Breaking the Negative Cycle: Exploring the Design Space of Stratification for First-Class Datalog Constraints

Jonathan Lindegaard Starup
Department of Computer Science, Aarhus University, Denmark

Magnus Madsen
Department of Computer Science, Aarhus University, Denmark

Ondřej Lhoták
David R. Cheriton School of Computer Science, University of Waterloo, Canada

Abstract

The λ_{DAT} calculus brings together the power of functional and declarative logic programming in one language. In λ_{DAT}, Datalog constraints are first-class values that can be constructed, passed around as arguments, returned, composed with other constraints, and solved.

A significant part of the expressive power of Datalog comes from the use of negation. Stratified negation is a particularly simple and practical form of negation accessible to ordinary programmers. Stratification requires that Datalog programs must not use recursion through negation.

For a Datalog program, this requirement is straightforward to check, but for a λ_{DAT} program, it is not so simple: A λ_{DAT} program constructs, composes, and solves Datalog programs at runtime. Hence stratification cannot readily be determined at compile-time.

In this paper, we explore the design space of stratification for λ_{DAT}. We investigate strategies to ensure, at compile-time, that programs constructed at runtime are guaranteed to be stratified, and we argue that previous design choices in the Flix programming language have been suboptimal.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Datalog, first-class Datalog constraints, negation, stratified negation, type system, row polymorphism, the Flix programming language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.31

1 Introduction

Datalog is an expressive and powerful declarative logic programming language. A Datalog program consists of facts and rules. Facts represent knowledge (e.g. "an owl is a bird") whereas rules allow one to infer new facts from existing facts (e.g. "if x is a bird and x is not a penguin then x can fly."). The facts and rules imply a minimal model, a unique solution to every Datalog program [18] (e.g. "an owl is a bird" and "an owl can fly").

Datalog has been used in a diverse set of applications including big-data analytics [20, 39, 41], social network analysis [39, 40], machine learning [32, 34], bio-informatics [25, 38], disassembly [15], micro-controller programming [46], networking and distributed systems [1, 10, 29], program analysis [7, 42, 43], and smart contract security [44].

Over the years, a plethora of Datalog extensions have been developed, adding support for object types [4], logic formulas [6], decidable arithmetic functions [23], disjunctive rule heads [13], distributed evaluation [24], and more. Many Datalog solvers have also been developed, including DLV [2], Soufflé [22], LogicBlox [3], QL [4], Formulog [6], and Flix [31].

A significant part of both the theoretical and practical expressive power of Datalog comes from the use of negation. However, negation also brings challenges: ensuring a meaningful semantics for logic programming with negation is a historically well-studied problem [14, 16, 17, 27, 36, 37]. Stratified negation has emerged as a simple and practical

semantics that is accessible to ordinary programmers [48]. Informally, a Datalog program is stratified when there is no recursion through negation, i.e. no predicate symbol can negatively depend on itself. Stratification splits a Datalog program D into a sequence of Datalog programs $D_1, \cdots, D_n$ where the "output facts" of D_i become the "input facts" of D_{i+1}. Computing *whether* a Datalog program is stratified, and if so, computing its strata, is straightforward, e.g. using Ullman's algorithm [45].

The λ_{DAT} calculus extends the lambda calculus with first-class Datalog constraints [30]. In λ_{DAT}, Datalog constraints, or programs, are values that can be constructed, passed as arguments to functions, returned from functions, composed with other Datalog program values, and have their minimal model computed. The minimal model is itself a set of facts, hence a Datalog value. This makes it possible to construct pipelines of Datalog computations. The type system of λ_{DAT} is based on Hindley-Milner [12, 47] extended with row polymorphism [28]. The type system permits Datalog constraints to be polymorphic, while ensuring type safety [30]. The λ_{DAT} calculus and its type system has been implemented in the Flix programming language.

For a λ_{DAT} program, we cannot readily determine whether the Datalog values constructed at runtime are stratified. We *can* defer stratification until runtime, but this has two significant downsides: (1) we must perform the stratification repeatedly, and worse, (2) we have to abort execution if a non-stratified Datalog program is ever constructed.

