuring that the argument variable in Lambda has the value of the argument expression in App if that argument expression has some return value.

```
ExpReachesVar(p_id, var) :- AppStep(_, _, arg_id, _, var, _),
                            ExpReachesRet(p_id, arg_id).
```

The rules for PrimBin will be similar to, but simplified from the analysis rules for App. In the case of this analysis, we are not ensuring that the return values for the two expressions are safe to use with a particular operator, such as numeric values for '+', although this would be a helpful additional analysis of the program. The first two rules for the control flow analysis are the same as others, stepping through each of the two expressions and ensuring that they are both reachable and return, and that if that is the case, the entire expression can also return. There is one additional rule for PrimBin which states,

```
ExpReachesRet(prim_id, prim_id) :-
                            SynPrimBin(prim_id, _, _, arg2_id),
                            ReachableRet(arg2_id).
```

This indicates that whatever data or expression results from the evaluation of the PrimBin can now be treated as a constant that returns from itself. Now that we have established a completed analysis, we can return to our example program

```
(app
   (let ([y (const 5)])
      (lambda (x) (+ (ref x) (ref y))))
   (const 6))

SynConstInt(3, 5).
SynRef(6, "x").
SynRef(7, "y").
SynPrimBin(5,"+", 6, 7).
SynLambda(4, "x", 5).
SynLet(2, "y", 3, 4).
SynConstInt(8, 6).
SynApp(1, 2, 8).
SynProg(1).
```

and show that our intuitions about how this program will be stepped through were correct. Running this program through our analysis yields the following results for control flow:

ReachableExp
1
2
3
4
5
6
7
8

ReachableRet
1
2
3
4
5
6
7
8

Figure 2. Results of ReachableExp and ReachableRet rules for example program control flow analysis

In this case, the control flow analysis is not particularly interesting, because every part of the program is reachable, and every part of the program returns. For the data flow analysis, the results are slightly more interesting:

ExpReachesRet	
4	2
3	3
4	4
5	5
8	6
3	7
8	8

ExpReachesVar	
3	y
8	x

Figure 3. Results of ExpReachesRet and ExpReachesVar rules for example program data flow analysis

It is clear from ExpReachesVar that the reference variables 'x' and 'y' are associated with the values 6 and 5 respectively. ExpReachesRet tells us not only are the four producer expressions, 3, 4, 5, and 8 returned from themselves, but that the Lambda, 4, is returned from Let, 2, and the two constant integer nodes, 8 and 3, are returned from their reference points, 6 and 7.

Although this example does not utilize every aspect of this written analysis, it is clear that even a simple analysis such as this one is helpful for understanding the flow of information throughout a program. Such analyses can are useful for things like security and program efficiency, allowing the

programmer to see and protect possible data insecurities or remove code that is either never reached by the program, or reached more times than necessary. Although program analysis is a difficult and complex problem, even simple solutions yield useful results which is why promising research in this field is essential.

5 Conclusion

Datalog is a unique and interesting language with promising applications to the field of static program analysis. It uses purely logical syntax in the form of Horn clauses to generate deductive databases, and is particularly unique in its ability to recursively compute closures in relations, such as the ancestors problem. There are many implementations of the Datalog, but the most notable in this paper is Souffle, who's syntax is used for examples throughout. Souffle's implementation of Datalog first translates the Datalog to C++ code and then compiles and runs the C++ which allows this implementation all of the speed of a traditional imperative language, with the simplicity and flexibility of a declarative language. This makes Souffle particularly apt for program analysis.

Program analysis was explored using a simple 'toy' functional programming language. Data flow and control flow, two common analysis types were performed for a sample program in this language, although the analysis could be applied to any program in the language, which is one of the advantages of declarative programming. The flow of the program can be modeled as a graph of interconnected nodes without having to fully model the semantics of the language. This analysis only computes basic information flow of a program, but it could be improved upon to include analysis to determine whether If statements evaluate to True or False as well as checking constraint to ensure mathematical operators only evaluate numerical values and boolean operators only evaluate boolean values.

This paper seeks only to serve as an introduction to the language of Datalog and its current implementations for the purpose of static program analysis. This is an active research field with far more to explore beyond the basic analysis example provided in this text. Ideally, this paper will spark further interest in the topics introduced and provide a starting point for further explorations of Datalog and static program analysis.

Acknowledgments

The completion of this paper and the exploration of its related topics would not have been possible without the help and mentorship of Dr. Thomas Gilray. I am thankful to have worked on this project with his assistance and guidance throughout and appreciate all of the extra time he has put forth to ensure its completion.

I am also thankful to the researchers in this field who's works have contributed to the study of static program analysis and enabled me to write this introduction to the topic using their work.

References

[1] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[2] S. Ceri, G. Gottlob, and L. Tanca. Extensions of pure datalog. 1990.

[3] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[4] David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. *Datalog: Concepts, History, and Outlook*, page 3–100. Association for Computing Machinery and Morgan & Claypool, 2018.

[5] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[6] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 196–206, New York, NY, USA, 2016. Association for Computing Machinery.

[7] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005.