Constraint-Based Synthesis of Datalog Programs

Aws Albarghouthi[1(✉)], Paraschos Koutris[1], Mayur Naik[2], and Calvin Smith[1]

[1] University of Wisconsin–Madison, Madison, USA
aws@cs.wisc.edu
[2] Unviersity of Pennsylvania, Philadelphia, USA

Abstract. We study the problem of synthesizing recursive Datalog programs from examples. We propose a constraint-based synthesis approach that uses an SMT solver to efficiently navigate the space of Datalog programs and their corresponding *derivation trees*. We demonstrate our technique's ability to synthesize a range of graph-manipulating recursive programs from a small number of examples. In addition, we demonstrate our technique's potential for use in *automatic construction of program analyses* from example programs and desired analysis output.

1 Introduction

The program synthesis problem—as studied in verification and AI—involves constructing an executable program that satisfies a specification. Recently, there has been a surge of interest in *programming by example* (PBE), where the specification is a set of input–output examples that the program should satisfy [2,7,11,14,22]. The primary motivations behind PBE have been to (*i*) allow end users with no programming knowledge to automatically construct desired computations by supplying examples, and (*ii*) enable automatic construction and repair of programs from tests, e.g., in test-driven development [23].

In this paper, we present a constraint-based approach to synthesizing Datalog programs from examples. A Datalog program is comprised of a set of *Horn clauses* encoding *monotone, recursive constraints* between relations. Our primary motivation in targeting Datalog is to expand the range of synthesizable programs to the new domains addressed by Datalog. Datalog has been used in information extraction [28], graph analytics [4,26,32], and in specifying static program analyses [29,33], amongst others. We believe a PBE approach to Datalog has the potential to simplify programming in an exciting range of applications. We demonstrate how our approach can automatically synthesize a popular static analysis from examples. We envision a future in which developers will be able to automatically synthesize static analyses by specifying examples of information they would like to compute from their code. For instance, the synthesizer can live in the background of an IDE and learn what kind of information a developer likes to extract. Our approach is a concrete step towards realizing these goals.

To synthesize Datalog programs, we exploit a key technical insight: We are searching for a Datalog program whose least fixpoint—maximal *derivation tree*—includes all the positive examples and none of the negative ones. Encoding the

J.C. Beck (Ed.): CP 2017, LNCS 10416, pp. 689–706, 2017.
DOI: 10.1007/978-3-319-66158-2_44

search space of all Datalog programs and their fixpoints in some first-order theory results in a complex set of constraints. Instead, we construct a set of quantifier-free constraints that encode (*i*) all sets of clauses up to a given size and (*ii*) all *derivations*—proof trees—of a fixed size for all those clauses. In other words, we encode *underapproximations of the least fixpoints*. We then employ an *inductive synthesis loop* (as shown in Fig. 1) to ensure the program is correct and restart otherwise.

Our choice of a constraint-based synthesis technique is advantageous in (*i*) simulating execution of Datalog programs and (*ii*) steering synthesis towards desirable programs. First, we exploit the axioms of the McCarthy's first-order theory of arrays [18] to encode Datalog proof trees. Second, we define the notion of *clause templates*: additional constraints that impose a certain structure on synthesized clauses. Clause templates (*i*) constrain the search space and (*ii*) steer the synthesizer towards programs satisfying certain properties: for example, if we want programs in the complexity class NC—i.e., efficiently parallelizable—we can apply a template that ensures that all clauses are *linear*.

The field of inductive logic programming (ILP) has extensively studied the problem of inducing logic programs from examples [8,20]. Generally, the emphasis there has been on synthesizing classifiers, and therefore more examples are used and not all examples need be classified correctly. Our emphasis here is on programming-by-example, where the user provides a small number of examples and we want to match all of them. Our technical contribution can be viewed as a novel ILP technique that completely delegates the combinatorial search to an off-the-shelf SMT solver. To the best of our knowledge, this is the first such use of SMT solvers in synthesizing logic programs. We refer to Sect. 7 for a detailed comparison with related works.

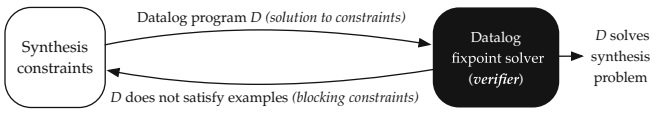


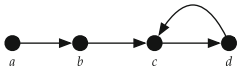
Fig. 1. High-level view of inductive synthesis loop.

Contributions. First, we demonstrate that constraint solving can be applied to the problem of synthesizing recursive Datalog programs. Second, we demonstrate how to constrain the search using logical encodings of clause templates. Third, we implement our approach and use it to synthesize a collection of recursive Datalog programs. In addition to efficiently synthesizing a range of standard Datalog programs, we demonstrate our approach's novel ability to synthesize program analyses from examples.

2 Overview and Examples

2.1 Datalog Overview

Datalog is a logic programming language where programs are composed of a set of Horn clauses over relations. For illustration, let us fix the domain (*universe*) to be $\mathcal{U} = \{a, b, c, d, e, \ldots\}$. Suppose that we are given the binary relation $E = \{(a, b), (b, c), (c, d), (d, c)\}$. We can think of relations as (hyper-)graphs, where nodes are elements of the universe $\mathcal{U}$ and (hyper-)edges denote that a tuple is in the relation. Pictorially, we can view E as representing a graph, where there is an edge from node x to node y *iff* $(x, y) \in E$, i.e., $E(x, y)$ is a *fact*.



To compute the transitive closure of the *input relation* E, we can write the Datalog program in Fig. 3(a), where T is an *output relation* that will contain the transitive closure after executing the program. X, Y, and Z are interpreted as universally quantified variables. For instance, the second clause says: *for all values of* X, Y *and* Z *picked from* $\mathcal{U}$*, if* $(X, Z) \in E$ *and* $(Z, Y) \in T$*, then* (X, Y) *must also be in* T.

One can view the execution of a Datalog program as a sequence of *derivations*, where in each step we add a new tuple to the output relation, until we reach a fixpoint. Figure 2 pictorially illustrates the process of deriving the transitive closure for our example. T starts out as the empty set, denoted T_0. By instantiating variables in the first Horn clause with constants, we can derive the edge (a, b) and add it to T, resulting in T_1. After 9 derivations, we arrive at the fixpoint, T_9, which is the full transitive closure.

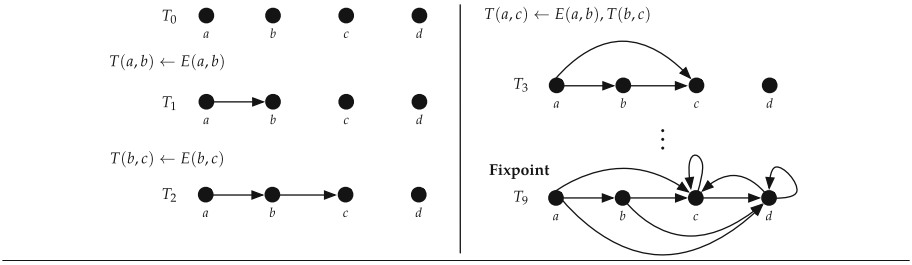


Fig. 2. Derivation sequence for transitive closure example.

