Ben Greenman

Abstract

These are post-lecture notes for my February 7th presentation for HOPL 2017. The theme is the use of Datalog as a framework for building static analyses. The lecture covered the history of Datalog, who its original application to static analysis, and its later success.

These notes are a bare-minimum transcription of my on-paper notes for the lecture. Better than nothing.

Datalog for Static Analysis

Last year, there were two new papers about an old domain-specific language.

- *From Datalog to* FLIX [14] at PLDI 2016
- *Datafun: A Functional Datalog* [4] at ICFP 2016

Both have a similar motivation, essentially:

> Datalog has been successful because it lies at a confluence between *logic programming* and *static analysis.*

And both papers suggest new extensions for Datalog. The goal today is to understand why and how Datalog has been successful.

To start, we'll trace the papers' citations. Both cite work regarding the LogicBlox Datalog engine: FLIX cites the DOOP program analyzer (for Java programs) [6] and Datafun cites a paper that summarizes the engine [3]. They also both cite Whaley and Lam's bddbddb. Finally they each cite one early work on Datalog. FLIX cites Ullman's text on *Principles of database and knowledge-base systems* and Datafun cites the proceedings of a 1978 workshop on Logic and Databases, hosted by Gallaire, Minker, and Nicholas. (See Figure 1 for a picture of this paragraph. The solid lines represent citations in the 2016 papers.)

At this point, the earliest cited work that ues Datalog to implement a static analysis is the bddbddb paper, from 2004.[1] The bddbddb paper in turn cites work by Reps [17] that uses the CORAL deductive database [16] (a Datalog database) to implement demand-driven program slicing. But the idea of Datalog for static analysis goes (at least!) one step further: to Uwe Aβmann's dissertation work.

That's where we'll start today. The road map is (1) Aβmann, (2) classic Datalog, (3) bddbddb, and (4) DOOP. In terms of deltas and contributions, we'll (1) see a uniform framework for expressing three static analyses, (2) rephrase that framework as Datalog, (3) see how context-sensitive points-to analysis turned Datalog from a novelty to a necessity, and (4) the power of a mature Datalog implementation.

[1] Ullman's text demonstrates how Datalog can express and solve the reaching definitions problem, that is likely the first work to present the *idea* of implementing a static analysis with Datalog.

