

Recovering Explanations from Transformed Rule-Based Ontologies^{*}

Alex Ivliev^(✉) , Markus Krötzsch , and Maximilian Marx 

Knowledge-Based Systems Group, TU Dresden, Dresden, Germany
{alex.ivliev, markus.kroetzsch, maximilian.marx}@tu-dresden.de

Abstract Datalog rules are often used to define ontologies over Knowledge Graphs. Rule reasoners routinely optimise such ontologies by rewriting their rules into a form that can be evaluated more efficiently. These transformations preserve the entailed facts, but not the structure of the underlying derivations. A proof tree under the rewritten rules explains why a fact holds, but does not readily yield an explanation in terms of the original rules. We study the problem of constructing, from a proof of entailment under the rewritten rules, a proof under the original ones: we establish its computational complexity and identify two practically relevant languages for specifying proof transformations.

Keywords: Datalog · Proof transformations · Explainability

1 Introduction

Datalog and its numerous extensions form a family of rule languages that can express ontological knowledge in the Semantic Web context [42,10,28]. OWL RL and OWL EL, two profiles of the Web Ontology Language OWL 2 [49], can be captured by Datalog rules [59,39,40], and extensions of Datalog cover even larger subsets of OWL 2 [17,18]. Datalog is also used for query answering [15,16], for encoding ontological reasoning [40], for data extraction and analysis [52,25], and for transforming Knowledge Graphs [57,21]. A key benefit of Datalog as an ontology language is the inherent explainability of its consequences: any inferred fact can be justified by a *proof tree* that shows which rules were applied to which facts, tracing every inference back to the input. Computing such proof trees is well-studied for both OWL [38,33,2,3,5,47] and Datalog [23,60,30,14]. For Datalog, it usually takes the form of why-provenance, which asks for not one, but *all* proof trees of a fact; selecting the “most helpful” among them is a research topic in its own right [4,13]. Even when proof trees grow very large, summarisation and visualisation make them accessible to users [48,21,47]. Another major strength of Datalog is the availability of highly optimised engines that support efficient reasoning over knowledge graphs with billions of facts [50,58,37,35,45]. This efficiency is partly achieved by rewriting the input rules into an equivalent,

^{*} This is the technical report accompanying our ISWC’26 paper [36].

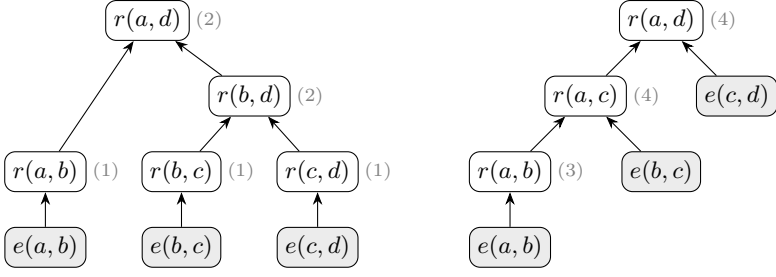


Figure 1. Proof trees for ontologies Π_1 (left) and Π_2 (right) in Example 1.

optimised ontology, using techniques such as program minimisation [55], the Magic Sets rewriting for goal-directed evaluation [8], removal of redundant parameters [53], folding and unfolding of rules [51], and propagation of filter conditions through recursive rules [32]. Some of these techniques were even discovered by studying transformations on proof trees [54,19]. Such transformations preserve the consequences of an ontology, but may substantially change the structure of its derivations, as demonstrated in the following example:

Example 1. The transitive closure r of a binary relation e can be defined by two equivalent ontologies Π_1 and Π_2 :

Ontology Π_1 (bushy):

$$r(x, y) \leftarrow e(x, y) \quad (1)$$

$$r(x, y) \leftarrow r(x, z), r(z, y) \quad (2)$$

Ontology Π_2 (left-linear):

$$r(x, y) \leftarrow e(x, y) \quad (3)$$

$$r(x, y) \leftarrow r(x, z), e(z, y) \quad (4)$$

Both ontologies derive the same r -facts on any input database. However, the bushy rule (2) allows splitting a path at any intermediate node, which may result in balanced proof trees of logarithmic depth. The left-linear rule (4) extends a path by one edge at a time, producing left-leaning chains. Figure 1 shows proof trees for $r(a, d)$ over $\mathcal{D} = \{e(a, b), e(b, c), e(c, d)\}$.

A Datalog engine may rewrite one of these ontologies into the other to evaluate it efficiently. Proof trees are then computed over the rewritten ontology, and thus fail to explain why a fact follows under the *original* one. We study the problem of recovering such an explanation, i.e., of constructing a corresponding proof tree for the original ontology from a proof tree for the transformed ontology. We establish the computational complexity of this task (Section 4) and, building on tree transducers [7] and Monadic Second-Order-definable transductions [20,24], identify two suitable formalisms for specifying proof tree transformations (Section 5). The appendix of this paper contains further examples.

2 Preliminaries

We recall the syntax of Datalog (with built-in relations) and define its semantics based on proof trees. A recent survey [41] offers a broader treatment.

Syntax We consider a vocabulary of mutually disjoint sets $\mathbf{P}$ of *predicates*, $\mathbf{C}$ of *constants*, and $\mathbf{V}$ of *variables*. We associate an *arity* $\text{ar}(p) \geq 0$ with every predicate $p \in \mathbf{P}$. A *term* is an element of the set $\mathbf{T} := \mathbf{C} \cup \mathbf{V}$. We use $\bar{t}$ to denote a list of terms $\langle t_1, \dots, t_k \rangle$ where $|\bar{t}| = k$. Similarly, we write, e.g., $\bar{x}$ or $\bar{y}$ for lists of variables. An *atom* has the form $p(\bar{t})$ where $p \in \mathbf{P}$ and $\text{ar}(p) = |\bar{t}|$. Atoms that do not contain any variables are called *facts*. The sets of all facts and all atoms are denoted $\mathfrak{F}$ and $\mathfrak{A}$, respectively. A *Datalog rule* ρ is an expression of the form

$$H \leftarrow B_1, \dots, B_n \quad (5)$$

where H and $B_1, \dots, B_n$ are atoms. We call H the *head* and $B_1, \dots, B_n$ the *body* of ρ . We use notation $\text{head}(\rho) := H$ and $\text{body}_i(\rho) := B_i$ for $1 \leq i \leq n$, and set $|\rho| := n$. We require that every variable in the head of ρ also appears in its body (*safety*). Rule bodies are allowed to be empty; in this case, the symbol $\leftarrow$ is omitted. A *program* Π is a triple $\langle R, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ consisting of a finite set R of rules, and sets $\mathbf{P}_{\text{in}}$ and $\mathbf{P}_{\text{out}}$ of *input* and *output predicates*, respectively, such that input predicates may not appear in the head of a rule.

Semantics and Proofs A (potentially infinite) set of facts $\mathcal{D}$ is a *database*. It is an *input database* for $\Pi = \langle R, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ if the predicate of every fact is from $\mathbf{P}_{\text{in}}$. We allow for infinite databases $\mathcal{D}$ to express conceptually infinite built-in relations such as $\leq$ and $\neq$. An *ontology* $\mathcal{O} = \langle \Pi, \mathcal{D} \rangle$ consists of a program Π together with an input database $\mathcal{D}$.

An *assignment* μ for a rule ρ maps every variable x in ρ to a constant $\mu(x) \in \mathbf{C}$. We extend μ to terms by setting $\mu(c) := c$ for constants $c \in \mathbf{C}$, to lists of terms by $\mu(\langle t_1, \dots, t_k \rangle) := \langle \mu(t_1), \dots, \mu(t_k) \rangle$, and to atoms by $\mu(p(\bar{t})) := p(\mu(\bar{t}))$.

An *ordered tree* is a tuple $T = \langle V, \text{root}, \text{ch} \rangle$ where V is a finite set of *nodes*, $\text{root} \in V$ is called the *root* node, and $\text{ch}: V \rightarrow V^*$ maps each node v to a finite sequence $\text{ch}(v) = \langle v_1, \dots, v_n \rangle$ of distinct *children* of v , such that every node except root appears in exactly one such sequence and root appears in none. We call v a *leaf* if $\text{ch}(v)$ is empty. A *proof tree* for a fact f over a program $\Pi = \langle R, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and a database $\mathcal{D}$ is an ordered tree $T = \langle V, \text{root}, \text{ch} \rangle$ together with labelling functions $\text{fact}: V \rightarrow \mathfrak{F}$ and $\text{rule}: V \rightarrow R \cup \{\perp\}$, such that

1. $\text{fact}(\text{root}) = f$;
2. if $\text{ch}(v) = \langle v_1, \dots, v_n \rangle$ with $n \geq 1$, then $\text{rule}(v) \in R$ with $|\text{rule}(v)| = n$, and there exists an assignment μ for $\text{rule}(v)$ such that $\mu(\text{head}(\text{rule}(v))) = \text{fact}(v)$ and $\mu(\text{body}_i(\text{rule}(v))) = \text{fact}(v_i)$ for $1 \leq i \leq n$; and
3. for every leaf v , either:
 - (a) $\text{rule}(v) = \perp$ and $\text{fact}(v) \in \mathcal{D}$, or
 - (b) $\text{rule}(v) \in R$ with $|\text{rule}(v)| = 0$ and $\text{fact}(v) = \text{head}(\text{rule}(v))$.

We say that a fact f is *derived* from Π and $\mathcal{D}$, written $\Pi, \mathcal{D} \vdash f$, if there exists a proof tree T for f over Π and $\mathcal{D}$. The *leaf database* $\mathcal{D}_T$ of a proof tree T is the set of all facts $\text{fact}(v)$ for leaves v of T with $\text{rule}(v) = \perp$. Every proof tree T for f over Π and $\mathcal{D}$ is also a proof tree for f over Π and $\mathcal{D}_T$. The *output* of Π over $\mathcal{D}$ is the set $\text{out}(\Pi, \mathcal{D}) := \{f \mid \Pi, \mathcal{D} \vdash f \text{ and the predicate of } f \text{ is in } \mathbf{P}_{\text{out}}\}$.

Remark 1. For finite input databases, $\text{out}(\Pi, \mathcal{D})$ coincides with the set of output facts in the least fixpoint of the standard immediate-consequence operator. With infinite built-in relations, practical systems ensure termination by, e.g., requiring that every variable in a built-in also occurs in a body atom with a finite predicate. Such considerations do not affect our results.

Containment Let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be programs over the same input and output predicates. We say that Π_1 is *contained* in Π_2 , written $\Pi_1 \sqsubseteq \Pi_2$, if $\text{out}(\Pi_1, \mathcal{D}) \subseteq \text{out}(\Pi_2, \mathcal{D})$ for every input database $\mathcal{D}$. The two programs are *equivalent*, written $\Pi_1 \equiv \Pi_2$, if $\Pi_1 \sqsubseteq \Pi_2$ and $\Pi_2 \sqsubseteq \Pi_1$. A *program transformation* is a function t mapping programs Π to programs $t(\Pi)$. It is *sound* if $t(\Pi) \sqsubseteq \Pi$ and *complete* if $\Pi \sqsubseteq t(\Pi)$ for all programs Π .

3 Proof Transformations

A sound program transformation produces a new program Π_1 from an original program Π_2 , such that $\Pi_1 \sqsubseteq \Pi_2$. This guarantees that for every proof tree T_1 over Π_1 there exists a proof tree T_2 over Π_2 deriving the same (output) fact over the same input database. The central question of this paper is how to obtain T_2 from T_1 , and which computational resources this requires.