2.2 Illustrative Examples

Transitive Closure. Assume we have the same input relation E as above—the graph for which we want to compute the transitive closure. We can now supply *positive* and *negative* examples of what edges should appear in T. For instance,

$$Ex^{+} = \{(a, b), (b, c), (a, c), (a, d)\}, \qquad Ex^{-} = \{(a, a)\}$$

The synthesis problem is: Find a set of Horn clauses C, defining the relation T, such that: (i) $Ex^+ \subseteq T$, and (ii) $Ex^- \cap T = \emptyset$. In other words, we want all positive examples to appear in T, but none of the negative ones.

Our synthesis technique employs an inductive synthesis loop, where in each iteration (i) a set of clauses C are *synthesized*, and (ii) C are *verified* to ensure that they derive all positive examples and none of the negative ones. We illustrate two iterations below.

Iteration 1: Synthesis Phase. To synthesize a set of clauses C, we fix the maximum number of clauses in C and the maximum number of atoms in the body of a clause. Assume we fix the number of clauses to 2 and the number of atoms to 2. Then we are looking a set of two clauses, where each clause is of the form $\blacklozenge(\bullet,\bullet) \leftarrow \blacklozenge(\bullet,\bullet), \blacklozenge(\bullet,\bullet)$. Intuitively, we would like to replace the $\blacklozenge$'s with relation symbols and the $\bullet$'s with variables. To do so, the synthesis phase constructs a set of constraints Φ_{cl}, where every model of Φ_{cl} is one possible *completion* of the above. A naïve way to proceed here is to simply sample models of Φ_{cl} and verify whether the completion derives all positive examples and none of the negative ones. This *guess-and-check* process would take a very long time due to the large number of possible Datalog programs comprised of two clauses.

(a) $T(X,Y) \leftarrow E(X,Y).$
$T(X,Z) \leftarrow E(X,Y), T(Y,Z).$

(b) $T(X,Y) \leftarrow E(X,Y).$
$T(X,Z) \leftarrow E(X,Y), E(Y,Z).$

(c) $T(X,Y) \leftarrow E(X,Y).$
$T(X,Z) \leftarrow T(X,Y), T(Y,Z).$

Fig. 3. Transitive closure example.

Therefore, we need to be able to add a new constraint specifying that the least fixpoint of the completed Horn clauses should contain all the positive examples and none of the negative ones. This, however, is a complex constraint, as it requires encoding the least fixpoint in first-order SMT theories. Instead, we create a weaker constraint, one that encodes *every possible derivation of some finite length d, for every possible completion of the above clauses*. That is, instead of encoding derivations up to fixpoint, we fix a bound d, thus encoding an *under-approximation of the least fixpoint*. These *simulation constraints* Φ_{sim} allow us to look for a completion that has a *high chance* of solving the synthesis problem. Specifically, we can now find a model for $\Phi_{cl} \wedge \Phi_{sim} \wedge Ex$, where Ex is a constraint that specifies that none of the negative examples are derived in the bounded derivation, and *most* of the positive examples are derived—in other words, we want to maximize the number of derived positive examples. This is because not all positive examples may be derivable in the bounded derivation. A possible solution for the above constraints is the set of clauses shown in Fig. 3(b).

Iteration 1: Verification Phase. The verification phase will compute the fixpoint of these clauses and determine that they do not derive all the positive examples. As a result, the verification phase produces a *blocking constraint* Φ_{neg} that avoids all similar sets of clauses.

Iteration 2: In the second iteration, the synthesis phase computes a new set of clauses that satisfy the following constraints: $\Phi_{cl} \wedge \Phi_{sim} \wedge Ex \wedge \Phi_{neg}$. As a result, it might synthesize the correct set of clauses in Fig. 3(c).

Notice that the second clause is *non-linear*, meaning that an output relation, T, appears more than once in its body. Due to the symbolic encoding, it is simple to impose additional constraints that steer synthesis towards programs of a specific form: we call these constraints *clause templates*. For instance, if we impose a template specifying that all clauses are linear, then we synthesize the equivalent transitive closure program in Fig. 3(a).

Andersen's Pointer Analysis. In static program analysis, many analyses are routinely written as Datalog programs. A given program to be analyzed is represented as a set of input relations. The Horn clauses then *compute* the results of the static analysis from these input relations. Pointer analysis is a popular target for Datalog, where the output relation is an over-approximation of which variables point to which other variables.

We can specify a slice of the desired output of a static analysis, and have our synthesizer *automatically* detect and produce the desired analysis in the form of a Datalog program. Indeed, we show that our technique is able to synthesize *Andersen's pointer analysis* [3] from examples (shown above). Specifically, here we specify examples of tuples that should or should not appear in the relation *pt*, where *pt(a,b)* specifies that variable a points to (the location of) variable b in the program. The rest of the relations are input relations specifying the program to be analyzed. For example, $addressOf(a, b)$ indicates that there is a statement in the program of the form `a = &b`.

$$pt(X, Y) \leftarrow addressOf(X, Y).$$
$$pt(X, Z) \leftarrow assign(X, Y), pt(Y, Z).$$
$$pt(W, Z) \leftarrow store(X, Y), pt(Y, Z), pt(X, W).$$
$$pt(X, W) \leftarrow load(X, Y), pt(Y, Z), pt(Z, W).$$

3 Preliminaries

Horn Clauses. A *term* t is either a variable $X, Y, Z, \ldots$, or a constant $a, b, c, \ldots$. A *predicate symbol* P is associated with an arity $arity(P)$. An *atom* is an application of a predicate symbol to a vector of variables and constants, e.g., $P(X, Y, a)$ for a predicate P with arity 3. A *ground atom* is an application of a predicate symbol to constants, e.g., $P(a_1, \ldots, a_n)$, where $\{a_i\}_i$ are constants. A *substitution* θ is a mapping from variables to constants. Applying θ to an atom yields a ground atom. For example, if $\theta = \{X \mapsto a, Y \mapsto b\}$, then $H(X, Y)\theta$ is the ground atom $H(a, b)$. When clear from context, we simplify notation to $H\theta$.