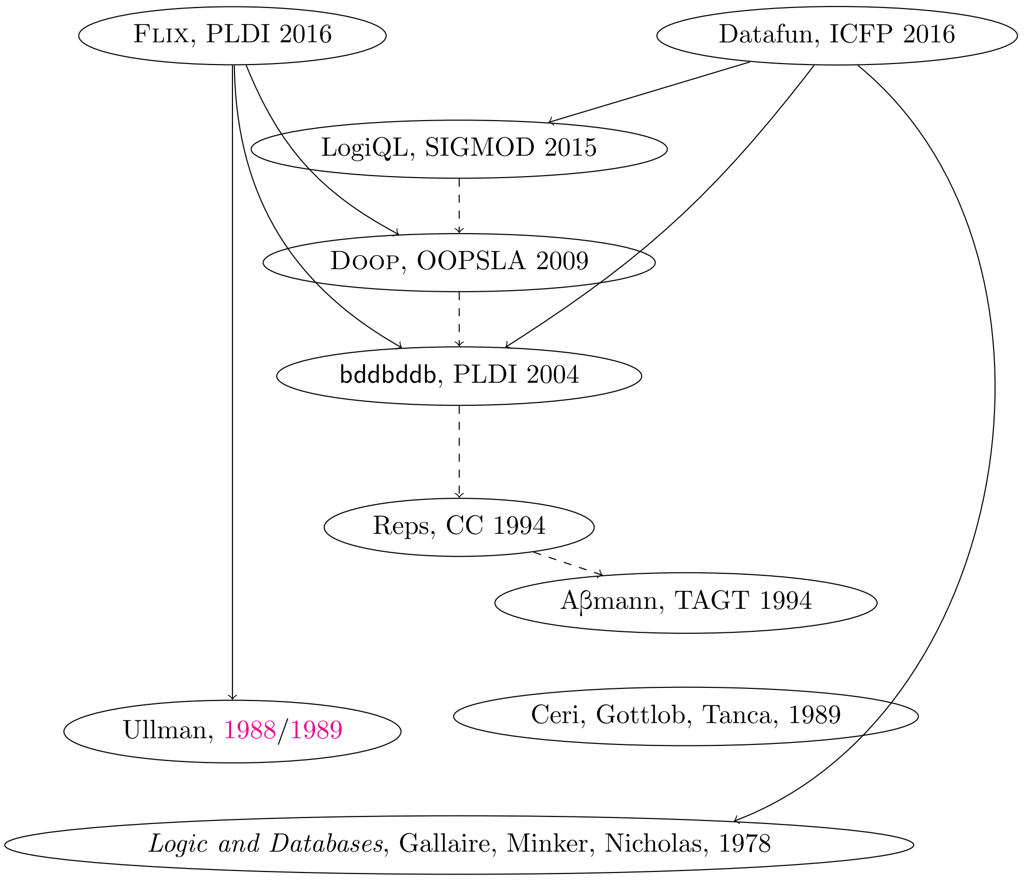

Figure 1: Tracing citations

EARS

Aβmann's framework for static analysis is EARS: Edge Addition Rewrite Systems. His 1994 paper demonstrates how EARS can efficiently solve three problems in static analysis.

3 Static Analysis Problems

Basic Block Ordering (BB) is the problem of building a control-flow graph for a set of basic blocks.

- A *basic block* is a straight-line sequence of machine instructions.
- Every basic block has a unique label.
- Machine instructions include definitions, jumps to other basic blocks, and conditional jumps to other basic blocks. (Jumps reference block labels.)

```
1       int y,z;
2       y = random();
3       if (y > 0.5) {
4           y = random();
5       }
6       z = 2 * y;
```

Figure 2: Example C code.

- The last instruction in any basic block is a jump. Intermediate instructions are not jumps.
- A control flow graph draws an edge from each basic block to its possible successors.

For example, the fragment of C code in Figure 2 might compile to three basic blocks. The first would represent lines 2 and 3. The second would represent line 4. The third would represent line 5. A control flow graph would connect the first block to the other two, and the second block to the third.

Reaching Definitions (RD) is the problem of associating variable references in a program with variable definitions. In Figure 2, the definition of `y` on line 2 reaches the use on line 3. The same definition also reaches line 6 because the first block (representing lines 2 and 3) is a direct predecessor of the third block (representing line 6).

To simplify this problem, Aβmann assumes two relations between basic blocks and definition instructions.

- The Def relation associates basic blocks to definitions that exit the block. If a block defines the same variable twice, only the latter definition is in the Def relation.
- The Preserved relation associates basic blocks to definitions; in particular, to definitions that the basic block does not overwrite.

These seem like big assumptions to me. Don't worry about it.

Equivalence Classes / Value Numbering (EQ) is the problem of finding *syntactically equal* subexpressions within a basic block. If the term `x + y` occurs twice in a basic block, these two occurrences belong in an equivalence class.

Uniform Representation

All three problems and their solutions can be expressed *uniformly* as rewrite rules on graphs. Figures 3, 4, and 5 present Aβmann's rules. Each rule has two parts: the left-hand side matches a sub-graph with labeled nodes and labeled

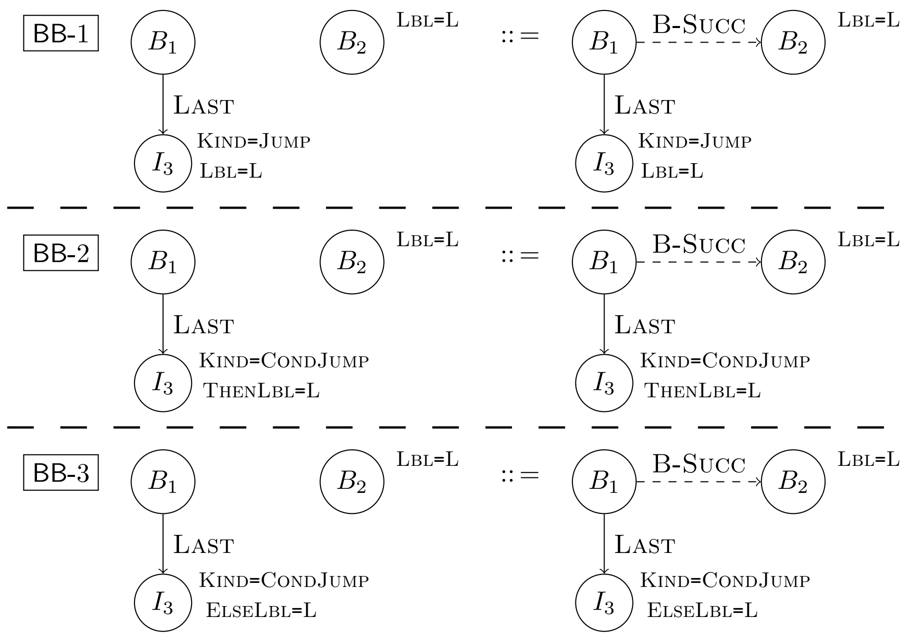

Figure 3: Graph rewrite system BB

edges (defined more formally in the next section). The right-hand side adds one (or more) edges to the left-hand side graph.