In a proof tree, the same fact may be derived multiple times in independent subtrees. A more compact representation merges such redundant derivations, yielding a directed acyclic graph. A *proof DAG* for a fact f over a program Π and database $\mathcal{D}$ is a tuple $G = \langle V, \text{root}, \text{ch}, \text{fact}, \text{rule} \rangle$ satisfying the same conditions as a proof tree, except that a node may appear as a child of multiple nodes. Notions such as the leaf database $\mathcal{D}_G$ carry over from proof trees. The *unfolding* of a proof DAG G is the proof tree $\text{unfold}(G)$ obtained by recursively duplicating shared nodes. Conversely, a proof tree may be compressed into a proof DAG by merging nodes labelled with the same fact and the same rule and whose subgraphs are isomorphic. For a fact $f = p(t_1, \dots, t_k)$, we define $|f| := k$. The size of a proof tree $T = \langle V, \text{root}, \text{ch}, \text{fact}, \text{rule} \rangle$ is $|T| := \sum_{v \in V} |\text{fact}(v)|$. The facts of a proof DAG G are $\text{facts}(G) := \bigcup_{v \in V} \text{fact}(v)$, and its size is $|G| := |\{\langle v, w \rangle \in V \times V \mid v \in \text{ch}(w)\}| + \sum_{t \in \text{facts}(G)} |t|$.

Definition 1 (Proof transformation). Let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be programs with $\Pi_1 \sqsubseteq \Pi_2$. A proof transformation from Π_1 to Π_2 is a function τ that maps every proof tree T_1 for a fact f with predicate $p \in \mathbf{P}_{\text{out}}$ over Π_1 and input database $\mathcal{D}$ to a proof DAG $\tau(T_1)$ for f over Π_2 and $\mathcal{D}$. The result of this transformation is the proof tree $\text{unfold}(\tau(T_1))$.

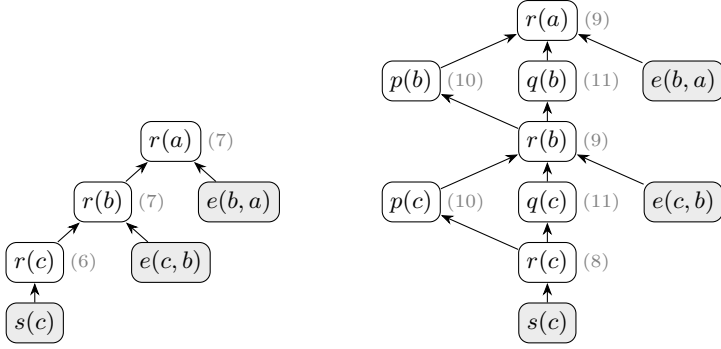


Figure 2. Proof tree T_1 for Π_1 (left) and proof DAG $\tau(T_1)$ for Π_2 (right) of Example 2

We require that $\tau(T_1)$ is a proof DAG instead of a proof tree because proof trees can be exponentially larger than the corresponding proof DAGs. The following example demonstrates such an unavoidable blowup.

Example 2. Consider the following equivalent programs Π_1 and Π_2 with input predicates $\mathbf{P}_{\text{in}} = \{s, e\}$ and output predicate $\mathbf{P}_{\text{out}} = \{r\}$.

Program Π_1 :

$$r(x) \leftarrow s(x) \quad (6)$$

$$r(y) \leftarrow r(x), e(x, y) \quad (7)$$

Program Π_2 :

$$r(x) \leftarrow s(x) \quad (8)$$

$$r(y) \leftarrow p(x), q(x), e(x, y) \quad (9)$$

$$p(x) \leftarrow r(x) \quad (10)$$

$$q(x) \leftarrow r(x) \quad (11)$$

Both programs derive $r(x)$ for exactly those x reachable from some s -fact along e -edges. However, Π_2 duplicates each derivation of $r(x)$ through the auxiliary predicates p and q . Consider $\mathcal{D} = \{s(c), e(c, b), e(b, a)\}$. A proof transformation τ from Π_1 to Π_2 must map nodes for rules (6) and (7) to nodes for rules (8) and (9), respectively, and introduce two nodes for rules (10) and (11) below each node for rule (9). Figure 2 shows the proof tree T_1 for $r(a)$ over Π_1 and $\mathcal{D}$, and the proof DAG $\tau(T_1)$ over Π_2 and $\mathcal{D}$. The unfolding $\text{unfold}(\tau(T_1))$ is exponential in the size of T_1 .

4 Complexity of Proof Transformations

We now focus on the computational problem of transforming a proof tree T_1 over Π_1 into a proof DAG T_2 over Π_2 for the same fact. A naive approach extracts the leaf database of T_1 and evaluates Π_2 from scratch, which takes time polynomial in the size of T_1 . In this section, we show that producing a valid output is PTIME-hard if T_1 is encoded as a directed acyclic graph, matching the naive upper bound, and LOGCFL-hard if T_1 is encoded as a tree.

4.1 The Transformation Problem

A proof transformation (Definition 1) assigns a fixed output proof to every input proof. Here, instead, we study the computational complexity of constructing *some* valid proof over Π_2 from a proof over Π_1 for the same fact, accepting any valid proof DAG as output. Intuitively, this captures the complexity of the best possible transformation algorithm: lower bounds apply to every proof transformation, while the upper bounds are established by a particular transformation algorithm. Formally, we define the search problem as follows.

Problem: $\text{TRANS}_{\text{enc}}(\Pi_1, \Pi_2)$ for programs $\Pi_1 \subseteq \Pi_2$
Input: A proof tree (if $\text{enc} = \text{Tree}$) or proof DAG (if $\text{enc} = \text{DAG}$) T_1 for a fact f whose predicate is in $\mathbf{P}_{\text{out}}$, over Π_1 and input database $\mathcal{D}_{T_1}$.
Output: A proof DAG T_2 for f over Π_2 and $\mathcal{D}_{T_1}$.

Note that we fix the programs $\Pi_1 \subseteq \Pi_2$ in the problem definition, so complexity is measured only in terms of the size of the input. Consequently, algorithms for $\text{TRANS}_{\text{enc}}(\Pi_1, \Pi_2)$ may exploit properties of the particular program pair. We consider both encodings $\text{enc} \in \{\text{Tree}, \text{DAG}\}$ of the input proof tree (both given as adjacency lists), because they may differ exponentially in size (see Example 2). This difference stems solely from sharing identical subderivations, making the tree encoding the more informative measure when sharing is limited.

A search problem associates every *instance* with a (non-empty) set of *solutions*. An algorithm *solves* such a problem if, for every instance, it computes a valid solution. To establish hardness, we use the following notion of reduction, adapted from the standard treatment of multivalued function problems in complexity theory [46]: An NC^1 *reduction* from a decision problem L to a search problem S is a pair of functions g and h , computable by logspace-uniform NC^1 circuit families [31], such that for every instance x of L : (1) $g(x)$ is an instance of S , and (2) for every solution y of $g(x)$, we have $h(y) = 1$ iff $x \in L$. Condition (2) quantifies over all solutions since an algorithm solving S may output any of them. A search problem S is $\mathcal{C}$ -hard under NC^1 reductions for a complexity class $\mathcal{C}$ if every decision problem in $\mathcal{C}$ reduces to S via an NC^1 reduction. We only consider NC^1 reductions here, since they are sufficiently weak to establish L-hardness, yet expressive enough to admit L-complete problems to reduce from.

4.2 Complexity Results

The naive approach gives us an upper bound for the transformation problem:

Proposition 1. *Let $\Pi_1 \subseteq \Pi_2$ be programs. Then there exists a polynomial-time algorithm that solves $\text{TRANS}_{\text{enc}}(\Pi_1, \Pi_2)$ for both encodings $\text{enc} \in \{\text{Tree}, \text{DAG}\}$.*

Proof. Given T_1 , we compute the least fixpoint of Π_2 over $\mathcal{D}_{T_1}$ in polynomial time [1], storing for each fact the rule instance that was used to derive it. We get $f \in \text{out}(\Pi_2, \mathcal{D}_{T_1})$ since $\Pi_1 \subseteq \Pi_2$. Then, the stored rule instances, starting from f , form a proof DAG T_2 for f over Π_2 and $\mathcal{D}_{T_1}$. $\square$

For the corresponding lower bound, we reduce from the Circuit Value Problem, which is known to be PTIME-complete for DAG-encoded circuits [31]. As an immediate corollary, we also obtain a L-hardness for $\text{TRANS}_{\text{Tree}}(\Pi_1, \Pi_2)$, as this problem is L-complete when the circuit is given as a tree [9]. We show a LOGCFL lower bound later.

Theorem 1. *There is a pair of programs $\Pi_1 \sqsubseteq \Pi_2$ for which the problem $\text{TRANS}_{\text{DAG}}(\Pi_1, \Pi_2)$ is PTIME-hard.*

Proof. We define programs Π_1 and Π_2 over the single output predicate **goal** and the set of input predicates $\mathbf{P}_{\text{in}} = \{\mathbf{t}_{\text{in}}, \mathbf{f}_{\text{in}}, \mathbf{and}, \mathbf{not}, \mathbf{out}\}$, where we use facts **and**(g, g_1, g_2) and **not**(g, g_1) to encode the structure of the circuit, $\mathbf{t}_{\text{in}}(g)$ and $\mathbf{f}_{\text{in}}(g)$ to declare the input gate g as true or false, and **out**(g) to mark the output gate. Program Π_2 evaluates the circuit as expected:

$$\begin{array}{ll}
 \text{true}(g) \leftarrow \mathbf{t}_{\text{in}}(g) & (12) \quad \text{false}(g) \leftarrow \mathbf{and}(g, g_1, g_2), \\
 \text{false}(g) \leftarrow \mathbf{f}_{\text{in}}(g) & (13) \quad \text{true}(g_1), \text{false}(g_2) \quad (17) \\
 \text{true}(g) \leftarrow \mathbf{and}(g, g_1, g_2), & (14) \quad \text{false}(g) \leftarrow \mathbf{and}(g, g_1, g_2), \\
 \quad \text{true}(g_1), \text{true}(g_2) & (14) \quad \text{false}(g_1), \text{false}(g_2) \quad (18) \\
 \text{false}(g) \leftarrow \mathbf{and}(g, g_1, g_2), & (15) \quad \text{true}(g) \leftarrow \mathbf{not}(g, g_1), \text{false}(g_1) \quad (19) \\
 \quad \text{false}(g_1), \text{true}(g_2) & (15) \quad \text{false}(g) \leftarrow \mathbf{not}(g, g_1), \text{true}(g_1) \quad (20) \\
 \text{goal} \leftarrow \text{false}(g), \text{out}(g) & (16) \quad \text{goal} \leftarrow \text{true}(g), \text{out}(g) \quad (21)
 \end{array}$$

Proof trees of Π_2 correspond to evaluations of the circuit encoded by the input facts, where the result of the output gate is obtained by checking the rule label at the root of the proof tree. When constructing Π_1 , we need to ensure that **goal** is only derived when Π_2 can derive a valuation for an output gate:

$$\text{maybe}(g) \leftarrow \mathbf{t}_{\text{in}}(g) \quad (22)$$

$$\text{maybe}(g) \leftarrow \mathbf{f}_{\text{in}}(g) \quad (23)$$

$$\text{maybe}(g) \leftarrow \text{maybe}(g_1), \text{maybe}(g_2), \mathbf{and}(g, g_1, g_2) \quad (24)$$

$$\text{maybe}(g) \leftarrow \mathbf{not}(g, g_1), \text{maybe}(g_1) \quad (25)$$

$$\text{goal} \leftarrow \text{maybe}(g), \text{out}(g) \quad (26)$$

Proof trees of Π_1 encode the structure of the circuit, but do not evaluate it. We can thus obtain a suitable input DAG T_1 from an arbitrary circuit by a simple relabelling that maps the gate types to the corresponding rules in Π_1 . $\square$

We now show the LOGCFL lower bound for $\text{TRANS}_{\text{Tree}}(\Pi_1, \Pi_2)$. As in the proof of Theorem 1, program Π_1 merely validates the structure of the input, while Π_2 decides a concrete problem. Here, we reduce from the word problem of a fixed context-free language.