A *Horn clause* c is of the form: $H(\boldsymbol{X}) \leftarrow B_1(\boldsymbol{X_1}) \wedge \ldots \wedge B_n(\boldsymbol{X_n})$, where $H(\boldsymbol{X}), B_1(\boldsymbol{X_1}), \ldots, B_n(\boldsymbol{X_n})$ are atoms. The atom $H(\boldsymbol{X})$ is called the *head* of the clause, denoted $head(c)$; the set of atoms $\{B_i(\boldsymbol{X_i})\}$ are the *body* of the clause, denoted $body(c)$. As is standard, we replace conjunctions ($\wedge$) in the body of with commas (,). We use C to denote a finite set of Horn clauses $\{c_1, \ldots, c_n\}$.

Herbrand Interpretations. We define the semantics of Horn clauses using *Herbrand interpretations*. First, assume we have a fixed *Herbrand Universe* $\mathcal{U}$, which is a set of *constants* that can appear in atoms, e.g., $\mathcal{U} = \{a, b, c, \ldots\}$. A *Herbrand interpretation* I of a set of Horn clauses C is a set of ground atoms with constants drawn from $\mathcal{U}$. For example, an interpretation I of our transitive closure example (Sect. 2) could be: $\{T(a, b), E(b, c), T(c, d)\}$.

Definition 1 (Herbrand models and minimality). *A Herbrand interpretation M for a set of clauses C is a* Herbrand model *for C* iff *for every clause $H \leftarrow B_1, \ldots, B_n \in C$, for all substitutions θ, if $\{B_1\theta, \ldots, B_n\theta\} \subseteq I$, then $H\theta \in I$. A Herbrand model M is a* minimal model *for C* iff *for all $M' \subset M$, M' is not a Herbrand model of C.*

Datalog Programs. A *Datalog program* C is a finite set of Horn clauses. The predicates of the program can be partitioned into two disjoint sets, $\mathcal{R}_{in}(C)$ and $\mathcal{R}_{out}(C)$: $\mathcal{R}_{in}(C)$ are the predicates that appear only in the bodies of clauses in C, and are called the *input relations*. $\mathcal{R}_{out}(C)$ are the predicates that appear at the heads of clauses in C, and are called the *output relations*.

Semantics and Derivations. The input of a Datalog program C is a finite set of *facts* F, which are ground atoms over the input relations $\mathcal{R}_{in}(C)$. A Herbrand model M of C with input F is a model of C such that $F \subseteq M$. The interpretation of a Datalog program C with input F is its minimal model M. Computing the minimal model M is done using the clauses in C to *derive* all possible facts until the least fixpoint is reached. We denote the minimal model M as $C(F)$.

There always exists a unique minimal model, thus, semantics are well-defined. Figure 4 encodes the least fixpoint computation of $C(F)$ as two rules that monotonically populate the model M with more facts. The rule INIT initializes M to the set of facts F. The rule DERIVE uses clauses in C to derive a new fact to be added to M. Observe that (i) the set M is monotonically increasing and (ii) the fixpoint computation eventually terminates, as the derived facts can only contain constants from the set of facts F, which is finite.

Definition 2 (Derivation sequence). *Given a Datalog program C with input F, a* derivation sequence *is a sequence of sets of ground atoms: $M_0 \xrightarrow{c_{i_1}, \theta_1} M_1 \xrightarrow{c_{i_2}, \theta_2} M_2 \xrightarrow{c_{i_3}, \theta_3} \cdots \xrightarrow{c_{i_n}, \theta_n} M_n$, where $M_0 = F$, and M_j is the set of ground facts resulting from applying* DERIVE *to M_{j-1} with the clause $c_{i_j} \in C$ and substitution θ_j. A* maximal derivation sequence *is one where* DERIVE *cannot be applied to M_n, i.e., $M_n = C(F)$.*

Datalog Synthesis Problem. A *Datalog synthesis problem* S, or synthesis problem for short, is a tuple $(\mathcal{R}, F, E)$, where: (i) $\mathcal{R} = (\mathcal{R}_{in}, \mathcal{R}_{out})$ is a pair of input and output predicate sets that are disjoint. (ii) F is a finite set of facts—ground atoms over predicates in $\mathcal{R}_{in}$; (iii) $E = (E^+, E^-)$: E^+ is a finite set of *positive examples*, which are ground atoms over predicates in $\mathcal{R}_{out}$. E^- is a finite set of *negative examples*, which are also ground atoms over predicates in $\mathcal{R}_{out}$. We assume that $E^+ \cap E^- = \emptyset$.

$$\frac{}{M \leftarrow F}\ \text{INIT} \qquad \frac{H \leftarrow B_1, \ldots, B_n \in C \quad \{B_1\theta, \ldots, B_n\theta\} \subseteq M \quad H\theta \notin M}{M \leftarrow M \cup \{H\theta\}}\ \text{DERIVE} \qquad \theta \text{ is an arbitrary substitution}$$

Fig. 4. Rules for deriving minimal Herbrand model.

Definition 3 (Solution to Datalog synthesis problem). *A* solution *to a synthesis problem $S = (\mathcal{R}, F, E)$ is a Datalog program C with $\mathcal{R}_{in}(C) = \mathcal{R}_{in}$ and $\mathcal{R}_{out}(C) = \mathcal{R}_{out}$ such that the following two conditions hold: (*i*) $E^+ \subseteq C(F)$, i.e., all positive examples are in the minimal model of C; and (*ii*) $E^- \cap C(F) = \emptyset$, i.e., C does not derive any of the negative examples.*

4 Constraint-Based Synthesis Algorithm

We now formally define our synthesis algorithm. Recall that, given a synthesis problem $S = (\mathcal{R}, F, E)$, our goal is to discover a set of clauses C where the least fixpoint $C(F)$ contains all positive examples and none of the negative ones.

To avoid encoding least fixpoints, we will encode derivations of a fixed size—i.e., we encode under-approximations of the least fixpoint—and search for a set of clauses with a bounded derivation that derives *most* positive examples and none of the negative ones. In Sects. 4.1 and 4.2, we show how to encode the space of clauses and bounded derivations. In Sect. 4.3, we present an inductive synthesis loop that alternates between synthesizing clauses and verifying them until arriving at a solution.

4.1 Clause Constraints

Preliminaries. We describe here the *clause constraints*, a set of first-order constraints that define the space of all possible Datalog programs of a given size.

Throughout this section we shall assume a fixed Datalog synthesis problem $S = (\mathcal{R}, F, E)$, where $\mathcal{R}_{in} = \{R_1, \ldots, R_n\}$ and $\mathcal{R}_{out} = \{R_{n+1}, \ldots, R_m\}$. Without loss of generality, we shall assume that all predicates are of arity 2. In Sect. 6, we describe how we implement the algorithm for arbitrary arities. Our goal is to synthesize a solution C. We shall fix the *maximum number of clauses*, $n_c > 0$, to appear in C and a *maximum number of body atoms per clause*, $n_b > 0$. We will construct the clause constraints such that they capture every set of n_c clauses $C = \{c_1, \ldots, c_{n_c}\}$, where for each $c_i \in C$, $|body(c_i)| = n_b$.