For BB, assume a LAST relation associating basic blocks to their final (jump) instruction. Then the rewrite rules to compute the "successor" relation between blocks are:

- (BB-1) if the last instruction in block B_1 is a direct jump, the jump's target is a successor of B_1.
- (BB-2, BB-3) if the last instruction in block B_1 is a conditional jump, the "then" and "else" targets are successors of B_1.

(Aβmann simulataneously defines a "predescessor" relation between blocks.)

For RD, assume the DEF and PRESERVED relations noted above.

- (RD-1) if a block defines a variable, the definition reaches the end of the block.
- (RD-2) if a definition flows into block B_1 and is not re-defined, the definition flows out of B_1.
- (RD-3) if a definition flows out of a block, it flows in to that block's successors.

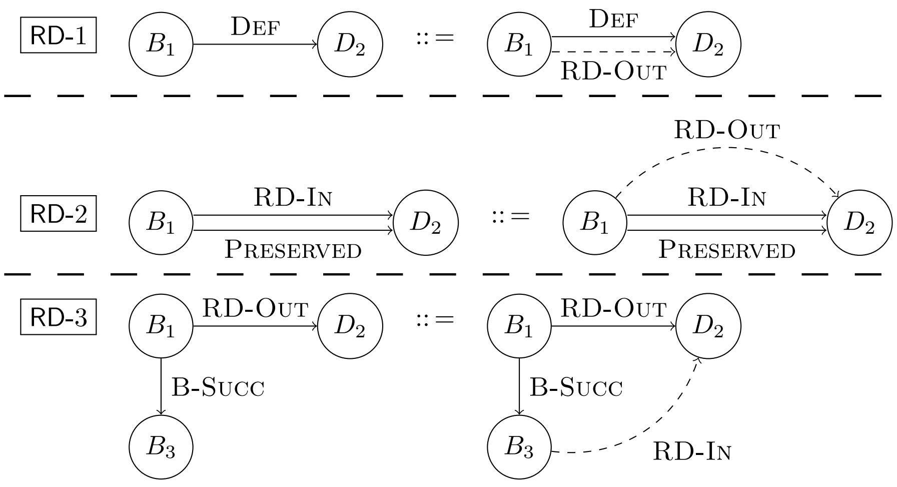

Figure 4: Graph rewrite system RD

For EQ, the goal is to compute "shallow" and "deep" equality relations (SIMPLE= and TREE=, respectively). These rules in Figure 5 work for numbers and addition; any expression node E is either a number or the sum of two expression nodes.

- (EQ-1) two number nodes with equal values are both SIMPLE= and TREE=.
- (EQ-2) any two sum nodes are SIMPLE=.
- (EQ-3) if two nodes are SIMPLE= and have TREE= children, they are TREE=.

To find the solution of these problems on a program P, first convert P to an appropriate graphs and then apply the rewrite rules to a fixed point.

Termination and Strong Confluence

The above graph rewrite systems are useful because they can *express* a variety of static analysis problems and furthermore yield a terminating, strongly confluent algorithm for computing the solution. Strong confluence means that applying the rewrite rules in *any* order gives the same solution.