In this paper, we explore the design space of compile-time techniques, which ensure that λ_{DAT} programs never construct non-stratified Datalog values at runtime. We also show that these techniques enable stratification of Datalog programs with lattice semantics.
In summary, the paper makes the following contributions:

- (**Design Space**) We explore the design space of compile-time stratification in the presence of first-class Datalog constraints.
- (**Framework**) We formulate the design space in a mathematical framework that allows us to express each design point as a specific instantiation of the framework. We introduce the notion of a *labelled dependency graph* and discuss how it can be used to soundly over-approximate the dependency edges of a Datalog expression.
- (**Comparison**) We identify the current state-of-the-art, i.e. the technique currently used in the Flix programming language, in the design space and illustrate that some of its design choices have been sub-optimal.
- (**Implementation**) We extend the Flix programming language with different design choices that admit more programs (i.e. allow more safe programs to pass the type checker). In particular, our extension uses rule-level granularity (**choice 1c**), and uses predicate arity and predicate term types in the labelled dependency graph (**choice 2b** and **2c**).
- (**Case Study**) We conduct a case study of a graph library that we implement in Flix. The study shows that use of stratified negation and lattice semantics is prevalent.

This paper is structured as follows: We motivate our work in Section 2. In Section 3 we present background material on Datalog, on stratified negation, and on the λ_{DAT} calculus. We explore the design space of stratification for λ_{DAT} in Section 4, Section 5, and Section 6. In Section 7 we discuss the design choices that we have made and our implementation in the Flix programming language. We use this implementation for a graph library case study in Section 8. Section 9 presents related work and Section 10 concludes. To ensure that the paper is self-contained, the background section has to cover a lot of material. Readers who are already familiar with Datalog, stratified negation, or the λ_{DAT} calculus are encouraged to skip the background material.

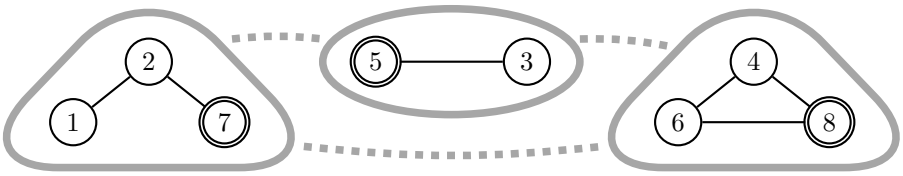


Figure 1 An undirected graph. We want to compute the connected components of the graph and introduce edges that connect them. The components are indicated by thick gray lines. The edges we want to compute are indicated by dashed lines. The double circled nodes are the representatives of each connected component.

2 Motivation

We motivate our work with an example. Consider the following problem:

> Given an undirected graph, compute its connected components and introduce an edge between each component.

Figure 1 illustrates the problem with an example. The graph has nodes numbered one to eight. The connected components are $\{1, 2, 7\}$, $\{3, 5\}$, and $\{4, 6, 8\}$. We want to compute the three (undirected) edges: $\{1, 2, 7\} \leftrightarrow \{3, 5\}$, $\{1, 2, 7\} \leftrightarrow \{4, 6, 8\}$, and $\{3, 5\} \leftrightarrow \{4, 6, 8\}$, as shown in Figure 1.

We can use the Flix programming language, with its support for first-class Datalog constraints, to elegantly solve this problem. Figure 2 shows a Flix program that does so. The program consists of two functions: connectedComponents and connectGraph.

The connectedComponents function computes the connected components (CCs) of the given (undirected) graph. The graph is represented as a set of nodes (of type Set[Int32]) and a set of edges (of type Set[(Int32, Int32)]). First, the function converts the nodes and edges into Node and Edge facts. Next, the function defines a local variable r which is a Datalog program value. The Datalog program defines Reachable as the (undirected) transitive closure of the Edge relation. Using Reachable, it computes the representative of each node in a CC as follows: Every node in a CC is associated with the lexicographically largest node in the same CC (ReachUp). The representative of a CC is then the node which is the largest in each CC, i.e. has no larger parent. Finally, the connectedComponents function composes the node (ns) and edge (es) facts with the Datalog program (r), computes its minimal model, and extracts all the ComponentRep facts. The polymorphic row extension in the return type $\#\{\cdots \mid r\}$ allows the caller to type the returned Datalog program with additional predicates.