Variables and Constraints. For each clause c_i, for $i \in [1, n_c]$, we will introduce the following integer variables:

$$h_i, b_{i,1}, \ldots, b_{i,n_b} \qquad (\mathcal{V}_{\text{PREDS}})$$

$$vh_{i,1}, vh_{i,2}, vb_{i,1}, vb_{i,2}, \ldots, vb_{i,2n_b-1}, vb_{i,2n_b} \qquad (\mathcal{V}_{\text{ARGS}})$$

The variables h_i denote the predicate symbol in the head of the clause c_i; similarly, $b_{i,j}$ denote the j'th predicate symbol in the body of c_i. Specifically, the value of the variable will be the index of the predicate symbol to appear at that location. For instance, if $h_2 = 5$, then the head of clause c_2 will be $R_5 \in \mathcal{R}_{out}$. The variables $\mathcal{V}_{\text{ARGS}}$ denote the arguments (variables) in the atoms of the clauses. For instance, $vh_{i,1}$ and $vh_{i,2}$ denote the arguments to the head of clause c_i.

Since heads of clauses can only be output predicates, and body predicates can be any predicate in $\mathcal{R}_{in} \cup \mathcal{R}_{out}$, we formulate the following constraints:

$$\varphi^h_{cl} \triangleq \bigwedge_{i \in [1,n_c]} n+1 \leq h_i \leq m \qquad \varphi^b_{cl} \triangleq \bigwedge_{i \in [1,n_c]} \bigwedge_{j \in [1,n_b]} 1 \leq b_{i,j} \leq m$$

We do not impose any constraints on $\mathcal{V}_{\text{ARGS}}$; we will simply use their values to partition arguments into equivalence classes. For instance, if $vh_{i,1} = vh_{i,2}$, the head of c_i will be an atom of the form $R(X, X)$, for some predicate $R \in \mathcal{R}_{out}$; otherwise, if $vh_{i,1} \neq vh_{i,2}$, it would be of the form $R(X, Y)$. Finally, the clause constraints are defined as follows: $\boxed{\Phi_{cl} \triangleq \varphi^h_{cl} \wedge \varphi^b_{cl}}$

Denotation and Properties. A model m of Φ_{cl}, denoted $m \models \Phi_{cl}$, maps every variable in $\mathcal{V}_{\text{PREDS}} \cup \mathcal{V}_{\text{ARGS}}$ to an integer. We now show how to transform a model m into a set of clauses C. We start by defining the function $[\![.]\!]_m$ as follows:

$$[\![h_i]\!]_m = R_{m(h_i)} \quad [\![b_i]\!]_m = R_{m(b_i)} \quad [\![vh_i]\!]_m = vmap(vh_i) \quad [\![vb_i]\!]_m = vmap(vb_i)$$

where $m(x)$ is the value of variable x in model m. Let us partition $\mathcal{V}_{\text{ARGS}}$ into equivalence classes, defined by $m(.)$—i.e., $x, y \in \mathcal{V}_{\text{ARGS}}$ are equivalent *iff* $m(x) = m(y)$. We shall now assign to each equivalence class a unique argument from the set $\{X, Y, Z, \ldots\}$. The function *vmap* maps each variable in $\mathcal{V}_{\text{ARGS}}$ to the argument assigned to its equivalence class. Using $[\![.]\!]_m$, the head of clause c_i is $[\![h_i]\!]([\![vh_{i,1}]\!], [\![vh_{i,2}]\!])$, and the j'th body atom of c_i is $[\![b_{i,j}]\!]([\![vb_{i,2j-1}]\!], [\![vb_{i,2j}]\!])$. We abuse notation and use $[\![m]\!]$ to represent the set of clauses C denoted by m.

Example 1. The above constraints and their solution are best demonstrated through a simple example. Suppose that $\mathcal{R}_{in} = \{R_1\}$ and $\mathcal{R}_{out} = \{R_2\}$, and suppose that $n_c = 1$ and $n_b = 2$. Φ_{cl} will then be $\varphi^h_{cl} \wedge \varphi^b_{cl}$, where $\varphi^h_{cl} \triangleq 2 \leq h_1 \leq 2$ and $\varphi^b_{cl} \triangleq 1 \leq b_{1,1} \leq 2 \wedge 1 \leq b_{2,2} \leq 2$. Suppose we solve Φ_{cl} and get the model $m \models \Phi_{cl}$:

$$m = \begin{bmatrix} h_1 \mapsto 2 & b_{1,1} \mapsto 1 & b_{1,2} \mapsto 2 & vh_{1,1} \mapsto 1 \\ vh_{1,2} \mapsto 3 & vb_{1,1} \mapsto 1 & vb_{1,2} \mapsto 2 & vb_{1,3} \mapsto 2 \quad vb_{1,4} \mapsto 3 \end{bmatrix}$$

The denotation $[\![m]\!]$ is the clause $R_2(X, Z) \leftarrow R_1(X, Y), R_2(Y, Z)$. Observe that the first argument of the head and the first argument of the first body atom are the same; this is because $m(vh_{1,1}) = m(vb_{1,1})$. Observe also that the predicate symbol in the head is R_2 and the first symbol in the body is R_1; this is because $m(h_1) = 2$ and $m(b_{1,1}) = 1$.

Theorem 1. *Let $\mathcal{C}$ be the set of all Datalog programs with n_c clauses, n_b atoms per clause, and no constants in atoms. Let $\mathcal{L}$ be the set of models of Φ_{cl}. Then, for each $C \in \mathcal{C}$, there exists a model $m \in \mathcal{L}$ such that $[\![m]\!]$ is equivalent to C.*

4.2 Simulation Constraints

Arrays and Monotonic Derivations. The goal of the simulation constraints is to encode all derivation sequences of the set of clauses represented by the clause constraints, Φ_{cl}. Due to the complexity of encoding all maximal derivation sequences (least fixpoints), we place a bound d on the number of derivations.[1] That is, the simulation constraints will encode all derivations with exactly d steps. It is critical to recall that a derivation, as we define it in this paper, always produces a new fact. Contrast this with the standard database-theoretic definition, where we can derive the same fact multiple times.

Recall that, given a Datalog program C with input F, a derivation sequence of length d is $M_0 \xrightarrow{c_{i_1},\theta_1} M_1 \xrightarrow{c_{i_2},\theta_2} M_2 \xrightarrow{c_{i_3},\theta_3} \cdots \xrightarrow{c_{i_d},\theta_d} M_d$. Thus, we will create a set of constraints Φ_{sim} that encode *all possible* derivations of length d from *all possible* sets of clauses C defined by Φ_{cl}.