Aβmann defines an *edge-addition rewrite system* (EARS) $\mathcal{E}$ as a pair (S, Z) where:

- Z is a graph with labeled nodes, labeled edges, and a dictionary of attributes for each node. A given pair of nodes can have multiple edges between them, but each such edge must have a *unique* label. Here is Aβmann's definition: $Z = \langle N, E, \Sigma_N, \Sigma_E, l_N, m_N, A_N \rangle$ where
  - N is a set of nodes

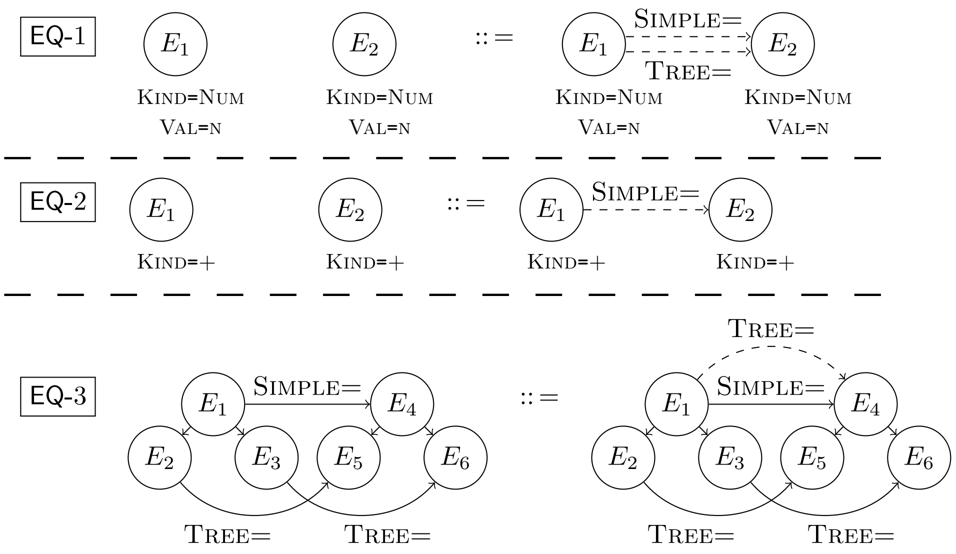

Figure 5: Graph rewrite system EQ

- E is a set of triples (N, N, Σ_E) (directed, labeled edges).
- Σ_N is a set of node labels (e.g. a set of strings).
- Σ_E is a set of edge labels.
- $l_N : N \mapsto \Sigma_N$ associates nodes to labels.
- $m_N : E \mapsto \Sigma_E$ associates edges to labels.[2][3][4]
- A_N is a set of functions that map nodes to a particular attribute. Aβmann writes $A_N = \{f \mid f : N \mapsto A_i, i \in \mathbb{N}\}$

• S is a set of pairs of such graphs, such that $(G_L, G_R) \in S$ implies that G_L and G_R have the *same nodes* and the edges of G_L are a subset of the edges of G_R.

Intuition: Z represents a program and S is the graph rewrite rules.

Next, *rule application* for an EARS is the non-deterministic process of selecting a rewrite rule in S and applying it to Z. Let $(S, Z) \to_R (S, Z')$ mean that applying rule $R \in S$ to graph Z yield graph Z'. Aβmann states three conditions for $R \in S$ to apply to the EARS (S, Z). Let $R = (G_L, G_R)$:

1. There exists a graph morphism $g_L : G_L \to Z$.

[2]Seems redundant to me.

[3]I should point out: around this time in the presentation, Mitch said "you all should look at a definition like this and, as scientists, pinpoint *why it is wrong*." I think his point was, if the formalism isn't as simple and clear as possible, you need to fix the formalism.

[4]Around the same time, Olin said "these compiler guys usually have a bunch of hacks and pseudo-math, and a very cool idea underneath"

2. The morphism g_L preserves node attributes.[5]

3. There is no morphism $g_R : G_R \to Z$ such that $g_L(G_L) \subseteq g_R(G_R)$.

Condition 3 ensures that rule application adds an edge. In other words, $(S, Z) \to_R (S, Z')$ implies that Z is a strict subgraph of Z'.

Theorem. *the transitive closure of rule application is terminating and strongly confluent*

Proof. Termination is obvious: rule application always adds an edge, and every graph contains at most one edge for every label and pair of nodes.

Confluence is also easy to prove, but relies on the reflexive-transitive closure of rule application, written $(S, Z) \to^* (S, Z')$. Suppose there are two ways an EARS can step:

$$\begin{tikzcd}
 & (S,Z) \arrow[dl, "R_1"'] \arrow[dr, "R_2"] & \\
(S,Z_1) & & (S,Z_2)
\end{tikzcd}$$

Then there exists an EARS (S, Z_3) such that:

$$\begin{tikzcd}
(S,Z_1) \arrow[dr, "*"] & & (S,Z_2) \arrow[dl, "*"'] \\
 & (S,Z_3) &
\end{tikzcd}$$

The proof is by cases on the edges added by R_1 and R_2. Let these edges be the sets E_1 and E_2.