Theorem 2. *There is a pair of programs $\Pi_1 \sqsubseteq \Pi_2$ for which the problem $\text{TRANS}_{\text{Tree}}(\Pi_1, \Pi_2)$ is LOGCFL-hard.*

Proof. Let L be a context-free language with grammar $G = \langle V, \Sigma, P, S \rangle$ and $\Sigma \subseteq \mathbf{C}$. For this reduction, we pick L to be a language whose word problem is LOGCFL-complete under NC^1 reductions [43] and assume G to be given in quadratic Greibach normal form [6], i.e., each production $p \in P$ is of the form $A \rightarrow \sigma A_1 \cdots A_n$ for $0 \leq n \leq 2$, a terminal symbol $\sigma \in \Sigma$, and nonterminal symbols $A, A_1, \dots, A_n \in V$. We use input predicates $\mathbf{P}_{\text{in}} = \{\text{first}, \text{next}, \text{last}\}$ and the output predicate goal , and encode a word $w = \sigma_1 \cdots \sigma_\ell \in \Sigma^+$ as the database

$$\mathcal{D}_w := \{\text{first}(c_1), \text{next}(c_1, \sigma_1, c_2), \dots, \text{next}(c_\ell, \sigma_\ell, c_{\ell+1}), \text{last}(c_{\ell+1})\},$$

where $\text{next}(x, \sigma, y)$ expresses that position y follows position x when reading the letter σ . Program Π_1 consists of the following rules, where rule (27) is instantiated for every $\sigma \in \Sigma$:

$$r(x, y) \leftarrow \text{next}(x, \sigma, y) \quad (27)$$

$$r(x, y) \leftarrow r(x, z), r(z, y) \quad (28)$$

$$\text{goal} \leftarrow r(x, y), \text{first}(x), \text{last}(y) \quad (29)$$

Program Π_2 includes the rules of form (27) and rule (28). For each nonterminal $A \in V$, it uses binary predicates $\mathbf{A}$ and $\bar{\mathbf{A}}$, intended to hold on a span of the word iff A can, respectively cannot, produce that span. Program Π_2 further contains the following two rules, which can be used to decide if $w \in L$:

$$\text{goal} \leftarrow \text{first}(x), \text{last}(y), \mathbf{S}(x, y) \quad (30)$$

$$\text{goal} \leftarrow \text{first}(x), \text{last}(y), \bar{\mathbf{S}}(x, y) \quad (31)$$

The predicates $\mathbf{A}$ are derived by the rules given below, which are instantiated for each production $A \rightarrow \sigma A_1 \cdots A_n \in P$:

$$\mathbf{A}(x, x_{n+1}) \leftarrow \text{next}(x, \sigma, x_1), \mathbf{A}_1(x_1, x_2), \dots, \mathbf{A}_n(x_n, x_{n+1}) \quad (32)$$

Deriving the complements $\bar{\mathbf{A}}$ is more challenging, as Π_2 must verify that *no* production of A can produce the span. For every production $p \in P$, we use binary predicate $\bar{\mathbf{p}}$ that holds on every span that cannot be produced by p . Then, Π_2 contains for each nonterminal $A \in V$ derived by productions $\mathbf{p}^1, \dots, \mathbf{p}^m \in P$:

$$\bar{\mathbf{A}}(x, y) \leftarrow \bar{\mathbf{p}}^1(x, y), \dots, \bar{\mathbf{p}}^m(x, y) \quad (33)$$

A production p with leading terminal σ fails on every span that does not start with σ . We derive these spans using an auxiliary predicate $\bar{\sigma}$:

$$\bar{\sigma}(x, y) \leftarrow \text{next}(x, \sigma', y) \quad (34)$$

$$\bar{\mathbf{p}}(x, y) \leftarrow \bar{\sigma}(x, y) \quad (35)$$

$$\bar{\mathbf{p}}(x, y) \leftarrow \bar{\sigma}(x, z), r(z, y) \quad (36)$$

for all $\sigma \neq \sigma' \in \Sigma$ and productions $p \in P$ with lead terminal σ . The remaining causes of failure depend on the shape of p . A production $p = A \rightarrow \sigma$ additionally fails on every span of length at least two:

$$\bar{\mathbf{p}}(x, y) \leftarrow \text{next}(x, t, z), r(z, y) \quad (37)$$

A production $p = A \rightarrow \sigma A_1$ additionally fails on spans of length one, which are too short, and on spans whose remainder after σ cannot be produced by A_1 :

$$\bar{p}(x, y) \leftarrow \text{next}(x, t, y) \quad (38)$$

$$\bar{p}(x, y) \leftarrow \text{next}(x, \sigma, z), \bar{A}_1(z, y) \quad (39)$$

Finally, consider a production $p = A \rightarrow \sigma A_1 A_2$. Spans of length one or two are again too short:

$$\bar{p}(x, y) \leftarrow \text{next}(x, t, y) \quad (40)$$

$$\bar{p}(x, y) \leftarrow \text{next}(x, t, z), \text{next}(z, t', y) \quad (41)$$

On longer spans, p fails iff *every* way of splitting the word between A_1 and A_2 fails. We capture this condition with two auxiliary predicates. The first predicate $\bar{p}_s(z, s, y)$ states that the span from z to y cannot be split at position s , since A_1 fails on the part before s or A_2 fails on the part after it. The second predicate $\bar{p}_{\leq s}(z, s, y)$ states that all split positions up to s fail:

$$\bar{p}_s(z, s, y) \leftarrow \bar{A}_1(z, s), r(s, y) \quad (42)$$

$$\bar{p}_s(z, s, y) \leftarrow r(z, s), \bar{A}_2(s, y) \quad (43)$$

$$\bar{p}_{\leq s}(z, s, y) \leftarrow \text{next}(z, t, s), \bar{p}_s(z, s, y) \quad (44)$$

$$\bar{p}_{\leq s}(z, s', y) \leftarrow \bar{p}_{\leq s}(z, s, y), \text{next}(s, t, s'), \bar{p}_s(z, s', y) \quad (45)$$

$$\bar{p}(x, y) \leftarrow \text{next}(x, \sigma, z), \bar{p}_{\leq s}(z, s, y), \text{next}(s, t, y) \quad (46)$$

For containment, consider an input database $\mathcal{D}$ with $\text{goal} \in \text{out}(\Pi_1, \mathcal{D})$. Then there are constants $c_1, \dots, c_{\ell+1}$ and $s_1, \dots, s_\ell$ such that $\mathcal{D}$ contains the facts $\text{first}(c_1)$, $\text{last}(c_{\ell+1})$, and $\text{next}(c_i, s_i, c_{i+1})$ for all $1 \leq i \leq \ell$, spelling out the word $w = s_1 \cdots s_\ell$. If $w \in L$, we can use rules of form (32) to simulate productions of G , deriving $A(c_i, c_{j+1})$ if the nonterminal A can produce the span $s_i \cdots s_j$. Hence, we have $\Pi_2, \mathcal{D} \vdash \bar{S}(c_1, c_{\ell+1})$, and thus $\Pi_2, \mathcal{D} \vdash \text{goal}$ by rule (30).

For the case $w \notin L$ we show, by induction on the length of a span $c_i, \dots, c_{j+1}$, that Π_2 derives $\bar{A}(c_i, c_{j+1})$ for every nonterminal $A \in V$ that does not produce $s_i \cdots s_j$. It suffices by rule (33) to derive $\bar{p}(c_i, c_{j+1})$ for every production $p = A \rightarrow \sigma A_1 \cdots A_n \in P$ that cannot produce $s_i \cdots s_j$. If $s_i \neq \sigma$, this follows from rules (35) and (36). Otherwise, we distinguish the three shapes of p . For $n = 0$, the production fails only if the span is longer than one letter, which is covered by rule (37). For $n = 1$, it fails if the span consists of a single letter (rule (38)), or if A_1 does not produce the remainder $s_{i+1} \cdots s_j$, in which case the induction hypothesis yields $\bar{A}_1(c_{i+1}, c_{j+1})$ and rule (39) applies. For $n = 2$, spans of one or two letters are too short (rules (40) and (41)); on longer spans, every split of the remainder must fail. By the induction hypothesis, rules (42) and (43) then derive $\bar{p}_s(c_{i+1}, c_k, c_{j+1})$ for every split position c_k , so that rules (44) and (45) derive $\bar{p}_{\leq s}(c_{i+1}, c_j, c_{j+1})$ and therefore rule (46) applies. We obtain $\Pi_2, \mathcal{D} \vdash \bar{S}(c_1, c_{\ell+1})$ for the full span, and hence $\text{goal} \in \text{out}(\Pi_2, \mathcal{D})$ by rule (31).

On the encoding $\mathcal{D}_w$ of a word w exactly one of $\bar{S}(c_1, c_{\ell+1})$ and $\bar{S}(c_1, c_{\ell+1})$ is derivable. Hence the rule at the root of every proof tree for goal over Π_2 and $\mathcal{D}_w$ is determined by whether $w \in L$. $\square$

5 Proof Transformation Languages

We now turn to the question of *how* to specify proof transformations and identify two practical transformation languages. In Section 5.1, we show that the notion of uniform containment between programs admits local transformations defined by proof tree homomorphisms. We consider Monadic Second-Order Logic (MSO) interpretations as a transformation language in Section 5.2, which define transformations computable in logarithmic space for tree-encoded inputs.

5.1 Proof Tree Homomorphisms

A simple class of proof transformation arises from *uniform containment*, a notion introduced by Sagiv for minimising Datalog programs [55]. In this setting, an atom or a rule is considered redundant if removing it from the program results in an equivalent program. Checking containment between two Datalog programs is undecidable [56], so uniform containment can be used as a decidable alternative.

Definition 2 (Uniform containment). *Let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be programs over the same input and output predicates. Program Π_1 is uniformly contained in Π_2 , denoted $\Pi_1 \sqsubseteq_u \Pi_2$, if for every database $\mathcal{D}$ (not only input databases for Π_1 or Π_2), we have $\text{out}(\Pi_1, \mathcal{D}) \subseteq \text{out}(\Pi_2, \mathcal{D})$.*

Note that containment compares the output of the programs on input databases, while for uniform containment $\text{out}(\Pi_1, \mathcal{D}) \subseteq \text{out}(\Pi_2, \mathcal{D})$ must hold over all databases $\mathcal{D}$. Hence, $\Pi_1 \sqsubseteq_u \Pi_2$ implies $\Pi_1 \sqsubseteq \Pi_2$, but the converse does not hold in general. Let $\omega: \mathbf{V} \rightarrow \mathbf{C}$ be an injective mapping from variables to constants. We can decide whether $\Pi_1 \sqsubseteq_u \Pi_2$ by verifying for every rule $\rho_1 \in R_1$, that its head can be derived from its body using Π_2 , i.e., that $\Pi_2, \mathcal{D} \vdash \omega(\text{head}(\rho_1))$ for $\mathcal{D} = \{\omega(\text{body}_i(\rho_1)) \mid 1 \leq i \leq |\rho_1|\}$ [55]. This leads to a class of proof transformations that replace each node of a proof tree for Π_1 with the matching derivation of the same fact in Π_2 . This corresponds naturally to the notion of tree homomorphisms [27], which we adapt for proof trees as follows.