The connectGraph function computes a set of edges that connect CCs in a graph. That is, the function returns a set of edges that connect *sets of nodes*. The connectGraph function takes a graph as input (using the same representation as before), and calls the connectedComponents function to compute the CCs, specifically the ComponentRep relation which holds the representative of each node in the graph. The function defines the local variable d which is a Datalog program value. The Datalog program uses the ComponentRep to build a map Component which maps each representative to the set of nodes it represents. The rule:

```
1 Component(rep; Set#{n}) :- ComponentRep(n, rep).
```

states that if there is a ComponentRep(n, rep) fact, for some n and rep, then we infer a Component(rep; Set#{n}) *lattice* fact where rep is mapped to the singleton set {n}. The Component lattice implicitly combines facts using the ordering on Set[Int32], which is subset

```
/// Given an undirected graph represented by nodes and edges,
/// computes its connected components and returns a relation
/// that maps each node to its representative.
def connectedComponents(nodes: Set[Int32], edges: Set[(Int32, Int32)]):
                        #{ ComponentRep(Int32, Int32) | r } =
  let ns = inject nodes into Node;
  let es = inject edges into Edge;
  let r = #{
    // Reachable(n1, n2) captures that n1 can reach n2.
    // All nodes can reach themselves.
    Reachable(n, n) :- Node(n).
    // n1 can reach n2 directly.
    Reachable(n1, n2) :- Edge(n1, n2).
    // n2 can reach n1 directly, since the graph is undirected.
    Reachable(n2, n1) :- Edge(n1, n2).
    // n1 can reach n3 by transitivity.
    Reachable(n1, n3) :- Reachable(n1, n2), Reachable(n2, n3).
    // ReachUp contains nodes that can reach at least one other node
    // with a higher value. That is, ReachUp contains all nodes which
    // are not a representative of their connected component.
    ReachUp(n1) :- Reachable(n1, n2), if n1 < n2.
    // The representative, rep, of a node, n, in a connected component
    // is any reachable node that is not contained in ReachUp.
    ComponentRep(n, rep) :- Reachable(n, rep), not ReachUp(rep).
  };
  solve ns, es, r project ComponentRep

/// Given an undirected graph represented by nodes and edges,
/// connects all connected components. The returned edges are
/// between components, i.e. they are edges between *sets* of nodes.
def connectGraph(nodes: Set[Int32], edges: Set[(Int32, Int32)]):
                    #{ Edge(Set[Int32], Set[Int32]) | r } =
  let d = #{
    // Component(rep; c) captures that the node rep is the
    // representative of the component c which is a set of nodes. The
    // semicolon makes c use lattice semantics which aggregates all
    // the nodes represented by rep into one set.
    Component(rep; Set#{n}) :- ComponentRep(n, rep).
    // Introduce an edge between every pair of components sets.
    // The fix keyword ensures that the Component relation is fully
    // materialized before this rule is evaluated, i.e. that a
    // component contains all its nodes.
    Edge(c1, c2) :- fix Component(_; c1), fix Component(_; c2).
  };
  solve connectedComponents(nodes, edges), d project Edge

def main(): Unit \ IO =
  let connectedGraph = connectGraph(Set.range(1, 9),
    Set#{ (1, 2), (2, 7), (5, 3), (8, 6), (6, 4), (4, 8)});
  let result = query connectedGraph select (c1, c2) from Edge(c1, c2);
  println(result)
```

Figure 2 Flix program that connects an undirected graph using Datalog.

inclusion. In other words, if there are multiple ComponentRep facts with the same rep then every set is union'ed together. The last rule introduces edges between the components:

```
Edge(c1, c2) :- fix Component(_; c1), fix Component(_; c2).
```

The fix keyword ensures that the Component lattice *is fully computed* before the rule is evaluated. We will explain the full details in Section 6.

2.1 Stratified Negation

The Flix program in Figure 2 uses negation in the following rule:

```
ComponentRep(n, rep) :- Reachable(n, rep), not ReachUp(rep).
```

where the ReachUp predicate symbol occurs negated.

Datalog has the theoretically interesting and practically useful property that every Datalog program has a unique solution; the minimal model. However, in the presence of negation, an additional side condition is necessary to ensure the existence of the minimal model.

To understand why, consider a Datalog program with the two rules:

$$\mathsf{WinBlack}(x) \Leftarrow \mathsf{not}\,\mathsf{WinWhite}(x). \quad \mathsf{WinWhite}(x) \Leftarrow \mathsf{not}\,\mathsf{WinBlack}(x).$$

The rules try to capture the idea that "if x is not a winning move for white then x is a winning move for black" and "if x is not a winning move for black then x is a winning move for white". The problem is that this Datalog program has *two* models, neither of which is minimal: $\{\mathsf{WinBlack}(p)\}$ and $\{\mathsf{WinWhite}(p)\}$ for some p. We want to avoid such situations.