A key observation in our technique is that M_i grows monotonically, that is, $\forall i \in [0, d-1].M_i \subsetneq M_{i+1}$. We exploit this property of Datalog to encode the set of true facts after every derivation using *McCarthy's theory of arrays* [18]. An array $arr : X \to Y$ is a map from some domain X to another domain Y. The i'th element of an array is denoted $arr[i]$. We shall therefore use arrays to represent input and output relations. Specifically, the arrays will be of the type $\mathcal{U}^2 \to \mathbb{B}$, that is, from pairs of elements of the universe to a Boolean value indicating whether the pair is in the relation. The axiom of the theory of arrays that allows us to model derivations is *read-over-write*. Specifically, read-over-write allows us to model adding one element to the relation, without explicitly having to state the *frame condition*—that all other array elements remain unchanged.

We decompose the definition of simulation constraints into (i) constraints encoding the initial *state* of all relations, (ii) constraints encoding a single application of DERIVE, and (iii) constraints encoding derivation sequences.

Encoding the Initial State. For each $R_i \in \mathcal{R}_{in}$, we create an array variable

$$in_i : \mathcal{U}^2 \to \mathbb{B} \qquad (\mathcal{V}_{\text{INRELS}})$$

The universe $\mathcal{U}$ is set to be all constants appearing in facts F and examples E.[2] For each output relation $R_i \in \mathcal{R}_{out}$, we create a set of arrays:

$$out_{i,0}, \ldots, out_{i,d} \qquad (\mathcal{V}_{\text{OUTRELS}})$$

[1] Encoding maximal derivations requires unrollings up to the size of the Herbrand base, along with universal quantification.

[2] For Datalog without constants, we can assume w.l.o.g. that the constants in the examples E are a subset of the constants in F.

where $out_{i,j}$ will represent what facts have been derived over R_i after the first j applications of DERIVE. The input and output arrays are constrained as follows:

$$\varphi^{\mathcal{R}_{in}}_{init} \triangleq \bigwedge_{R_i \in \mathcal{R}_{in}} \bigwedge_{\boldsymbol{d} \in \mathcal{U}^2} in_i[\boldsymbol{d}] \iff R_i(\boldsymbol{d}) \in F \quad \varphi^{\mathcal{R}_{out}}_{init} \triangleq \bigwedge_{R_i \in \mathcal{R}_{out}} \bigwedge_{\boldsymbol{d} \in \mathcal{U}^2} \neg out_{i,0}[\boldsymbol{d}]$$

Encoding a Single Derivation. We now show how to encode a single step of the derivation sequence (the i'th derivation): $M_{i-1} \longrightarrow M_i$. We define the formula $derive_{i,j}$ to encode the effect of applying clause j in the i'th derivation.

To formally define $derive_{i,j}$, we need to first introduce a set of variables representing the substitution θ_i that is used in the i'th derivation. Specifically, we introduce the following variables of type $\mathcal{U}$, for $i \in [1, d]$:

$$sh_{i,1}, sh_{i,2}, sb_{i,1}, sb_{i,2}, \ldots, sb_{i,2n_b-1}, sb_{i,2n_b} \qquad (\mathcal{V}_{\text{SUBS}})$$

where $(sh_{i,1}, sh_{i,2})$ denote the substitutions to the arguments in the head of the clause used in the i'th derivation, and $(sb_{i,2j-1}, sb_{i,2j})$ denote the substitutions to the arguments in the j'th body atom of the clause used in the i'th derivation. We constrain these variables such that they adhere to the arguments of the clause used in the i'th derivation. For example, if a body atom is $R(X, X)$, then we want to ensure that any substitution is of the form $R(a, a)$, for $a \in \mathcal{U}$. Therefore, we introduce the constraint $latches_{i,j}$, which indicates that, if any two arguments in atoms of clause j are the same variable, then they should always get the same substitution at position i in the derivation:

$$latches_{i,j} \triangleq \bigwedge_{vx_{j,k}, vx_{j,l} \in \mathcal{V}_{\text{ARGS}}} vx_{j,k} = vx_{j,l} \Rightarrow \sigma(vx_{j,k}) = \sigma(vx_{j,l})$$

where the notation $vx_{j,k}$ denotes any variable $vh_{j,k}$ or $vb_{j,k}$ in $\mathcal{V}_{\text{ARGS}}$, and the function σ is defined such that $\sigma(vh_{j,k}) = sh_{i,k}$ and $\sigma(vb_{j,k}) = sb_{i,k}$; that is, σ encodes the correspondence between the argument variables of the j'th clause and the substitution variables in the i'th derivation.

Now, $derive_{i,j}$ is a conjunction of two constraints: (i) $derive^b_{i,j}$, which specifies that all ground atoms in the body of clause j should be *true* in M_{i-1}, and (ii) $derive^h_{i,j}$, which specifies the new fact derived by applying the clause j at point i of the derivation. (We ensure that no fact is derived more than once.)

$$\begin{aligned} derive^b_{i,j} \triangleq & \bigwedge_{k \in [1,n]} \bigwedge_{l \in [1,n_b]} b_{j,l} = k \Rightarrow in_k[(sb_{i,2l-1}, sb_{i,2l})] \\ & \wedge \bigwedge_{k \in [n+1,m]} \bigwedge_{l \in [1,n_b]} b_{j,l} = k \Rightarrow out_{k,i-1}[(sb_{i,2l-1}, sb_{i,2l})] \\ derive^h_{i,j} \triangleq & \bigwedge_{k \in [n+1,m]} h_j = k \Rightarrow (\neg out_{k,i-1}[(sh_{i,1}, sh_{i,2})] \wedge out_{k,i}[(sh_{i,1}, sh_{i,2}) \mapsto \mathit{true}]) \end{aligned}$$

Encoding All Derivation Sequences. Now that we have defined how to encode a single step of the derivation, we can present the encoding of a derivation sequence of a fixed length d. First, we introduce the following integer variables:

$$s_1, \ldots, s_d \qquad (\mathcal{V}_{\text{DERIVCLS}})$$

where s_i encodes which clause is applied in the i'th point in the derivation sequence. Since Φ_{cl} fixes the number of clauses to n_c, we require the condition $\varphi^c_{sim} \triangleq \bigwedge_{i\in[1,d]} 1 \leq s_i \leq n_c$.