Definition 3 (Proof tree template). *Let $\Pi = \langle R, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be a program and ρ be a rule. A proof tree template T for ρ over Π is a tuple $\langle V, \text{root}, \text{ch}, \text{atom}, \text{rule} \rangle$ where $\langle V, \text{root}, \text{ch} \rangle$ is an ordered tree with labelling functions $\text{atom}: V \rightarrow \mathfrak{A}$ and $\text{rule}: V \rightarrow R$, such that*

1. $\text{atom}(\text{root}) = \text{head}(\rho)$;
2. if $\text{ch}(v) = \langle v_1, \dots, v_n \rangle$ with $n \geq 1$, then $\text{rule}(v) \in R$ with $|\text{rule}(v)| = n$, and there exists a substitution $\theta_v: \mathbf{V} \rightarrow \mathbf{T}$ such that $\theta_v(\text{head}(\text{rule}(v))) = \text{atom}(v)$ and $\theta_v(\text{body}_i(\text{rule}(v))) = \text{atom}(v_i)$ for $1 \leq i \leq n$; and
3. for every leaf node ℓ , there is an index ρ_ℓ with $1 \leq \rho_\ell \leq |\rho|$ such that $\text{atom}(\ell) = \text{body}_{\rho_\ell}(\rho)$.

Let T be a proof tree template for a rule ρ with $|\rho| = n$ over Π and $L \subseteq V$ the set of leaves of T . Furthermore, let $T_1, \dots, T_n$ be proof DAGs over Π with

$T_i = \langle V_i, \text{root}_i, \text{ch}_i, \text{fact}_i, \text{rule}_i \rangle$ that do not share nodes with T or with each other. If there is a substitution $\sigma: \mathbf{V} \rightarrow \mathbf{C}$ with $\sigma(\text{body}_i(\rho)) = \text{fact}(\text{root}_i)$ for $1 \leq i \leq n$, we write $T[T_1, \dots, T_n]$ to denote the proof DAG S that results from uniformly replacing each node in $\langle V', \text{root}', \text{ch}', \text{fact}', \text{rule}' \rangle$ with a fresh copy, where

- $V' := (V \cup \bigcup_{i=1}^n V_i) \setminus L$;
- $\text{root}' := \text{root}$;
- $\text{ch}' := \text{ch} \cup \bigcup_{i=1}^n \text{ch}_i$ replacing every occurrence of a leaf node $\ell \in L$ by root_{ρ_ℓ} ;
- $\text{fact}'(v) := \sigma(\text{atom}(v))$ for $v \in V$ and $\text{fact}'(v) := \text{fact}_i(v)$ for $v \in V_i$; and
- $\text{rule}'(v) := \text{rule}(v)$ for $v \in V$ and $\text{rule}'(v) := \text{rule}_i(v)$ for $v \in V_i$.

Further, we use T_v to denote the subgraph of a proof DAG T rooted at v . We can now define proof tree homomorphisms as follows.

Definition 4 (Proof tree homomorphism). Let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be programs with $\Pi_1 \sqsubseteq \Pi_2$, and let $\{S_\rho\}_{\rho \in R_1}$ be a family of proof tree templates where S_ρ is a proof tree template for $\rho \in R_1$ over Π_2 . The proof tree homomorphism induced by $\{S_\rho\}_{\rho \in R_1}$ is a proof transformation τ defined as follows. Given a proof tree $T = \langle V, \text{root}, \text{ch}, \text{fact}, \text{rule} \rangle$ over Π_1 and some database $\mathcal{D}$, we obtain $\tau(T)$ recursively:

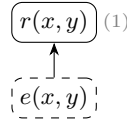
1. if v is a leaf node, then $\tau(T_v) := T_v$;
2. otherwise, if $\text{ch}(v) = \langle v_1, \dots, v_n \rangle$ and $\text{rule}(v) = \rho$, then

$$\tau(T_v) := S_\rho[\tau(T_{v_1}), \dots, \tau(T_{v_n})]$$

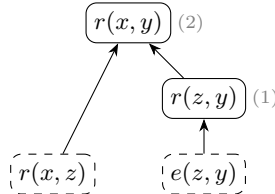
From the above characterisation of uniform containment, we immediately obtain:

Theorem 3. Let Π_1 and Π_2 be programs. Then there exists a proof tree homomorphism from Π_1 to Π_2 if and only if $\Pi_1 \sqsubseteq_u \Pi_2$.

Example 3. Recall the programs Π_1 (bushy) and Π_2 (left-linear) from Example 1. We have $\Pi_2 \sqsubseteq_u \Pi_1$, so we can obtain a proof tree homomorphism from Π_2 to Π_1 . For the base rule (3), the template $T_{(3)}$ applies the corresponding base rule (1) of Π_1 directly:



For the recursive rule (4), the template $T_{(4)}$ uses the bushy rule (2) at the root and derives $r(z, y)$ from the second body atom $e(z, y)$ via (1):



Appendix A contains further examples of proof tree homomorphisms.

5.2 Transformations Based on Monadic Second-Order Logic

Proof tree homomorphisms are restricted to local transformations and cannot consider the global structure of the input tree. For instance, the reverse direction of Example 3, transforming bushy proof trees of Π_1 into left-linear proof trees of Π_2 , cannot be realised by proof tree homomorphisms. MSO transductions [20,24] are a well-studied formalism for tree-to-tree transformations that can express such global restructurings. However, the output of MSO transductions is at most linear in the input, while proof transformations may require polynomial growth:

Example 4. Consider the following programs Π_1 and Π_2 :

Program Π_1 :

$$r(x) \leftarrow s(x) \quad (47)$$

$$r(y) \leftarrow r(x), n(x, y) \quad (48)$$

$$\text{goal} \leftarrow r(x), e(x) \quad (49)$$

Program Π_2 :

$$r(x, x) \leftarrow s(x) \quad (50)$$

$$r(x, y') \leftarrow r(x, y), n(y, y') \quad (51)$$

$$r(x', y') \leftarrow r(x, y), n(x, x'), s(y'), e(y) \quad (52)$$

$$\text{goal} \leftarrow r(x, y), e(x), e(y) \quad (53)$$

Program Π_1 derives **goal** if there is a path of n -edges from some s -fact to an e -fact. Program Π_2 implements a counter, traversing all possible pairs of values along the n -chain. Hence, proof trees for **goal** over Π_1 are of size linear in the input, while proof trees for **goal** over Π_2 are of size quadratic in the input.

We accommodate for such polynomial growth of degree k by using *MSO interpretations* [11], a generalisation of MSO transductions in which output elements are defined as k -tuples of input elements.

Encoding proof trees as logical structures The goal is to define proof transformations as MSO formulae. With each proof tree, we therefore associate a relational structure consisting of the following relations over a universe of tree nodes and constants:

- $\text{child}_i(x, y)$ if x is the i th child of y ;
- $\text{rule}_\rho(x)$ if rule ρ is applied to obtain x ; and
- $\text{arg}_j(x, d)$ if the j th argument of the fact at x is d .

We can now express proof trees as databases over this signature.

Definition 5. Let $T = \langle V, \text{root}, \text{ch}, \text{fact}, \text{rule} \rangle$ be a proof tree over a program $\Pi = \langle R, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and some database $\mathcal{D}$. The associated interpretation of T is the database $\mathcal{I}(T)$, which contains exactly the following facts:

- $\text{child}_i(v, w)$ for every $v, w \in V$ such that $\text{ch}(w) = \langle v_1, \dots, v_n \rangle$ with $v = v_i$;
- $\text{rule}_\rho(v)$ for every $v \in V$ and $\rho \in R$ such that $\text{rule}(v) = \rho$; and
- $\text{arg}_j(v, d)$ for every $v \in V$ such that $\text{fact}(v) = p(d_1, \dots, d_m)$ with $d = d_j$.

MSO transformations We can now use MSO formulae to specify a proof transformation by defining the output structure over the nodes of the input proof tree. For an introduction to Monadic Second-Order Logic, we refer to Libkin [44]. The following definition for MSO interpretations is adapted to our setting from Gallot *et al.* [26].

Definition 6 (MSO interpretation). *Let $\Pi = \langle R, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be a program. Let n be the maximum arity of any rule in R and let m be the maximum arity of any atom occurring in R . A k -dimensional MSO interpretation for Π with $k \geq 1$ consists of*

- a finite set $\mathcal{C}$ of components,
- for each component $\alpha \in \mathcal{C}$, an MSO formula $\varphi_{\text{dom}}^\alpha(\bar{x})$ with k free variables,
- for each $\alpha \in \mathcal{C}$ and $\rho \in R$, an MSO formula $\varphi_\rho^\alpha(\bar{x})$ with k free variables,
- for each $\alpha, \beta \in \mathcal{C}$ and $1 \leq i \leq n$, an MSO formula $\varphi_{\text{child}}^{\alpha, \beta, i}(\bar{x}, \bar{y})$ with $2k$ free variables, and
- for each $\alpha \in \mathcal{C}$, and $1 \leq j \leq m$, an MSO formula $\varphi_{\text{arg}}^{\alpha, j}(\bar{x}, d)$ with $k + 1$ free variables.

Conceptually, an MSO interpretation defines an output tree whose nodes are tuples of input nodes selected by $\varphi_{\text{dom}}^\alpha(\bar{x})$ for each component $\alpha \in \mathcal{C}$. The remaining formulae define the labellings of children, rules, and facts. We write $\mathcal{D}, \bar{v} \models \varphi(\bar{x})$ to denote that the MSO formula φ is satisfied in the database $\mathcal{D}$ when the free variables $\bar{x}$ are assigned to the elements $\bar{v}$.

Definition 7 (MSO proof transformation). *Let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ and $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be programs with $\Pi_1 \subseteq \Pi_2$. A proof transformation τ from Π_1 to Π_2 is an MSO proof transformation if there is a k -dimensional MSO interpretation for Π_2 and some $k \geq 1$ and some set of components $\mathcal{C}$ such that for every proof tree $T_1 = \langle V_1, \text{root}_1, \text{ch}_1, \text{fact}_1, \text{rule}_1 \rangle$ for some fact f with predicate $p \in \mathbf{P}_{\text{out}}$ over Π_1 and database $\mathcal{D}$, there is a proof DAG $G_2 = \tau(T_1) = \langle V_2, \text{root}_2, \text{ch}_2, \text{fact}_2, \text{rule}_2 \rangle$ for f over Π_2 and $\mathcal{D}$ where*

- the nodes of G_2 are pairs $\langle \alpha, \bar{v} \rangle \in \mathcal{C} \times V_1^k$ with $\mathcal{I}(T_1), \bar{v} \models \varphi_{\text{dom}}^\alpha(\bar{x})$;
- the rule at node $\langle \alpha, \bar{v} \rangle$ is the unique $\rho \in R_2$ such that $\mathcal{I}(T_1), \bar{v} \models \varphi_\rho^\alpha(\bar{x})$;
- node $\langle \beta, \bar{w} \rangle$ is the i -th child of $\langle \alpha, \bar{v} \rangle$ iff $\mathcal{I}(T_1), \bar{v}, \bar{w} \models \varphi_{\text{child}}^{\alpha, \beta, i}(\bar{x}, \bar{y})$; and
- the j -th argument of the fact at $\langle \alpha, \bar{v} \rangle$ is d iff $\mathcal{I}(T_1), \bar{v}, d \models \varphi_{\text{arg}}^{\alpha, j}(\bar{x}, d)$.