Stratified negation overcomes this problem by imposing a simple restriction: A predicate symbol cannot negatively depend on itself. This requirement is sometimes expressed as "no recursion through negation". It is straightforward to determine if a Datalog program is stratified: We simply compute the dependency graph of the program and determine if it contains a cycle with a negative edge.

Returning to Figure 2, if we look at all the rules, we can see the following negative cycle:

$$\mathsf{Edge} \leftrightarrow \mathsf{Component} \leftarrow \mathsf{ComponentRep} \leftrightarrow \mathsf{ReachUp} \leftarrow \mathsf{Reachable} \leftarrow \mathsf{Edge}$$

But does the Flix program in Figure 2 actually construct a Datalog value with a negative cycle at runtime? Fortunately this is not the case. The reason is as follows: the Edge, Reachable, and ReachUp predicates are used to compute the ComponentRep relation. This Datalog program is fully solved before ComponentRep is used to compute a new set of Edges.

We can now describe the problem this paper aims to solve:

> In Flix, in the presence of first-class Datalog values, how can it be statically guaranteed that every Datalog value that may be constructed at runtime will be stratified?

In the example above, we used very ad-hoc reasoning to justify that a negative cycle cannot occur. While a very powerful and precise control– and data-flow analysis may be able to provide similar justification, in this paper we are interested in simpler and more light-weight techniques. We want to build on the type system of the λ_{DAT} calculus and of Flix. In the dependency Edge ← ComponentRep, the type of Edge is (Set[Int32], Set[Int32]), whereas in the dependency Reachable ← Edge, the type of Edge is (Int32, Int32). The type system ensures that predicate symbols with different types cannot occur in the same Datalog program value. This means that we can exclude the existence of edges based solely on the type information. Interestingly, we could also use the type system in a different way to exclude the negative cycle. The rule using negation mentions three predicate symbols: ComponentRep, Reachable, and ReachUp. The rule which closes the supposed cycle mentions two predicate symbols: Component and Edge. Since the program has no expression with a type that contains *all* these predicate symbols, we know that the cycle cannot occur.

We can now summarize the overall goal of this paper:

> We want to explore the design space of type-based techniques that can statically ensure that Flix programs, in the presence of first-class Datalog constraints, are stratified.

3 Background

We begin with an introduction to Datalog and stratified negation before we move on to describe the λ_{DAT} calculus [30]. This paper contains a lot of background material. Readers who are already familiar with Datalog are encouraged to jump to Section 3.3.

3.1 Datalog

We give a brief introduction to Datalog. A comprehensive introduction is available in [9, 18].

3.1.1 Syntax

A Datalog *program* D is a set of constraints $C_1, \cdots, C_n$. A *constraint* is of the form $A_0 \Leftarrow B_1, \cdots, B_n$ where A_0 is the *head atom* and each B_i is a *body atom*. A head atom $p(t_1, \cdots, t_n)$ consists of a predicate symbol p and a sequence of terms $t_1, \cdots, t_n$. A *body atom* is similar to a head atom, except that it can be negated, which is written with "not" in front of the predicate symbol. A constraint without a body is called a *fact*. A constraint with a body is called a *rule*. A *term* is either a variable x or a literal constant c. An atom without variables is said to be *ground*. A fact or rule with only ground atoms is said to be ground. Figure 3 shows the grammar of Datalog.

$$
\begin{array}{rcl@{\qquad}rcl}
D \in \textit{Programs} & = & C_1, \cdots, C_n & t \in \textit{Terms} & = & x \mid c \\
C \in \textit{Constraints} & = & A_0 \Leftarrow B_1, \cdots, B_n. & c \in \textit{Literals} & = & \text{a set of literal constants.} \\
A \in \textit{Head Atoms} & = & p(t_1, \cdots, t_n) & x, y \in \textit{VarSym} & = & \text{a set of variable symbols.} \\
B \in \textit{Body Atoms} & = & p(t_1, \cdots, t_n) & p, q \in \textit{PredSym} & = & \text{a set of predicate symbols.} \\
 & \mid & \mathsf{not}\, p(t_1, \cdots, t_n) & & &
\end{array}
$$

Figure 3 Syntax of Datalog.