We now encode the effect of an application of DERIVE. The following constraint specifies, for every value s_i could take (from 1 to n_c), the effect on the output arrays in the i'th step of the derivation sequence.

$$\varphi^{der}_{sim} \triangleq \bigwedge_{i\in[1,d]} \bigwedge_{j\in[1,n_c]} s_i = j \Rightarrow \mathit{derive}_{i,j} \wedge \mathit{latches}_{i,j}$$

Finally, the simulation constraints are defined as follows:

$$\boxed{\Phi_{sim} \triangleq \varphi^{\mathcal{R}_{in}}_{init} \wedge \varphi^{\mathcal{R}_{out}}_{init} \wedge \varphi^{c}_{sim} \wedge \varphi^{der}_{sim}}$$

Correctness. The following theorem states correctness of simulation constraints by showing that, for a fixed set of clauses C, the models of Φ_{sim} have a one-to-one correspondence with the derivations of C of length d. Intuitively, the facts true in the output relation after d steps of a derivation are encoded in the input arrays in_i and final output arrays $out_{i,d}$. Given a model $m \models \Phi_{sim}$, we define $\mathit{final}(m)$ to denote the set of all facts at the end of the derivation defined by m: $\mathit{final}(m) = \{R_k(\boldsymbol{a}) \mid \boldsymbol{a} \in \mathcal{U}^2, m(out_{k,d}(\boldsymbol{a})) = \mathit{true} \vee m(in_k(\boldsymbol{a})) = \mathit{true}\}$.

Theorem 2. *Let $m \models \Phi_{cl}$. Let T be the set of all unique derivation sequences of $[\![m]\!]$ of length d. Let $\mathcal{L}$ be the set of all models $m' \models \Phi_{sim}$, where m' agrees with m on valuations of $\mathcal{V}_{\text{PREDS}}, \mathcal{V}_{\text{ARGS}}$. There is a bijection $f : \mathcal{L} \to T$ s.t., for all $m \in \mathcal{L}$ and $t \in T$, if $f(m) = t$ then $\mathit{final}(m) = M_d$ (final set of facts in t).*

4.3 Inductive Synthesis Loop

We now present our inductive synthesis loop (Fig. 5), given a synthesis problem $S = (\mathcal{R}, F, E)$. We fix the maximum number of clauses n_c, the maximum number of body atoms n_b, and we assume that the simulation is of length $d \leq |E^+|$ (otherwise, the simulation constraint may be UNSAT). SYNTH begins by constructing the clause and simulation constraints, Φ_{cl} and Φ_{sim}. It then employs a synthesize–verify loop.

Synthesis Phase. In line 6, SYNTH finds a model m for the constraints, which denotes a set of clauses $C = [\![m]\!]$. This can be performed using an off-the-shelf SMT solver. We impose two additional constraints. First, ψ^- ensures that no negative examples in E^- are derived in the d steps of the derivation sequence. Second, ψ^+_{soft} is a soft constraint that attempts to *maximize* the number of positive examples derived in the d steps of the derivation sequence. This is because *not all* positive examples may be derivable in d derivations.

Verification Phase. In line 8, SYNTH verifies whether C results in a solution to the synthesis problem. Specifically, it computes the fixpoint $C(F)$ and checks

whether all positive examples are in the fixpoint and none of the negative ones. If so, a solution is found and SYNTH terminates. The verification step can be performed using an off-the-shelf Datalog solver.

Blocking Constraints. If verification fails, we create a set of constraints, $block(m)$, that removes sets of Horn clauses equivalent to $[\![m]\!]$. Specifically, we first characterize a set of models whose denotation is equivalent to m:

$$\bigwedge_{i\in[1,n_c]} \left(\bigwedge_{v\in\mathcal{V}^i_{\text{PREDS}}} v = m(v) \wedge \bigwedge_{v,v'\in\mathcal{V}^i_{\text{ARGS}}, m(v)=m(v')} v = v' \right)$$

where $\mathcal{V}^i_{\text{ARGS}}$ and $\mathcal{V}^i_{\text{PREDS}}$ denote the respective subsets of $\mathcal{V}_{\text{ARGS}}$ and $\mathcal{V}_{\text{PREDS}}$ of the i'th clause. Therefore, the above constraint specifies all models whose denotation is syntactically equivalent to $[\![m]\!]$, modulo variable renaming. $block(m)$ is the negation of the above constraint. Note that characterizing all models whose denotation is equivalent to $[\![m]\!]$ is an undecidable problem [1].

The following theorem states soundness and completeness of SYNTH, relative to a fixed n_c and n_b. Note that, in point 2, if SYNTH terminates with no solution, then this means that we have proven non-existence of a solution with $\leq n_c$ clauses and $\leq n_b$ atoms. Point 2 is true because all programs that are smaller than n_c and n_b can be written as a program with exactly n_c clauses and n_b body atoms—simply by duplicating clauses and body atoms.

Theorem 3 (Soundness and completeness). *(1) If* SYNTH(S) *returns a Datalog program D, then D is a solution to S. (2) If* SYNTH(S) *terminates with no solution, then no solution exists with $\leq n_c$ clauses and $\leq n_b$ atoms per body of each clause. (3)* SYNTH(S) *terminates in finitely many steps.*

```
1:  function SYNTH(Synthesis problem S)
2:      Construct Φ_cl and Φ_sim for S
3:      Φ_neg ← true
4:      ψ⁻ ← ⋀_{R_i(d)∈E⁻} ¬out_{i,d}[d]
5:      ψ⁺_soft ← maximize |{R_i(d) ∈ E⁺ | out_{i,d}[d] = true}|
6:      while ∃m ⊨ Φ_cl ∧ Φ_sim ∧ Φ_neg ∧ ψ⁻ ∧ ψ⁺_soft do
7:          C ← [[m]]
8:          if E⁺ ⊆ C(F) and E⁻ ∩ C(F) = ∅ then
9:              return C
10:         Φ_neg ← Φ_neg ∧ block(m)
11:     return no solution exists for S
```

Fig. 5. Inductive synthesis loop.

5 Encoding Templates

We now present *clause templates:* additional constraints that exploit the use of the symbolic encoding to impose a certain structure on the synthesized clauses.

Non-recursive Clauses. The most natural clause template is the one that ensures that at least one of the clauses is a *base case*—with no output relation in the body. To define this template, we designate one of the clauses (say the first) to be the base case. Recall that the predicate symbols appearing in the body of the first clause are $b_{1,1}, \ldots, b_{1,n_b}$, where each b variable holds a value from 1 to m indicating the index of the predicate symbol. Since all indices of input relations are in $[1, n]$, all we need to impose is the following constraint: $\text{BASECASE} \triangleq \bigwedge_{i \in [1,n_b]} b_{1,i} \leq n$. If we specify that every clause is non-recursive, then we syntactically restrict the solution to be in the class of *Unions of Conjunctive Queries* (UCQs), a fundamental query class [1], since it captures the class of positive SQL queries.

Linear Clauses. A clause is *linear* when there is at most one occurrence of an output predicate in its body. Linear Datalog programs—a strict subset of Datalog—are in the complexity class NC (Nick's class): the set of problems solvable in polylogarithmic time with a polynomial number of processors. Informally, a problem in NC is inherently parallel. In addition to their theoretical niceties, linear Datalog programs have also proven useful in distributed processing [27]. In order to synthesize linear programs, we impose the following constraint:

$$\text{LINEAR} \triangleq \bigwedge_{i \in [1,n_c]} \neg \bigvee_{j,k \in [1,n_b], j \neq k} b_{i,j} \geq n+1 \wedge b_{i,k} \geq n+1$$

The above constraint states that for every clause i, no two predicate symbols in the body, $b_{i,j}$ and $b_{i,k}$, refer to output relations.