- If $E_1 = E_2$ then $Z_1 = Z_2 = Z_3$. Do nothing.
- If $E_1 \subset E_2$ then $Z_2 = Z_3$. Apply $(S, Z_1) \to_{R_2} (S, Z_3)$.
- If $E_2 \subset E_1$ then $Z_1 = Z_3$. Apply $(S, Z_2) \to_{R_1} (S, Z_3)$.
- Otherwise, apply $(S, Z_1) \to_{R_2} (S, Z_3)_{R_1} \leftarrow (S, Z_2)$.

□

Efficent Solutions

These problems (BB, RD, EQ) are old problems in static analysis. When Aßmann began developing his framework, all three problems had well-known and efficient solutions. But the solutions for each problem were ad-hoc; it was not clear how those solutions could systematically lead to efficient solutions for other problems.

The main contribution of the EARS framework is that it provides a *systematic* method of deriving *efficient solutions* to problems. Aßmann shows that

[5] Aßmann formalizes this. I won't.

$$\begin{array}{lll}
S & ::= & R \ldots \\
R & ::= & F \mid Q \mid L \mathrel{:-} L \ldots \\
F, Q, L & ::= & p(a \ldots) \\
p & ::= & \langle \texttt{symbol} \rangle \\
a & ::= & \langle \texttt{symbol} \rangle \mid X
\end{array}$$

Figure 6: Datalog syntax

these solutions are as efficent as the previously-known, ad-hoc solutions. See the paper for details: the idea is to start with a straightforward, iterative solution and optimize the for-loops based on the number of nodes in a rewrite rule with no incoming edges.

To Datalog

In a short subsection, Aβmann notes that his graph rewrite rules can be mapped to Datalog. EARS are "just Datalog", and Datalog is "just EARS". The next section introduces Datalog and explains the mapping.

EARS External Links

- Sourceforge page for Optimix: http://optimix.sourceforge.net/ Optimix is part of Aβmann's dissertation, it is a language for specifying EARS-based transformations for C and Java code.
- DBLP page for the TAGT conference (where the EARS paper appeared): http://dblp.uni-trier.de/db/conf/gg/index.html

Datalog

One quick definition of Datalog is "Prolog without function symbols or negation". Another is "a deductive database"; that is, a database extended with first-order Horn clauses.

Figure 6 presents a syntax for Datalog. A program S is a sequence of facts F, rules, and queries Q. A rule $L_0 \mathrel{:-} L_1 \ldots L_n$ is an axiom for deducing new information (L_0) from existing facts (L_i, for $i \in [1, n]$). Facts, rules, and queries consist of predicate symbols p and atoms a. Atoms are either symbols (typeset in lowercase, serif font) or variables (captalized, in math mode, e.g. X).

There are two syntactic restrictions:

1. A fact may not contain variables.
2. Any free variables in the head of a rule must be appear in the rule's body. Let $\mathsf{fvs}(L)$ denote the set of variables that appear in L. Then the restriction on $L_0 \mathrel{:-} L_1 \ldots L_n$ is that $\mathsf{fvs}(L_0) \subseteq \bigcup_{i\in[1,n]} \mathsf{fvs}(L_i)$.

$$
\begin{aligned}
\mathsf{b\_succ}(B_0, B_1) \text{ :– } \quad & \mathsf{basic\_block}(B_0) \\
& \mathsf{basic\_block}(B_1) \\
& \mathsf{last}(B_0, I) \\
& \mathsf{kind}(I, \mathsf{jump}) \\
& \mathsf{lbl}(I, L) \\
& \mathsf{lbl}(B_1, L)
\end{aligned}
$$

Figure 7: Datalog encoding of BB-1

Figure 7 encodes the rule BB-1 in Datalog. This encoding assumes there are predicates basic_block, last, kind, and lbl corresponding to the edges and labels in the BB-1 graph. Using these predicates, the rule describes how to deduce that a block B_0 has a successor block B_1.