A Datalog program must satisfy three syntactic properties that are not naturally captured by the grammar in Figure 3: (i) every fact must be ground, (ii) every variable that occurs in the head of a rule must also occur in its body, and (iii) every variable that occurs in a negated body atom must also occur in at least one positive body atom of the rule. If a program satisfies these properties it is said to be *well-formed*. In addition, every Datalog program which uses negation must be stratified, as we will explain in Section 3.2.

3.1.2 Semantics

The meaning of a Datalog program is usually defined in terms of the *minimal model*: the smallest interpretation (i.e. set of facts) that satisfy all the constraints (i.e. rule instantiations) of the program [18]. While the semantics of Datalog – and logic programs in general – is an interesting subject worthy of study, in this paper our focus is on stratification.

3.2 Stratified Negation

A significant part of the expressive power of Datalog comes from the use of negation, but unrestricted use of negation poses problems. Recall the Datalog program from Section 2:

$$\mathsf{WinBlack}(x) \Leftarrow \mathsf{not}\, \mathsf{WinWhite}(x). \quad \mathsf{WinWhite}(x) \Leftarrow \mathsf{not}\, \mathsf{WinBlack}(x).$$

If the program contains the constant 42, then the program has *two* solutions (models):

$$M_1 = \{\mathsf{WinBlack}(42)\} \quad \text{and} \quad M_2 = \{\mathsf{WinWhite}(42)\}$$

Neither model is a subset of the other. Hence neither model is minimal. This breaks one of the fundamental properties of Datalog: that every program has a unique solution. Defining a consistent semantics for logic programming languages with negation has long been studied and many proposals have been made [14, 16, 17, 27, 36, 37]. Stratified negation has emerged as a particularly simple semantics that can be mastered by ordinary programmers [48].

Informally, a Datalog program is said to be *stratified* if its predicate symbols can be partitioned into a sequence of *strata* such that a predicate symbol in a stratum only depends on predicate symbols in the same or lower strata. Intuitively, stratification splits a Datalog program D into a sequence of sub-programs $D_1, \cdots, D_n$ such that the output of D_i becomes the input of D_{i+1}.

We can determine if a Datalog program is stratified by computing its dependency graph:

▶ **Definition 1** (Dependency Graph). *The* dependency graph *(also called the precedence graph) of a Datalog program D is a directed graph of predicate symbols that contains:*

- *a* positive edge $a \leftarrow b$ *if D contains a rule where a is the predicate symbol of the head atom and b is a predicate symbol of a positive body atom, and*
- *a* negative edge $a \leftarrow\!\!\!\!\times\; b$ *if D contains a rule where a is the predicate symbol of the head and b is a predicate symbol of a negative body atom.*

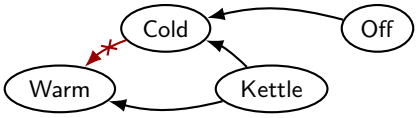
We write dependency edges as $a \leftarrow b$ and $a \leftarrow\!\!\!\!\times\; b$ since this matches the "direction" of Datalog rules. We say that a depends on b. If $a \leftarrow\!\!\!\!\times\; b$ we say that b must be computed before a. We write $\mathcal{DG}(D)$ for the dependency graph of the Datalog program D. Note that the dependency graph of a Datalog program D is unique.

We can now formally state when a Datalog program is stratified:

▶ **Definition 2** (Stratified). *A Datalog program D is* stratified *if its dependency graph contains no cycles with a negative edge.*

▶ **Example 3.** We will use the following running example. Consider the following Datalog program and its dependency graph:

$$\mathsf{Cold}(x) \Leftarrow \mathsf{Kettle}(x), \mathsf{Off}(x).$$
$$\mathsf{Warm}(x) \Leftarrow \mathsf{Kettle}(x), \mathsf{not}\ \mathsf{Cold}(x).$$



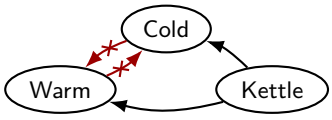
which does not contain a cycle with a negative edge. The strata of this program are:

$$s_1 = \{\mathsf{Cold}, \mathsf{Kettle}, \mathsf{Off}\} \quad s_2 = \{\mathsf{Warm}\}$$

which means that the Cold, Kettle, and Off relations must be computed before we compute the Warm relation.