Example 5. The proof transformation from bushy trees to left-linear trees from Example 1 can be described as a 1-dimensional MSO proof transformation with components $\mathcal{C} = \{\alpha\}$ as follows.

We assume a relation $v \leq w$ that holds whenever $v = w$ or w is an ancestor of v and further define $\text{Leaf}(v) := \neg \exists u. \text{child}(u, v)$ and $\text{Root}(v) := \neg \exists w. \text{child}(v, w)$ where $\text{child}(u, v) := \bigvee_i \text{child}_i(u, v)$. We then order the nodes above the leaves (derived by rule (1)) of the input tree from left to right:

$$\begin{aligned} u <_\ell v &:= \text{rule}_{(1)}(u) \wedge \text{rule}_{(1)}(v) \\ &\quad \wedge \exists p, a, b. \text{child}_1(a, p) \wedge \text{child}_2(b, p) \wedge u \leq a \wedge v \leq b \end{aligned} \tag{54}$$

We can check whether two nodes u and v are next to each other in the order

$$\text{Cons}(u, v) := u <_{\ell} v \wedge \neg \exists w. u <_{\ell} w \wedge w <_{\ell} v \quad (55)$$

We then identify the leftmost and rightmost nodes labelled with rule (1):

$$\text{First}(v) := \text{rule}_{(1)}(v) \wedge \neg \exists w. \text{Cons}(w, v) \quad (56)$$

$$\text{Last}(v) := \text{rule}_{(1)}(v) \wedge \neg \exists w. \text{Cons}(v, w) \quad (57)$$

With this in place, we can define the structure of the output tree as follows:

$$\varphi_{\text{dom}}(x) := \neg \text{rule}_{(2)}(x) \quad (58)$$

$$\varphi_{\text{child}}^1(x, y) := (\text{child}_1(x, y) \wedge \text{First}(y)) \vee \text{Cons}(x, y) \quad (59)$$

$$\varphi_{\text{child}}^2(x, y) := \neg \text{First}(y) \wedge \text{rule}_{(1)}(y) \wedge \text{child}_1(x, y) \quad (60)$$

$$\varphi_{(3)}(x) := \text{First}(x) \quad (61)$$

$$\varphi_{(4)}(x) := \exists y. \text{Cons}(y, x) \quad (62)$$

$$\begin{aligned} \varphi_{\text{arg}}^1(x, d) := & (\text{Leaf}(x) \wedge \text{arg}_1(x, d)) \\ & \vee (\text{rule}_{(1)}(x) \wedge \exists r. \text{Root}(r) \wedge \text{arg}_1(r, d)) \end{aligned} \quad (63)$$

$$\varphi_{\text{arg}}^2(x, d) := \text{arg}_2(x, d) \quad (64)$$

Complexity We now show that MSO proof transformations can be evaluated in logarithmic space when the input is encoded as a tree. Our argument uses the fact that MSO model checking over structures of bounded treewidth can be performed in logarithmic space [22]. We begin by recalling the relevant notions. Given a tree T with vertex set V and a subset $X \subseteq V$, we write $T[X]$ for the subtree of T induced by X , that is, the tree obtained by restricting T to the nodes in X while preserving the child relationship among them. For a database $\mathcal{D}$, we write $\mathcal{A}(\mathcal{D})$ for the set of all constants occurring in $\mathcal{D}$.

Definition 8 (Treewidth). *Let $\mathcal{D}$ be a database. A tree decomposition of $\mathcal{D}$ is a labelled tree $T = \langle V, E, B \rangle$ with labelling function $B: V \rightarrow \{X \mid X \subseteq \mathcal{A}(\mathcal{D})\}$ such that*

1. *for all $a \in \mathcal{A}(\mathcal{D})$, the induced subtree $T[X]$ with $X = \{v \in V \mid a \in B(v)\}$ is nonempty and connected; and*
2. *for every $p(t_1, \dots, t_k) \in \mathcal{D}$, there is a node $v \in V$ with $\{t_1, \dots, t_k\} \subseteq B(v)$.*

The width of a tree decomposition is $\max_{v \in V} |B(v)| - 1$. The treewidth of $\mathcal{D}$ is the minimal width of any of its tree decompositions; the treewidth of a proof tree T is the treewidth of $\mathcal{I}(T)$.

The treewidth of a proof tree T over Π might grow arbitrarily large due to the presence of $\text{arg}_j(v, d)$ facts in $\mathcal{I}(T)$. However, we can find a proof tree T' for the same fact such that its treewidth only depends on Π . The key observation is that we can rename apart all constants in T such that two occurrences of a constant remain equal only when the validity of the proof tree requires them to be. This results in a tree where each constant appears only within a connected region: namely, within the subtree rooted at the rule instance that first introduced it.

Lemma 1. *For every program Π there exists a constant k_Π such that for every proof tree $T = \langle V, \text{root}, \text{ch}, \text{fact}, \text{rule} \rangle$ for f over Π and database $\mathcal{D}$, there is a database $\mathcal{D}'$ and a proof tree T' for f over Π and $\mathcal{D}'$ with treewidth at most k_Π .*

Proof. In the following, we write $\text{vars}(\mathcal{E})$ and $\text{terms}(\mathcal{E})$ for the set of all variables and all terms, respectively, that appear in an expression $\mathcal{E}$. We obtain $T' = \langle V, \text{root}, \text{ch}, \text{fact}', \text{rule} \rangle$ from T , where fact' is defined recursively for each $v \in V$ as follows:

- If $v = \text{root}$, then $\text{fact}'(v) := \text{fact}(v)$.
- If v is the i -th child of $w \in V$ and $\text{rule}(w) = \rho$, let $\theta: \text{vars}(\text{head}(\rho)) \rightarrow \mathbf{C}$ be a substitution such that $\theta(\text{head}(\rho)) = \text{fact}'(w)$. Given $\text{fact}(v) = p(t_1, \dots, t_k)$ and $\text{body}_i(\rho) = p(s_1, \dots, s_k)$, we set $\text{fact}'(v) := p(t'_1, \dots, t'_k)$ where

$$t'_j = \begin{cases} \theta(s_j) & \text{if } s_j \in \text{vars}(\text{head}(\rho)) \text{ or } s_j \in \mathbf{C}; \text{ and} \\ \langle t_j, w \rangle & \text{otherwise.} \end{cases}$$

Since the number of distinct constants in any single rule instance is bounded by the number of terms appearing in the rules of Π , we can construct a tree decomposition whose bags correspond to rule instances, each containing only the constants mentioned in that instance. Let $D = \langle V, E, B \rangle$ where $\{v, w\} \in E$ if v is a child of w in T . We define $B(\ell) := \text{terms}(\text{fact}'(\ell)) \cup \{\ell\}$ for each leaf node ℓ in T ; for all other nodes $v \in V$ with $\text{ch}(v) = \langle v_1, \dots, v_n \rangle$ we set

$$B(v) := \{v\} \cup \{v_i \mid 1 \leq i \leq n\} \cup \text{terms}(\text{fact}'(v)) \cup \bigcup_{1 \leq i \leq n} \text{terms}(\text{fact}'(v_i)) \quad (65)$$

Then D defines a tree decomposition for T whose width depends only on the maximal number of distinct terms in a rule of Π , giving us the desired bound k_Π . Condition 1 of Definition 8 is satisfied because every constant is confined to a connected subtree; condition 2 holds because all constants of each fact are witnessed in the bag of the node where the corresponding rule is applied. $\square$

Since MSO model checking over structures of bounded treewidth can be performed in logarithmic space [22, Theorem 1.2], we thus obtain the following.

Theorem 4. *Let τ be an MSO proof transformation from program Π_1 to Π_2 . For any proof tree $T = \langle V, \text{root}, \text{ch}, \text{fact}, \text{rule} \rangle$ for fact f over Π_1 and database $\mathcal{D}$, the proof DAG $\tau(T)$ can be computed by an $O(\log |T|)$ space bounded transducer.*

Proof. If τ is an MSO proof transformation, there exists a k -dimensional MSO interpretation and set of components $\mathcal{C}$ satisfying Definition 7. For a proof tree T , we evaluate each MSO formula over $\mathcal{I}(T')$, where we obtain $T' = \langle V, \text{root}, \text{ch}, \text{fact}', \text{rule} \rangle$ as in Lemma 1. Note that we do not need to store T' as an intermediate step, since we can compute $\text{fact}'(v)$ for any $v \in V$ on demand in logarithmic space. For each term position j in $\text{fact}(v)$, we determine t'_j by walking upward from v toward the root, following how the variable at position j is propagated through the heads of successive rule instances. If the

variable appears in the head of the current rule, we continue upward, inheriting the corresponding position in the parent’s fact. Otherwise, the current node is the ancestor that introduced this constant, and we return the tagged pair $\langle t_j, w \rangle$.

By Lemma 1, $\mathcal{I}(T')$ has bounded treewidth, so each formula can be evaluated in logarithmic space [22, Theorem 1.2]. To instantiate each formula, we need k counters over the vertex set V for $\varphi_{\text{dom}}(\bar{x})$ and $\varphi_\rho(\bar{x})$, $2k$ counters over V for $\varphi_{\text{child}}(\bar{x}, \bar{y})$, k counters over V , and 1 counter over the set of all terms occurring in fact labels for $\varphi_{\text{arg}}(\bar{x}, d)$. This yields a proof DAG $\tau(T)$ for f over $\mathcal{D}'$. We rename each fresh constant $\langle t, w \rangle$ to t to get a proof DAG for f over $\mathcal{D}$. $\square$

While this suggests that there are program transformations that cannot be reversed using an MSO proof transformation (unless $\text{L} = \text{NL} = \text{LOGCFL}$), we find that MSO proof tree transformations are expressive enough to capture many practical transformations. We have already seen that Example 5 can be expressed; we provide further examples in Appendix B.

6 Conclusions and Outlook

We have studied the problem of reversing program transformations at the level of proof trees and established PTIME-hardness for the DAG-encoded case and LOGCFL-hardness for the tree-encoded case. We have identified two practical transformation languages: Proof tree homomorphisms are characterised by uniform containment, and capture local transformations, such as transforming left-linear proof trees into bushy proof trees (see Example 3). MSO proof tree transformations handle a strictly larger class of restructurings (such as transforming from bushy proof trees back to left-linear proof trees, see Example 5), but can still be evaluated efficiently in logarithmic space on tree-encoded input.