The syntax and example should give an intuition for the semantics of a Datalog program. For one last bit of intuition, here is the translation ($\sim$) of a Datalog rule to first-order logic (specifically, a translation from rules to Horn clauses):

$$L_0 \text{ :–} L_1 \dots L_n \sim \forall \vec{X} \,.\, (L_1 \wedge \dots \wedge L_n) \Rightarrow L_0$$

$$\text{where } \vec{X} = \bigcup_{j \in [0,n]} \mathsf{fvs}(L_j)$$

More precisely, the semantics of a Datalog program is the least fixed point of the *elementary production principle* (EPP). The EPP describes how a Datalog program S can step to a program S' using a rule in S to derive a new fact:

$$
\begin{aligned}
S \to_R (S \cup (L_0\ \theta)) \quad \iff \quad & \exists R \in S \\
& \wedge\ R = L_0 \text{ :–} L_1 \dots L_n \\
& \wedge\ \exists F_1 \dots F_n \in S \\
& \wedge\ \vec{X} = \textstyle\bigcup_{j \in [0,n]} \mathsf{fvs}(L_j) \\
& \wedge\ \exists \theta : \vec{X} \mapsto F \text{ such that } \theta(X) \in S \\
& \wedge\ \forall i \in [1, n] \,.\, L_i\ \theta = F_i
\end{aligned}
$$

where $L\ \theta$ is the fact produced by replacing free variables in L by their value in θ.

The least fixed point of applying the EPP is the set of facts that a Datalog program represents.

Databases

The previous section explains how a Datalog program S represents a database of facts. This database is known as the *intensional database* (or IDB) represented by S [7].

It's often useful to connect a Datalog program to an existing database of facts. Such databases are called *extensional databases.*

Once we incorporate an extensional database, we can view a Datalog program P as a mapping from an EDB to an IDB:

$$P : \text{EDB} \rightarrow \text{IDB}$$

Typically, P is a *time-invariant* mapping from a *time-invariant* set of facts (the EDB) to some answer (the IDB).

(The EDB / IDB distinction is a pragmatic one. You could pretend that the EDB is a sequence of facts at the top of the Datalog program and the IDB is always fully computed and committed to an extensional database.)

History Lesson

My reference for "pure Datalog" is the 1989 article by Ceri, Gottlob, and Tanca [7]. At that time, Datalog was well-known and well understood in the *logic and databases* community. Here's a very quick timeline of the world until 1989.

- (19XX) databases are as old as computers. The early implementations[6] were *link-based*. Unlike computers, there was no early mathematical model of databases.
- (1970) Codd proposed the *relational model* of databases, founded on *relational algebra* [9]. This was a hit. For example, SQL databases are relational databases. Codd received a Turing award in 1981.
- (196X-197X) meanwhile, researchers began exploring the idea that mathematical logic could be useful for understanding databases.
- (1973) Colmeraur and his students develop the first PROLOG interpreter.
- (1974) Kowalski suggests logic as a programming language.
- (1976) Van Emden and Kowalski give a semantics of predicate logic as a programming langauge [19]. Datalog *is* predicate logic; their model-theoretic, proof-theoretic, and fixpoint semantics were the foundation.
- (1978) Gallaire, Minker, and Nicholas organized a workshop titled *Logic and Databases*. At this workshop:
  - Someone introduced Datalog.
  - Reiter stated the *closed-world axiom* implicit in relational models.
  - Clark stated the *completion axiom*, also implicit in relational databases.
- (198X) researchers explored extensions to Datalog and techniques for efficiently answering queries (without computing the entire IDB!). Extensions include:

[6]Such as IBM's hierarchical model.

  - negation
  - arithmetic, primitive functions such as `<`
  - complex objects, missing information `null`

Evaluation techinques include:

  - semi-naïve evaluation
  - the magic sets transformation
  - the query-subquery method

Suffice to say, Datalog was considered a successful experiment by 1989. No commercial implementations of Datalog existed, but the theory was all laid out.

In fact, the magic sets transformation is the premise of the 1994 paper by Reps [17]. The transformation minimizes the number of facts computed to answer a query based on symbols that appear in the query. Reps uses this transformation to implement *demand-driven* program slicing.

One more thing that doesn't fit anywhere else: Datalog corresponds to *positive relational algebra* with recursion. Full relational algebra has a set-difference operator.

One more thing: I posit that there is a paper that proves you can encode any polynomial time algorithm in Datalog [10]. (Flix cites this paper as such, but I did not understand the paper's theorems.)

Further Reading

- The survey by Gallaire, Minker, and Nicholas [11].
- The 20-year retrospective by Minker [15].