Connected Clauses. It is most often the case that arguments in the head of a clause also appear in its body. For instance, the clause $H(X, Y) \leftarrow B(Z, W)$ will end up deriving every possible tuple in $\mathcal{U}^2$ (assuming B is not empty), which is unlikely a program of interest. To avoid programs that are able to derive all possible tuples, we can impose the following constraint:

$$\text{CONN} \triangleq \bigwedge_{i \in [1,n_c]} \left(\bigvee_{j \in [1,2n_b]} vh_{i,1} = vb_{i,j} \wedge \bigvee_{k \in [1,2n_b]} vh_{i,2} = vb_{i,k} \right)$$

6 Implementation and Evaluation

We have implemented the presented synthesis technique in a new tool called *Zaatar*. Zaatar utilizes the open-source Z3 SMT solver [19] for satisfiability checking and evaluating Datalog programs (using Z3's fixpoint engine [13]).

Our implementation takes as input a synthesis problem where relations can be of arbitrary arities. The encoding in Sect. 4 assumes that all relations are binary. We extend the encoding to assume that all relations have the same arity, the maximum arity amongst all relations in $\mathcal{R}$. For example, if the maximum arity is 4, we describe a binary relation $R(X, Y)$ by disregarding the variables in $\mathcal{V}_{\text{ARGS}}$ that represent the third and fourth arguments.

Benchmarks. We collected a set of Datalog programs comprised of recursive and non-recursive programs. The benchmarks are fully listed in Table 1. The non-recursive benchmark programs include (*i*) path extraction programs (`path3` and `path4`, which extract all paths of length 3 or 4 from a graph); (*ii*) cycle and triangle extraction from a graph (`cycle` and `triangle`); and (*iii*) path extraction with alternating edge colors (`redblue` and `redblueUnd`). Our recursive benchmarks include standard Datalog programs, like transitive closure of a graph (`TC` and `TCUnd`) and *same generation* (`samegen`), which extracts all individuals of the same generation from a family tree. In addition to the standard graph-manipulating Datalog programs, we also synthesized (*i*) *least inductive invariant generation* (`leastInvariant`), which, given a finite-state program and its least inductive invariant, returns the *initiation* and *consecution* rules defining an inductive invariant [5]; and (*ii*) pointer analysis programs (`andersenFull` and `andersenSimple`).

Experimental Results. The experimental results are shown in Table 1. For each synthesis problem, we instantiated it with a small number of facts F in the input relations (3–8 facts per benchmark). Then, we supplied a small and sufficient number of positive and negative examples that describe the problem.

Table 1. Experimental results. Mac OS X 10.11; 4 GHz Intel Core i7; 16 GB RAM.

Benchmark	Time (s)	#Iters.	$\lvert F\rvert$	$\lvert\mathcal{R}_{in}\rvert$	$\lvert\mathcal{R}_{out}\rvert$	$\lvert E^+\rvert$	$\lvert E^-\rvert$	n_c	n_b	Description
Non-recursive benchmarks										
`path3`	0.13	1	4	1	1	1	0	1	3	Extract all pairs of vertices (x, y) where x reaches y in 3 steps
`path4`	0.23	1	5	1	1	2	0	1	4	Same as `path3`, but 4 steps
`redblue`	0.24	1	5	2	1	1	0	1	3	Extract all pairs of vertices (x, y) where x reaches y using one red edge followed by a blue edge
`redblueUnd`	1.19	1	5	2	1	2	0	2	3	Extract all pairs of vertices (x, y) where x (y) reaches y (x) using one red edge followed by a blue edge
`triangle`	1.32	6	5	1	1	2	8	1	3	Extract all triples (x, y, z) that form a triangle
`cycles`	0.01	1	6	1	1	3	0	1	2	Extract all vertices x where x is in a cycle of length 2
Recursive benchmarks										
`TC`	0.57	1	3	1	1	3	0	2	2	Compute transitive closure of a directed graph
`TCUnd`	4.25	1	3	1	1	6	0	3	2	Compute the undirected transitive closure of a directed graph
`pathsEven`	0.61	1	4	1	1	3	0	2	3	Extract all pairs of vertices (x, y) where x reaches y through a path of even length
`pathsOdd`	14.37	1	5	1	1	5	5	2	3	Same as `pathsEven`, but for odd length paths
`pathsMod3`	0.66	2	6	1	1	2	6	2	4	Extract all pairs of vertices (x, y) where x reaches y through a path of length divisible by 3
`pathsMod4`	3.43	5	8	1	1	2	5	2	5	Same as `pathsMod3`, but for paths divisible by 4
`redblueRec`	0.29	1	6	2	1	3	1	2	2	Extract all pairs of vertices (x, y) where x reaches y through a path with alternating red and blue edges
`redblueRecSep`	5.18	1	4	2	2	6	0	4	2	Extract two relations: one all pairs of vertices (x, y) where x reaches y through red edges only, and another with blue edges only
`samegen`	3.53	1	6	1	1	4	1	2	3	Extract all pairs of individuals (x, y) who are from the same generation in a family tree
`leastInvariant`	1.07	5	5	2	1	4	2	2	2	Compute the least inductive invariant of a finite-state program
`andersenSimple`	33.02	1	6	3	1	6	0	3	3	Andersen's pointer analysis (without load instructions)
`andersenFull`	115.87	1	7	4	1	7	1	4	3	Andersen's pointer analysis

The number of positive/negative examples required per benchmark are shown in Table 1 (see columns $|E^+|$ and $|E^-|$). For all benchmarks, we fixed the derivation bound d to be the number of positive examples $|E^+|$. The only templates we used were BASECASE (to force a base case in recursive programs) and CONN (see Sect. 5). (Without the CONN template imposed, most recursive benchmarks do not terminate within a reasonable amount of time: they keep synthesizing trivial programs with head arguments disjoint from body arguments, and therefore keep iterating through the synthesis loop.)

Our results indicate that our approach can synthesize non-trivial programs within a small amount of time. For most benchmarks, Zaatar synthesizes the correct solution within 0–5 s. The longest running benchmark—Andersen's analysis, `andersenFull`—requires around 2 min. Furthermore, the correct solution is usually discovered within the first iteration of SYNTH (see column #Iters. in Table 1). This result indicates that, using our approach, a small number of examples is sufficient to describe a non-trivial graph-based computation.

For most benchmarks, a very small number of negative examples is required—often none. We notice that the numbers of required negative examples increases with the size of the desired program (as defined by n_c and n_b) and the arities of relations. For example, in the `triangle` benchmark, there are many non-recursive programs correlating triples of vertices; we thus needed to supply 8 negative examples to ensure that the program is indeed only extracting triples that form triangles. Other benchmarks that require multiple negative examples are `pathsOdd` and `pathsMod3`, which have 3 and 4 atoms in the bodies of their clauses.