For future work, several directions remain open. For tree-encoded inputs, there remains a gap between the LOGCFL lower bound and the PTIME upper bound. It is unclear whether this can be closed. Similarly, a characterisation of the program transformations that admit a reversal by an MSO proof transformation, along the lines of the characterisation of first-order tree-to-tree transductions by Bojańczyk and Doumane [12], is desirable. On the more practical side, we are implementing proof transformations in our Datalog engine Nemo [35,34] and plan to assess the practical overhead of proof tree reconstruction on realistic knowledge graphs and optimised rule sets. Lastly, it would be interesting to study how proof tree transformations generalise to extensions of Datalog such as existential rules, aggregates, or stratified negation.

Acknowledgments. This work is supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in project number 389792660 (TRR 248, Center for Perspicuous Systems), by the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR, Federal Ministry of Research, Technology and Space) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), and in DAAD project 57616814 (SECAI, School of Embedded Composite AI) as part of the program Konrad Zuse Schools of Excellence in Artificial Intelligence.

Supplemental Material Statement. While the paper is self-contained, we provide further examples of proof transformations in the appendix.

Declaration of use of Generative AI. Claude (Anthropic) was used for proof-reading and preparing the figures. It also identified and helped correct an error in the proof of Theorem 2.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Alrabbaa, C., Baader, F., Borgwardt, S., Dachsel, R., Koopmann, P., Méndez, J.: Evonne: Interactive proof visualization for description logics (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Proc. 9th Int. Joint Conf. on Automated Reasoning (IJCAR'18). Lecture Notes in Computer Science, vol. 13385, pp. 271–280. Springer (2022). <https://doi.org/10.1007/978-3-031-10769-6>
3. Alrabbaa, C., Baader, F., Borgwardt, S., Koopmann, P., Kovtunova, A.: Finding small proofs for description logic entailments: Theory and practice. In: Albert, E., Kovács, L. (eds.) Proc. 13th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'20). EPIc Series in Computing, vol. 73, pp. 32–67. EasyChair (2020). <https://doi.org/10.29007/NHPP>
4. Alrabbaa, C., Borgwardt, S., Herrmann, P., Krötzsch, M.: The shape of $\mathcal{EL}$ proofs: A tale of three calculi. In: Proc. 38th Int. Workshop on Description Logics (DL'25). CEUR Workshop Proceedings, vol. 4091. CEUR-WS.org (2025), <https://ceur-ws.org/Vol-4091/paper45.pdf>
5. Alrabbaa, C., Borgwardt, S., Koopmann, P., Kovtunova, A.: Explaining ontology-mediated query answers using proofs over universal models. In: Governatori and Turhan [29], pp. 167–182. https://doi.org/10.1007/978-3-031-21541-4_11
6. Autebert, J., Berstel, J., Boasson, L.: Context-free languages and pushdown automata. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, Volume 1: Word, Language, Grammar, pp. 111–174. Springer (1997). https://doi.org/10.1007/978-3-642-59136-5_3
7. Baker, B.S.: Tree transducers and tree languages. Inf. Control. **37**(3), 241–266 (1978). [https://doi.org/10.1016/S0019-9958\(78\)90538-7](https://doi.org/10.1016/S0019-9958(78)90538-7)
8. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Silberschatz, A. (ed.) Proc. 5th Symposium on Principles of Database Systems (PODS'86). pp. 1–15. ACM (1986). <https://doi.org/10.1145/6012.15399>
9. Beaudry, M., McKenzie, P.: Circuits, matrices, and nonassociative computation. J. of Comput. Syst. Sci. **50**(3), 441–455 (1995). <https://doi.org/10.1006/JCSS.1995.1035>
10. Bellomarini, L., Sallinger, E., Gottlob, G.: The Vadalog system: Datalog-based reasoning for knowledge graphs. Proc. VLDB Endowment **11**(9), 975–987 (2018). <https://doi.org/10.14778/3213880.3213888>
11. Bojanczyk, M.: Transducers of polynomial growth. In: Baier, C., Fisman, D. (eds.) Proc. 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'22). pp. 1:1–1:27. ACM (2022). <https://doi.org/10.1145/3531130.3533326>
12. Bojanczyk, M., Doumane, A.: First-order tree-to-tree functions. In: Hermanns, H., Zhang, L., Kobayashi, N., Miller, D. (eds.) Proc. 35th Annual ACM/IEEE

- Symposium on Logic in Computer Science (LICS'20). pp. 252–265. ACM (2020). <https://doi.org/10.1145/3373718.3394785>
13. Borgwardt, S., Hirsch, A., Kovtunova, A., Wiehr, F.: In the eye of the beholder: Which proofs are best? In: Borgwardt, S., Meyer, T. (eds.) *Proc. 33rd Int. Workshop on Description Logics (DL'20)*. CEUR Workshop Proceedings, vol. 2663. CEUR-WS.org (2020), <https://ceur-ws.org/Vol-2663/paper-6.pdf>
 14. Calautti, M., Livshits, E., Pieris, A., Schneider, M.: The complexity of why-provenance for datalog queries. *Proc. ACM Manag. Data* **2**(2), 83 (2024). <https://doi.org/10.1145/3651146>
 15. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. *J. Web Semant.* **14**, 57–83 (2012). <https://doi.org/10.1016/j.websem.2012.03.001>
 16. Carral, D., Dragoste, I., Krötzsch, M.: Tractable query answering for expressive ontologies and existential rules. In: d'Amato, C., Fernández, M., Tamma, V., Lécué, F., Cudré-Mauroux, P., Sequeda, J.F., Lange, C., Heflin, J. (eds.) *Proc. 16th Int. Semantic Web Conf. (ISWC'17), Part I. LNCS*, vol. 10587, pp. 156–172. Springer (2017). https://doi.org/10.1007/978-3-319-68288-4_10
 17. Carral, D., Dragoste, I., Krötzsch, M.: The combined approach to query answering in horn-alchoiq. In: Thielscher, M., Toni, F., Wolter, F. (eds.) *Proc. 16th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'18)*. pp. 339–348. AAAI Press (2018), <https://aaai.org/proceeding/kr-16-2018/>
 18. Carral, D., Dragoste, I., Krötzsch, M.: Reasoner = logical calculus + rule engine. *Künstliche Intell.* **34**(4), 453–463 (2020). <https://doi.org/10.1007/S13218-020-00667-6>
 19. Chaudhuri, S., Vardi, M.Y.: On the equivalence of recursive and nonrecursive datalog programs. In: Vardi, M.Y., Kanellakis, P.C. (eds.) *Proc. 11th Symposium on Principles of Database Systems (PODS'92)*. pp. 55–66. ACM Press (1992). <https://doi.org/10.1145/137097.137109>
 20. Courcelle, B.: Monadic second-order definable graph transductions: A survey. *Theor. Comput. Sci.* **126**(1), 53–75 (1994). [https://doi.org/10.1016/0304-3975\(94\)90268-2](https://doi.org/10.1016/0304-3975(94)90268-2)
 21. Dachsel, R., Gerlach, L., Hanisch, P., Ivliev, A., Krötzsch, M., Marx, M., Méndez, J.: Declarative debugging for datalog with aggregation. In: Krause, A., Pimentel, J.F. (eds.) *Proc. Workshops of the EDBT/ICDT 2026 Joint Conf. CEUR WS Proceedings*. CEUR-WS (2026), <https://ceur-ws.org/Vol-4192/TGD-paper4.pdf>
 22. Elberfeld, M., Jakoby, A., Tantau, T.: Logspace versions of the theorems of bodlaender and courcelle. In: *Proc. 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS'10)*. pp. 143–152. IEEE Computer Society (2010). <https://doi.org/10.1109/FOCS.2010.21>
 23. Elhalawati, A., Krötzsch, M., Mennicke, S.: An existential rule framework for computing why-provenance on-demand for datalog. In: Governatori and Turhan [29], pp. 146–163
 24. Engelfriet, J., Maneth, S.: Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inf. Comput.* **154**(1), 34–91 (1999). <https://doi.org/10.1006/INCO.1999.2807>
 25. Furche, T., Gottlob, G., Grasso, G., Guo, X., Orsi, G., Schallhart, C., Wang, C.: DIADEM: Thousands of websites to a single database. *Proc. VLDB Endow.* **7**(14), 1845–1856 (2014). <https://doi.org/10.14778/2733085.2733091>
 26. Gallot, P., Lhote, N., Nguyễn, L.T.D.: The structure of polynomial growth for tree automata/transducers and MSO set queries. *CoRR* **abs/2501.10270** (2025). <https://doi.org/10.48550/ARXIV.2501.10270>

27. Gécseg, F., Steinby, M.: *Tree Automata*. Akadémiai Kiadó, Budapest, Hungary (1984)
28. González, L., Ivliev, A., Krötzsch, M., Mennicke, S.: Efficient dependency analysis for rule-based ontologies. In: Sattler, U., Hogan, A., Keet, M., Presutti, V., Almeida, J.P.A., Takeda, H., Monnin, P., Pirrò, G., d’Amato, C. (eds.) *Proc. 21st International Semantic Web Conference (ISWC 2022)*. LNCS, vol. 13489, pp. 267–283. Springer (October 2022). https://doi.org/10.1007/978-3-031-19433-7_16
29. Governatori, G., Turhan, A. (eds.): *Proc. 2nd Int. Joint Conf. on Rules and Reasoning (RuleML+RR’22)*, LNCS, vol. 13752. Springer (2022)
30. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Libkin, L. (ed.) *Proc. 26th Symp. on Principles of Database Systems (PODS’07)*. pp. 31–40. ACM (2007). <https://doi.org/10.1145/1265530.1265535>
31. Greenlaw, R., Hoover, H.J., Ruzzo, W.L.: *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press (1995)
32. Hanisch, P., Krötzsch, M.: Rule rewriting revisited: A fresh look at static filtering for Datalog and ASP. In: ten Cate, B., Funk, M. (eds.) *Proc. 29th Int. Conf. on Database Theory (ICDT’2026)*. LIPIcs, vol. 365. Dagstuhl Publishing (2026). <https://doi.org/10.4230/LIPIcs.ICDT.2026.5>
33. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in OWL. In: Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) *Proc. 7th Int. Semantic Web Conf. (ISWC’08)*. LNCS, vol. 5318, pp. 323–338. Springer (2008)
34. Ivliev, A., Ellmauthaler, S., Gerlach, L., Marx, M., Meißner, M., Meusel, S., Krötzsch, M.: Nemo: First glimpse of a new rule engine. In: Pontelli, E., Costantini, S., Dodaro, C., Gaggli, S.A., Calegari, R., d’Avila Garcez, A.S., Fabiano, F., Mileo, A., Russo, A., Toni, F. (eds.) *Proc. 39th Int. Conf. on Logic Programming (ICLP’23)*. EPTCS, vol. 385, pp. 333–335 (2023). <https://doi.org/10.4204/EPTCS.385.35>
35. Ivliev, A., Gerlach, L., Meusel, S., Steinberg, J., Krötzsch, M.: Nemo: Your friendly and versatile rule reasoning toolkit. In: Marquis, P., Ortiz, M., Pagnucco, M. (eds.) *Proc. 21st Int. Conf. on Principles of Knowledge Representation and Reasoning (KR’24)*. pp. 743–754. IJCAI Organization (2024). <https://doi.org/10.24963/kr.2024/70>
36. Ivliev, A., Krötzsch, M., Marx, M.: Recovering explanations from transformed rule-based ontologies. In: Koubarakis, M., et al. (eds.) *Proc. 25th Int. Semantic Web Conf. (ISWC’26)*. LNCS, Springer (2026), to appear
37. Jordan, H., Scholz, B., Subotic, P.: Soufflé: On synthesis of program analyzers. In: Chaudhuri, S., Farzan, A. (eds.) *Proc. 28th Int. Conf. on Computer Aided Verification (CAV’16)*, Part II. LNCS, vol. 9780, pp. 422–430. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_23
38. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. In: Aberer, K., Choi, K.S., Noy, N., Allemang, D., Lee, K.I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) *Proc. 6th Int. Semantic Web Conf. (ISWC’07)*. LNCS, vol. 4825, pp. 267–280. Springer (2007). https://doi.org/10.1007/978-3-540-76298-0_20
39. Krötzsch, M.: Efficient inferencing for OWL EL. In: Janhunen, T., Niemelä, I. (eds.) *Proc. 12th European Conf. on Logics in Artificial Intelligence (JELIA’10)*. LNAI, vol. 6341, pp. 234–246. Springer (2010)
40. Krötzsch, M.: Efficient rule-based inferencing for OWL EL. In: Walsh, T. (ed.) *Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI’11)*. pp. 2668–2673. AAAI Press/IJCAI (2011). <https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-444>