Context-sensitive Points-To Analysis (for Java)

Aβmann and Reps demonstrated that Datalog could successfully be used to implement static analyses. Very good. But for the next decade, there doesn't seem to be much work using Datalog for static analysis.[7]

(As I understand, Aβmann didn't use Datalog because (1) his goal was to add EARS to an existing compiler, which used adjacency lists as an IR and (2) he did not have access to a Datalog implementation. Reps used the Coral deductive database, but reported order-of-magnitude slowdowns for using Coral rather than C.)

It took the combination of an old problem and a new programming paradigm to revive interest. The paradigm was object-oriented programming. The problem was context-sensitive points-to analysis.

Points-to analysis is essentially the reaching definitions (RD) problem. The goal, in the OOP sense, is to describe the heap objects that a variable can

[7]Mitch says: that's not entirely true, there are definitely some papers where—if you squint—you realize they're using Datalog.

denote. A context-sensitive points-to analysis further indexes this information. Exactly how to index is a design choice; for example, one way to index is to track the previous three (generalization: k) method names on the call stack.

Proposition: many static analyses require points-to information.

In the early 2000's, if you wanted points-to information for your Java program, you had essentially two alternatives:

- Apply a know context-sensitive algorithm, limit yourself to programs of about 3,000 lines.
- Apply a unification-based [18] or context-insensitive algorithm.

Scaling Points-to Analysis

John Whaley and (advisor) Monica Lam at Stanford came up with a radical solution [20]. They decided to apply a context-*insensitive* algorithm to an *exploded* control-flow graph. That is, they added paths to a given control-flow graph to make context information explicit in the paths. Consequently, the insensitive algorithm would give context-sensitive results.

The obvious problem with this approach is that the number of contexts in a large, object-oriented program is usually large.[8] Whaley and Lam report that $10^1$4 contexts is "typical", and one of the DaCapo benchmarks had $10^2$3 contexts.[9] Yet they made the algorithm work at scale by:

- representing program paths as relations
- representing relations as boolean functions
- using binary decision diagrams (BDDs) [1] to efficiently represent the functions.

BDDs worked because these contexts had a lot of redundant information.

Datalog for Points-to Analysis

The PLDI 2004 paper [20] reports on the implementation of `bddbddb`: a binary-decision-diagram based deductive database. The database front-end is Datalog. Whaley and Lam encoded a points-to of C and Java programs as Datalog rules. (Later, with Livshits, Martin, Avots, Carbin, and Unkel, they encoded a variety of taint analyses in Datalog [12].[10])

The database back-end is binary decision diagrams. That's right, Whaley and Lam built an efficient deductive database from the ground up just to implement points-to analysis.

[8] This is exactly why previous algorithms did not work at scale.

[9] Writing this, I realize I don't know what they mean by "context". Like, what's the sensitivity. TODO fix.

[10] And *furthermore* developed a compiler from source-code patterns to Datalog rules.

Conclusion: they succeeded at a problem that nobody before they could solve. Their points-to analysis finished in minutes on programs selected from the DaCapo benchmarks [5].[11]

A Commercial Datalog Engine

Finally, I want to mention the DOOP system for points-to analysis of Java programs [6]. DOOP builds on bddbddb and the later PADDLE system [13] in that DOOP is:

- encodes call-graph construction in Datalog (as well as the points-to analysis);
- extends the logic from the PADDLE system [13] to encode more details of the Java language;
- handles reflection "better" (details were not apparent to me);
- handles exceptions "better" by building the relevant parts of the call graph[12]

DOOP is historically significant because it uses a commercial Datalog engine developed by LogicBlox.

I am not aware of any work before DOOP to mark the existence of a commercial (i.e. implementing all the optimizations from the 1980's) Datalog engine. That means its approximately 33 years from the initial conference on logic and databases to a practical realization.

Oh, and by modifying their Datalog rules to trigger certain behavior in the Datalog engine, Bravenboer and Smaragdakis achieve numbers that suggest DOOP is the fastest and most scalable implementation of points-to analysis for Java programs. Regarding scalability, they are able to index each pointer with 2 abstract objects or associate 2 contexts to each heap location. The PADDLE system implemented these settings, but did not produce results in reasonable time.

Conclusions

Datalog is:

- predicate logic as a programming language
- a convenient interface to a database

[11]Remember that program with 10^23 contexts? Well bddbddb could not actually encode all those, it stopped at $2^64 - 1$ contexts and merged all others into the last context.

[12]I think this is possible because the DOOP builds a call graph on-demand instead of precomputing, similar to how Reps used Datalog for demand-driven program slicing. But this is just a guess, beyond the phrase "on-the-fly" the details were not apparent to me from reading the paper.