Discussion. Our results demonstrate the merit of our approach at synthesizing a range of different Datalog programs in a small amount of time and with a small number of examples. It is important, however, to state the limitations. First, our search process imposes an upper bound on the number of clauses and atoms that can appear in programs. Second, as the size of the desired program increases, the number of examples required increases (and, therefore, the size of the derivation bound d), thus stressing the SMT solver.

7 Related Work and Discussion

Synthesis of Recursive Programs. We are not the first to synthesize recursive functions. Recent synthesis works from the community have focused on functional programs—for example, [2,10,15,16,22,24,24,31]. Our work synthesizes recursive graph-/relation-manipulating programs. While we can encode relations in a functional programming language, the synthesis task becomes tedious. Datalog is a more natural fit for relation manipulation.

Inductive Logic Programming. Our synthesis target is closest in nature to the rich field of inductive logic programming (ILP) [8,20]. Generally speaking, a primary focus of ILP research has been on inducing *theories*, e.g., a program explaining a biological process. The formulation, however, is very similar to

our setting, with the addition of *background knowledge* encoded as clauses. The search technique in ILP is often a bottom-up or top-down search for a theory. In the bottom-up setting, the synthesizer begins with no clauses, and incrementally grows the set of clauses (by climbing up a subsumption lattice of clauses) using the provided positive and negative examples; the top-down setting proceeds in the opposite direction. Our search strategy is rather different: we consider all examples at once and present a novel encoding that delegates the process to an SMT solver. Naturally, our approach directly benefits from future advances in SMT techniques. Additionally, ILP techniques often require a large number of examples. In our approach, our goal is to utilize only a small number of examples.

Synthesis of recursive clauses has not received as much attention in ILP. Flener and Yilmaz [9] provide a survey of ILP in the recursive setting. A recent line of work by Muggleton et al. presents a *meta-interpretive learning* (MIL) technique for synthesizing recursive clauses for various domains [6,17,21]. Compared to our technique, MIL requires *meta-rules* (similar to our clause templates) that can constrain the search towards recursive clauses. Meta-rules, however, are very specific: they (*i*) fix all the variables in the rule, (*ii*) exactly specify which relations in the rule are recursive, and (*iii*) fix the size of a rule. Thus, the only parameters of the search are the relations to appear in the body and the head of a clause. Our templates can encode meta-rules and are strictly more general than meta-rules. In practice, we do not restrict the variables at all (we only use CONN to eliminate ill-formed rules). Further, MIL tools like Metagol require a total ordering on the Herbrand base, which might not exist for certain examples, e.g., transitive closure and Andersen, where graphs are cyclic. Nonetheless, MIL also addresses *predicate invention*, the problem of introducing new predicates. This is an interesting and difficult problem that relates to synthesizing auxiliary procedures in the more general program synthesis setting.

Constraint-Based Synthesis. Our technique is inspired by symbolic synthesis techniques [12,14,25,30].Techniques like [12] encode all loop-free programs of up to a certain size, along with the examples, as a formula to be solved by an SMT solver. We encode the search for Horn clauses along with bounded derivations of the clauses as a first-order formula. Since efficient symbolic encodings can only express loop/recursion-free executions, we use bounded derivations to induce recursive programs symbolically, without having to encode the least fixpoint.

Acknowledgements. This work is supported by NSF awards 1566015, 1652140, and a Google Faculty Research Award.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
2. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_67

3. Andersen, L.O.: Program analysis and specialization for the C programming language. Ph.D. thesis, University of Cophenhagen (1994)
4. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and implementation of the logicblox system. In: Proceedings of 2015 ACM SIGMOD International Conference on Management of Data, pp. 1371–1382. ACM (2015)
5. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer Science and Business Media, Heidelberg (2007). doi:10.1007/978-3-540-74113-8
6. Cropper, A., Muggleton, S.H.: Learning efficient logical robot strategies involving composable objects. In: Proceedings of 24th International Joint Conference Artificial Intelligence (IJCAI 2015), pp. 3423–3429 (2015)
7. Cropper, A., Tamaddoni-Nezhad, A., Muggleton, S.H.: Meta-interpretive learning of data transformation programs. In: Proceedings of 24th International Conference on Inductive Logic Programming (2015)
8. De Raedt, L.: Logical and Relational Learning. Springer Science and Business Media, Heidelberg (2008)
9. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: achievements and prospects. JLP **41**, 141–195 (1999)
10. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: POPL. ACM (2016)
11. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. CACM **55**, 97–105 (2012)
12. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI (2011)
13. Hoder, K., Bjørner, N., De Moura, L.: μZ–an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_36
14. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (2010)
15. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: an explanation based generalization approach. JMLR **7**, 429–454 (2006)
16. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)
17. Lin, D., Dechter, E., Ellis, K., Tenenbaum, J.B., Muggleton, S.: Bias reformulation for one-shot function induction. In: ECAI, pp. 525–530 (2014)
18. McCarthy, J.: Towards a mathematical science of computation. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds.) Program Verification. SCS, vol. 14, pp. 35–56. Springer, Dordrecht (1993)
19. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
20. Muggleton, S.: Inductive logic programming. N. Gener. Comput. **8**, 295–318 (1991)
21. Muggleton, S.H., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: application to grammatical inference. Mach. Learn. **94**, 25–49 (2014)
22. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: PLDI (2015)
23. Perelman, D., Gulwani, S., Grossman, D., Provost, P.: Test-driven synthesis. In: PLDI (2014)

24. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 522–538. ACM (2016)
25. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). doi:10.1007/978-3-319-21668-3_12
26. Seo, J., Guo, S., Lam, M.S.: Socialite: datalog extensions for efficient social network analysis. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 278–289. IEEE (2013)
27. Shaw, M., Koutris, P., Howe, B., Suciu, D.: Optimizing large-scale semi-naïve datalog evaluation in hadoop. In: Barceló, P., Pichler, R. (eds.) Datalog 2.0 2012. LNCS, vol. 7494, pp. 165–176. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32925-8_17
28. Shen, W., Doan, A., Naughton, J.F., Ramakrishnan, R.: Declarative information extraction using datalog with embedded extraction predicates. In: Proceedings of 33rd international conference on Very large data bases, pp. 1033–1044. VLDB Endowment (2007)
29. Smaragdakis, Y., Balatsouras, G., et al.: Pointer analysis. Found. Trends Program. Lang. **2**, 1–69 (2015)
30. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
31. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23702-7_23
32. Wang, J., Balazinska, M., Halperin, D.: Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. Proc. VLDB Endow. **8**, 1542–1553 (2015)
33. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144. ACM (2004)