41. Krötzsch, M.: Modern Datalog: Concepts, methods, applications. In: Artale, A., Bienvenu, M., García, Y.I., Murlak, F. (eds.) Joint Proceedings of the 20th and 21st Reasoning Web Summer Schools (RW 2024 & RW 2025). OASICs, vol. 138. Dagstuhl Publishing (2025). <https://doi.org/10.4230/OASICs.RW.2024/2025.7>
42. Krötzsch, M., Thost, V.: Ontologies for knowledge graphs: Breaking the rules. In: Groth, P.T., Simperl, E., Gray, A.J.G., Sabou, M., Krötzsch, M., Lécué, F., Flöck, F., Gil, Y. (eds.) Proc. 15th Int. Semantic Web Conf. (ISWC'16, Part I). LNCS, vol. 9981, pp. 376–392 (2016)
43. Lautemann, C., McKenzie, P., Schwentick, T., Vollmer, H.: The descriptive complexity approach to LOGCFL. *J. of Comput. Syst. Sci.* **62**(4), 629–652 (2001). <https://doi.org/10.1006/JCSS.2000.1742>
44. Libkin, L.: Elements of Finite Model Theory. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). <https://doi.org/10.1007/978-3-662-07003-1>
45. Logica Devs: Logica project webpage (Retrieved 04 Sept 2025), <https://logica-web.github.io/>
46. Megiddo, N., Papadimitriou, C.H.: On total functions, existence theorems and computational complexity. *Theor. Comput. Sci.* **81**(2), 317–324 (1991). [https://doi.org/10.1016/0304-3975\(91\)90200-L](https://doi.org/10.1016/0304-3975(91)90200-L)
47. Méndez, J., Alrabbaa, C., Koopmann, P., Langner, R., Baader, F., Dachselt, R.: Evonne: A visual tool for explaining reasoning with OWL ontologies and supporting interactive debugging. *Comput. Graph. Forum* **42**(6) (2023). <https://doi.org/10.1111/CGF.14730>
48. Méndez, J., Gerlach, L., Wieland, T., Ivliev, A., Krötzsch, M., Dachselt, R.: Evaluating a visual query tracer and builder for learning declarative logic programming. In: Proc. 37th IEEE Conf. on Visualization and Visual Analytics (VIS'26). IEEE (2026), to appear
49. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (2009), available at <http://www.w3.org/TR/owl2-profiles/>
50. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFS: A highly-scalable RDF store. In: et al., M.A. (ed.) Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II. LNCS, vol. 9367, pp. 3–20. Springer (2015). https://doi.org/10.1007/978-3-319-25010-6_1
51. Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. *J. Log. Program.* **19/20**, 261–320 (1994). [https://doi.org/10.1016/0743-1066\(94\)90028-0](https://doi.org/10.1016/0743-1066(94)90028-0)
52. Piro, R., Nenov, Y., Motik, B., Horrocks, I., Hendler, P., Kimberly, S., Rossman, M.: Semantic technologies for data analysis in health care. In: Groth, P.T., Simperl, E., Gray, A.J.G., Sabou, M., Krötzsch, M., Lécué, F., Flöck, F., Gil, Y. (eds.) Proc. 15th Int. Semantic Web Conf. (ISWC'16, Part II). LNCS, vol. 9982, pp. 400–417 (2016). https://doi.org/10.1007/978-3-319-46547-0_34
53. Ramakrishnan, R., Beer, C., Krishnamurthy, R.: Optimizing existential datalog queries. In: Edmondson-Yurkanan, C., Yannakakis, M. (eds.) Proc. 7th Symposium on Principles of Database Systems (PODS'88). pp. 89–102. ACM (1988). <https://doi.org/10.1145/308386.308420>
54. Ramakrishnan, R., Sagiv, Y., Ullman, J.D., Vardi, M.Y.: Logical query optimization by proff-tree transformation. *J. of Comput. Syst. Sci.* **47**(1), 222–248 (1993). [https://doi.org/10.1016/0022-0000\(93\)90025-R](https://doi.org/10.1016/0022-0000(93)90025-R)

55. Sagiv, Y.: Optimizing datalog programs. In: Minker, J. (ed.) *Foundations of Deductive Databases and Logic Programming*, pp. 659–698. Morgan Kaufmann (1988). <https://doi.org/10.1016/B978-0-934613-40-8.50021-X>
56. Shmueli, O.: Decidability and expressiveness aspects of logic queries. In: Vardi, M.Y. (ed.) *Proc. 6th Symposium on Principles of Database Systems (PODS’87)*. pp. 237–249. ACM (1987). <https://doi.org/10.1145/28659.28685>
57. Skvortsov, E.S., Xia, Y., Bowers, S., Ludäscher, B.: Logica-TGD: Transforming graph databases logically. In: Boehm, M., Daudjee, K. (eds.) *Proc. Workshops of the EDBT/ICDT 2025 Joint Conf. CEUR WS Proceedings*, vol. 3946. CEUR-WS (2025), <https://ceur-ws.org/Vol-3946/TGD-4.pdf>
58. Urbani, J., Jacobs, C., Krötzsch, M.: Column-oriented Datalog materialization for large knowledge graphs. In: Schuurmans, D., Wellman, M.P. (eds.) *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI’16)*. pp. 258–264. AAAI Press (2016). <https://doi.org/10.1609/aaai.v30i1.9993>
59. Urbani, J., Piro, R., van Harmelen, F., Bal, H.E.: Hybrid reasoning on OWL RL. *Semantic Web* **5**(6), 423–447 (2014). <https://doi.org/10.3233/SW-130120>
60. Zhao, D., Subotic, P., Scholz, B.: Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Trans. Program. Lang. Syst.* **42**(2), 7:1–7:35 (2020). <https://doi.org/10.1145/3379446>

A Examples of Proof Tree Homomorphisms

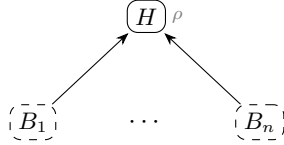
In this section, we give further examples of program transformations that can be reversed using proof tree homomorphisms. Let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be a program. We consider several common classes of program transformations that produce a program $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ with $\Pi_2 \subseteq \Pi_1$, and provide proof tree homomorphisms from Π_2 to Π_1 .

A.1 Atom Permutation

A simple program transformation swaps atoms within rule bodies [51]. This has no effect on the derived facts but may affect evaluation order in a reasoner. Formally, an *atom permutation* is a family $\{\pi_\rho\}_{\rho \in R_1}$, where $\pi_\rho: [n] \rightarrow [n]$ is a permutation with $|\rho| = n$ for each $\rho \in R_1$, and $[n] := \{1, \dots, n\}$. For a rule $\rho = H \leftarrow B_1, \dots, B_n$, we define

$$\pi_\rho(\rho) := H \leftarrow B_{\pi_\rho^{-1}(1)}, \dots, B_{\pi_\rho^{-1}(n)}.$$

The transformed program is Π_2 with $R_2 = \{\pi_\rho(\rho) \mid \rho \in R_1\}$. To reverse this transformation, we define, for each $\pi_\rho(\rho) \in R_2$, the proof tree template $S_{\pi_\rho(\rho)}$ for the rule $\pi_\rho(\rho)$ over Π_1 :



A.2 Rule Unfolding

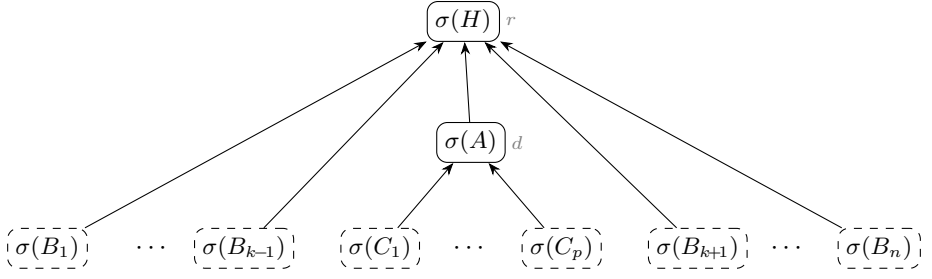
Rule unfolding replaces a body atom with the body of a rule that derives it, eliminating one level of indirection in the derivation [51].

For a rule ρ and a substitution σ , we write $\sigma(\rho)$ for the rule obtained by applying σ to each atom. Let $r = H \leftarrow B_1, \dots, B_n \in R_1$ be a rule and B_k be a body atom of r for some $1 \leq k \leq n$. Let $d = A \leftarrow C_1, \dots, C_p \in R_1$ be another rule (with $d \neq r$) such that the head of d unifies with B_k via a most general unifier σ . The *unfolding* of r at position k using d is the rule

$$\text{unfold}_{r,k}(d) := \sigma(H \leftarrow B_1, \dots, B_{k-1}, C_1, \dots, C_p, B_{k+1}, \dots, B_n) \quad (66)$$

A transformation from Π_1 to Π_2 is a *rule unfolding* if for every $\rho \in R_2$, either $\rho \in R_1$ holds, or there are $r, d \in R_1$ and $1 \leq k \leq |r|$ such that $\rho = \text{unfold}_{r,k}(d)$.

To reverse unfolding, consider an unfolded rule $\rho' = \text{unfold}_{r,k}(d)$. The template $S_{\rho'}$ reconstructs the original two-step derivation:



For rules $\rho \in R_2 \cap R_1$ that were not unfolded, the template S_ρ is the identity: a root node deriving the head of ρ connected to the leaf nodes that are labelled by the body atoms of ρ .

B Examples of MSO Proof Transformations

In this section, we present examples of program transformations that can be reversed using MSO proof transformations. We let $\Pi_1 = \langle R_1, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ be a program and define program transformations obtaining $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$, which can be reversed by proof transformations from Π_2 to Π_1 .

B.1 Rule Folding

Rule folding replaces multiple rules whose bodies share a common pattern by a single rule that uses an auxiliary predicate [51]. Let $c_1, \dots, c_n$ and $d_1, \dots, d_n$ be pairwise distinct rules in R_1 , and let A and H be atoms and $\bar{B}$ a list of atoms such that:

- $\text{head}(d_i)$ is unifiable with A using most general unifier σ_i ,
- $c_i = \sigma_i(H \leftarrow \bar{B}, \text{body}(d_i))$, and
- the head of every rule $r \notin \{d_1, \dots, d_n\}$ is not unifiable with A .