The high-level benefits of storing program facts in a database and encoding a static analysis in Datalog are:

- the encoding works for a variety of problems, and yields a uniform solution
- the Datalog encoding separates the *specification* of the analysis from *how to compute* the analysis
- techniques for efficient evaluation of Datalog are techniques for efficient evaluation of your analysis

Meta-lessons: watch out for

- confluence points among research areas
- sub-Turing languages that solve a real problem

FAQ

(Not really an FAQ. These are the questions I had answers to prior to the talk, and my answers to questions that came up at the end.)

Q. Does the `bddbddb` paper report the overhead of building the deductive database?
A. Not that I know of. I do know that the DOOP paper criticizes the overhead of BDDs [6], so lets say it's "significant", but not outrageous.

The AVERROES paper later criticized the overhead of representing a program analysis in DOOP [2]. So it goes.

Q. Given the overhead, do you still think Datalog is a good idea.
A. Yeah I think this is "the future". Data is cheap, and we should look to re-use quality research (e.g. evaluation techniques for Datalog) without reinventing them each time (building a `bddbddb`).

Q. Is there an easy way to try or use Datalog?
A. Hell yes. There's implementations everywhere. You can play with `#lang datalog` in Racket. For real work, there's the Datomic library for Clojure, Graal project for Java, and Dyna for Haskell. Also see the recent work on FLIX and Datafun.

Q. Why did you pick this topic?
A. Last Spring I was building on the Type Systems as Macros [8] approach to type systems to put static analysis in the Racket macro expander. Researching Datalog is a step in a broader goal of looking at other frameworks for static analysis to determine what will work best in the macro expander.

Q. Can you quickly explain this magic sets thing?
A. Sure, it's a source-code rewriting technique. Start with a query, for example $\mathsf{p}(\mathsf{a}, X)$. Then rewrite the $\mathsf{p}(A, B)$ rule to include the condition $\mathsf{relevant}(A)$. If

A is not relevant, you don't want to compute facts about it. This "relevant" predicate is a new rule that's introduced by the rewriting, along with the fact $\mathsf{relevant}(a)$ and some way of identifying other relevant atoms.

Q. Any opinions on FLIX and Datafun?
A. FLIX sounds exicting. It's not straightforward to express lattice-based program analysis in Datalog. It is straightforward to do so in FLIX.

What worries me though, is that they're building a Datalog engine from scratch. Hope they stick with it, and implement the optimizations.

Datafun is interesting, but I need to see more (or read the paper more carefully). For example, they cite other researchers extensions to Datalog as evidence that we could use a typed functional language for implementing Datalog extensions. But they don't show whether the language can express those existing extensions.

Q. Final remarks?
A. There's an interesting "wishlist" at the end of the Ceri, Gottlob, Tanca article [7]:

- To date, there are no useful applications of recursive or non-linear Datalog rules (points-to analysis certainly changed that)
- Datalog would be more useful if it was more of a programming language, e.g. with modules and structures, and the ability to implement a user interface. (Shouts out to Datafun.)
- Datalog is too declarative, sometimes you really do want to go low-level and control how things compute.
- Datalog focuses on interactions with a single database, but the modern trend is towards heterogenous systems.

They thought about a lot of interesting things in 1989.

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[2] Karim Ali and Ondřej Lhoták. AVERROES: Whole-program analysis without the whole program. In *ECOOP*, 2013.

[3] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, 2015.

[4] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: a functional Datalog. In *ICFP*, 2016.

[5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Anthony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[6] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243 – 262, 2009.

[7] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146 – 166, 1989.

[8] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *POPL*, 2017.

[9] E.F. Codd. A relational model of data for large shared data banks. 13(6), 1970.

[10] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *CSUR*, 2001.

[11] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys (CSUR)*, 16(2):153–185, 1984.

[12] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context sensitive program analysis as database queries. In *PODS*, pages 1 – 12, 2005.

[13] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *TOSEM*, 18(1), 2008.

[14] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to FLIX: A declarative language for fixed points on lattices. In *PLDI*, 2016.

[15] Jack Minker. Logic and databases: past, present, and future. 18(3):21–48, 1999.

[16] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL deductive database system. In *SIGMOD*, 1993.

[17] Thomas Reps. Demand interprocedural program analysis using logic databases. In *CC*, 1994.

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1995.

[19] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.

[20] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.