The *folding* of $c_1, \dots, c_n$ using $d_1, \dots, d_n$ is the rule $r := H \leftarrow A, \bar{B}$. The transformed program Π_2 is obtained from Π_1 by replacing $\{c_1, \dots, c_n\}$ with $\{r\}$.

We can reverse this transformation using a 1-dimensional MSO proof transformation with a single component. The idea is to replace the rule r with rule c_i if the A was derived by rule d_i .

$$\varphi_{\text{dom}}(x) := \bigwedge_{i=1}^n \neg \text{rule}_{d_i}(x) \quad (67)$$

$$\varphi_{c_i}(x) := \text{rule}_r(x) \wedge \exists w. \text{child}_1(w, x) \wedge \text{rule}_{d_i}(w) \quad (68)$$

$$\varphi_{\rho}(x) := \text{rule}_{\rho}(x) \quad \text{for } \rho \notin \{c_1, \dots, c_n\} \quad (69)$$

The child formula rewires the children of an r -node, placing the $\bar{B}$ -children first and then the children inherited from the absorbed d_i -node:

$$\begin{aligned} \varphi_{\text{child}}^j(x, y) &:= (\neg \text{rule}_r(y) \wedge \text{child}_j(x, y)) \\ &\vee (\text{rule}_r(y) \wedge \text{child}_{j+1}(x, y)) \quad \text{if } 1 \leq j \leq |\bar{B}| \end{aligned} \quad (70)$$

$$\begin{aligned} \varphi_{\text{child}}^j(x, y) &:= (\neg \text{rule}_r(y) \wedge \text{child}_j(x, y)) \\ &\vee (\text{rule}_r(y) \wedge \exists w. \text{child}_1(w, y) \wedge \text{child}_{j-|\bar{B}|}(x, w)) \quad \text{if } j > |\bar{B}| \end{aligned} \quad (71)$$

$$\varphi_{\text{arg}}^j(x, d) := \text{arg}_j(x, d) \quad (72)$$

B.2 Filter Propagation

Filter propagation is a technique that moves filter conditions such as $x = 3$ through the rules of a program, so that they are checked as early as possible during bottom-up evaluation [32].

Formally, a *filter propagation* is a mapping f that assigns to each body atom of each rule in R_1 a new rule and body position in which it should appear.

That is, f maps atom positions $\langle \rho, i \rangle$ with $\rho \in R_1$ and $1 \leq i \leq |\rho|$ to tuples $\langle \rho', j, \sigma \rangle$, where $\rho' \in R_1$, $j \geq 1$ and σ is a substitution adapting the variables. Atom positions with $f(\rho, i) = \langle \rho, i, id \rangle$ are *stationary*; all others are *moved*. The transformed program $\Pi_2 = \langle R_2, \mathbf{P}_{in}, \mathbf{P}_{out} \rangle$ has, for each $\rho' \in R_1$, a rule $f(\rho')$ whose head is $\text{head}(\rho')$ and whose j -th body atom is $\sigma(\text{body}_i(\rho))$ for the unique $\langle \rho, i \rangle$ with $f(\rho, i) = \langle \rho', j, \sigma \rangle$. We set $R_2 := \{f(\rho) \mid \rho \in R_1\}$.

Example 6. Consider the program Π_1 :

$$\rho_1 : r(x, y) \leftarrow e(x, y) \quad (73)$$

$$\rho_2 : r(x, y) \leftarrow r(x, z), e(z, y) \quad (74)$$

$$\rho_3 : \text{out}(y) \leftarrow r(x, y), x = a \quad (75)$$

The filter $x = a$ at position $(\rho_3, 2)$ is propagated to the base case, yielding:

$$f(\rho_1) : r(x, y) \leftarrow e(x, y), x = a$$

$$f(\rho_2) : r(x, y) \leftarrow r(x, z), e(z, y)$$

$$f(\rho_3) : \text{out}(y) \leftarrow r(x, y)$$

Here $f(\rho_3, 2) = \langle \rho_1, 2, id \rangle$ and all other positions are stationary. In a proof tree for Π_2 , the filter appears at every base-case leaf; in a proof tree for Π_1 , it appears only at the root.

An MSO proof transformation from Π_2 to Π_1 can be defined using a 1-dimensional MSO interpretation with a single component.

$$\varphi_{\text{dom}}(x) := \top \quad (76)$$

$$\varphi_\rho(x) := \text{rule}_{f(\rho)}(x) \quad (77)$$

$$\varphi_{\text{child}}^i(x, y) := \bigvee_{f(\langle \rho, i \rangle) = \langle \rho', j, \sigma \rangle} \text{rule}_{f(\rho)}(y) \wedge \exists p. \text{child}_j(x, p) \quad (78)$$

$$\wedge \text{rule}_{f(\rho')}(p) \wedge p \leq y$$

$$\varphi_{\text{arg}}^j(x, d) := \text{arg}_j(x, d) \quad (79)$$

The domain formula retains all nodes and the rule formula recovers the original rule name. The argument formula is inherited directly, since filter propagation does not change rule heads. The child formula links a filter that has been pushed down in the tree to its original ancestor.

B.3 Projection Pushing

Projection pushing removes argument positions in predicates such that the output of the program is unaffected [53].

Formally, let $\Pi_1 = \langle R_1, \mathbf{P}_{in}, \mathbf{P}_{out} \rangle$ be a program. A *position elimination* assigns to each IDB predicate p a set of *retained positions* $U(p) \subseteq \{1, \dots, \text{ar}(p)\}$ such that all positions of output and input predicates are retained. For an

atom $p(t_1, \dots, t_m)$, we write $p|_U(t_1, \dots, t_m)$ for the atom $p'(t_{j_1}, \dots, t_{j_k})$ where $U(p) = \{j_1 < \dots < j_k\}$ and p' is a fresh predicate of arity k . For a rule $\rho = H \leftarrow B_1, \dots, B_n$, the *reduced rule* is

$$\rho|_U := H|_U \leftarrow B_1|_U, \dots, B_n|_U \quad (80)$$

provided $\rho|_U$ is safe. The transformed program is $\Pi_2 = \langle R_2, \mathbf{P}_{\text{in}}, \mathbf{P}_{\text{out}} \rangle$ with rule set $R_2 = \{\rho|_U \mid \rho \in R_1\}$.

Example 7. Consider the program Π_1 computing the transitive closure of e :

$$\begin{aligned} r(x, y) &\leftarrow e(x, y) \\ r(x, y) &\leftarrow r(x, z), e(z, y) \\ \text{out}(y) &\leftarrow r(x, y) \end{aligned}$$

If we are only interested in the output predicate out , the first position of r can be eliminated. Setting $U(r) = \{2\}$ yields Π_2 :

$$\begin{aligned} r'(y) &\leftarrow e(x, y) \\ r'(y) &\leftarrow r'(z), e(z, y) \\ \text{out}(y) &\leftarrow r'(y) \end{aligned}$$

This transformation can be reversed using a 1-dimensional MSO interpretation with a single component. The tree structure is preserved and the rule labelling recovers the original rule from the reduced one for each $\rho \in R_1$:

$$\varphi_{\text{dom}}(x) := \top \quad (81)$$

$$\varphi_{\text{child}}^i(x, y) := \text{child}_i(x, y) \quad (82)$$

$$\varphi_\rho(x) := \text{rule}_{\rho|_U}(x) \quad (83)$$

Defining the argument formula is a bit more involved. For retained positions, the argument value is read directly from the reduced representation. For eliminated positions, we trace the value downward through the tree. Fix a rule $\rho \in R_1$ with head predicate p and let $j \notin U(p)$ be an eliminated position. Let x_j denote the variable at position j in $\text{head}(\rho)$. By safety, x_j occurs in at least one body atom of ρ . For each body atom $\text{body}_i(\rho)$ with predicate q in which x_j occurs at position j' , exactly one of the following holds:

1. q is an input predicate, or $j' \in U(q)$: the value can be read at the i -th child of x .
2. q is an IDB predicate and $j' \notin U(q)$: position j' is itself eliminated, and we must recurse at the i -th child.

To encode this recursive computation, we define a fixed finite graph that captures how variable bindings propagate through rules. The *predicate position graph* of Π_1 is the labelled directed graph G_Π whose vertices are predicate-position pairs $\langle p, j \rangle$ with p a predicate and $1 \leq j \leq \text{ar}(p)$. It contains an edge

$\langle p, j \rangle \rightarrow \langle q, j' \rangle$ labelled $\langle \rho, i \rangle$ whenever $\rho \in R_1$ has head predicate p , the variable at position j in $\text{head}(\rho)$ also appears at position j' in $\text{body}_i(\rho)$, and that body atom has predicate q . A vertex $\langle q, j' \rangle$ is *terminal* if $q \in \mathbf{P}_{\text{in}}$ or $j' \in U(q)$.

The MSO formula $\Phi_{p,j}(x, d)$ quantifies over a downward path in the proof tree together with a run of G_Π along that path. For each vertex $\langle q, j' \rangle$ of G_Π , we introduce a monadic variable $X_{q,j'}$ representing the set of tree nodes on the path that are visited in graph node $\langle q, j' \rangle$. Let $V(G_\Pi)$ be the set of vertices in G_Π and let $\bar{X}_{q,j'}$ be a list of monadic second-order variables $X_{q,j'}$ for each vertex $\langle q, j' \rangle$ in G_Π .

$$\begin{aligned} \Phi_{p,j}(x, d) := & \exists \bar{X}_{q,j'}. \exists y. \left(x \in X_{p,j} \wedge \forall v. \left(\right. \right. \\ & \bigwedge_{\langle q,j' \rangle \in V(G_\Pi)} \left(v \in X_{q,j'} \Rightarrow \right. \\ & \bigvee_{\substack{\langle q,j' \rangle \rightarrow \langle q',j'' \rangle \\ \text{labelled } \langle r,i \rangle}} \left(\text{rule}_{r|U}(v) \wedge \exists w. \text{child}_i(w, v) \right. \\ & \left. \left. \left. \wedge \begin{cases} w = y \wedge \text{arg}_{j''}(y, d) & \text{if } \langle q', j'' \rangle \text{ terminal} \\ w \in X_{q',j''} & \text{otherwise} \end{cases} \right) \right) \right) \end{aligned} \quad (84)$$

At the start, node x is in state $\langle p, j \rangle$. At each step, a node v in state $\langle q, j' \rangle$ picks an edge $\langle q, j' \rangle \rightarrow \langle q', j'' \rangle$ labelled $\langle \rho, i \rangle$, verifies that v uses rule $\rho|_U$, and descends to child i . If $\langle q', j'' \rangle$ is terminal, the value d is read at position j'' of that child. Otherwise, the child enters state $\langle q', j'' \rangle$ and the process continues.

The argument formula distinguishes retained and eliminated positions:

$$\varphi_{\text{arg}}^j(x, d) := \bigvee_{\substack{r \in R_1 \\ \text{head pred. } p}} \text{rule}_{r|U}(x) \wedge \begin{cases} \text{arg}_{j'}(x, d) & \text{if } j \in U(p) \\ \Phi_{p,j}(x, d) & \text{if } j \notin U(p) \end{cases} \quad (85)$$

where $j' = |\{k \in U(p) \mid k \leq j\}|$ is the position of j in the reduced predicate. For retained positions, the value is read directly from the reduced representation (adjusting for the change in indexing). For eliminated positions, the value is recovered via $\Phi_{p,j}$.