▶ **Example 4.** Consider a modification of the previous Datalog program with its new dependency graph:

$$\mathsf{Cold}(x) \Leftarrow \mathsf{Kettle}(x), \mathsf{not}\ \mathsf{Warm}(x).$$
$$\mathsf{Warm}(x) \Leftarrow \mathsf{Kettle}(x), \mathsf{not}\ \mathsf{Cold}(x).$$



$$
\begin{array}{rcl@{\qquad}rcl}
v \in \mathit{Val} & = & c \mid \lambda x.\, e \mid \#\{C_1, \cdots, C_n\} & C \in \mathit{Constraints} & = & A_0 \Leftarrow B_1, \cdots, B_n. \\
e \in \mathit{Exp} & = & x \mid v \mid e\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, e & A \in \mathit{Head\ Atoms} & = & p(t_1, \cdots, t_n) \\
 & \mid & e\ \texttt{<+>}\ e \mid \mathsf{solve}\, e \mid \mathsf{project}\, p\, e & B \in \mathit{Body\ Atoms} & = & p(t_1, \cdots, t_n) \\
c \in \mathit{Literals} & = & \text{a set of literal constants.} & & \mid & \mathsf{not}\, p(t_1, \cdots, t_n) \\
x, y \in \mathit{VarSym} & = & \text{a set of variable symbols.} & & \mid & \mathsf{fix}\, p(t_1, \cdots, t_n) \\
p, q \in \mathit{PredSym} & = & \text{a set of predicate symbols.} & t \in \mathit{Terms} & = & x \mid c
\end{array}
$$

Figure 4 Syntax of λ_{DAT}.

$$
\begin{array}{rcl@{\qquad}rcl}
\tau \in \mathit{Type} & = & \alpha \mid \iota \mid \tau_1 \to \tau_2 \mid r & \sigma \in \mathit{Scheme} & = & \forall \overline{\alpha}\, \forall \overline{\rho}.\, \tau \\
r, s \in \mathit{Row} & = & \rho \mid \{\,\} \mid \{\, p = (\tau_1, \cdots, \tau_n) \mid r \,\} & \alpha \in \mathit{TypeVar} & = & \text{a set of type variables.} \\
\iota & = & \text{a set of base types.} & \rho \in \mathit{RowVar} & = & \text{a set of row variables.}
\end{array}
$$

Figure 5 Type System of λ_{DAT}.

The graph contains a negative cycle between the Cold and Warm predicate symbols hence the program is not stratified and should be rejected. In this case, the negative cycle involves two predicate symbols and two negative dependencies, but in general a negative cycle consist of any number of dependency edges with at least one negative dependency edge.

3.3 First-Class Datalog Constraints

We now describe the λ_{DAT} calculus, a minimal lambda calculus with first-class Datalog constraints, originally introduced by [30]. We use a slightly simplified version of the calculus that illustrates the challenges posed by stratified negation.

3.3.1 Syntax

The grammar of λ_{DAT} is shown in Figure 4. The language includes the usual constructs from the lambda calculus: constants, variables, lambda abstractions, function applications, and let-bindings. Let-bindings support Hindley-Milner-style parametric polymorphism [11, 21, 33]. The values of λ_{DAT} include constants c, lambda abstractions $\lambda x.\, e$, and *Datalog values* $\#\{C_1, \cdots, C_n\}$. A Datalog value is a collection of Datalog facts and rules. The grammar of Datalog values mirrors that of Figure 3. The fix body atom will be explained in Section 6. The expressions of λ_{DAT} include variables x, values v, function applications $e\, e$, and let-bindings $\mathsf{let}\, x = e\, \mathsf{in}\, e$. The calculus has three expressions for working with Datalog values:

- **(i)** a *composition expression* e_1 <+> e_2 to compute the union of two Datalog values,
- **(ii)** a *project expression* $\mathsf{project}\, p\, e$ to extract all p facts from a Datalog value, and
- **(iii)** a *solve expression* $\mathsf{solve}\, e$ to compute the minimal model of a Datalog value.

The Flix programming language, which implements the λ_{DAT} calculus, supports a richer set of operations for working with Datalog values. However, for our purposes, the above calculus is sufficient to illustrate the challenges. For the full details on the λ_{DAT} calculus, we refer the reader to [30].

3.3.2 Type System

The λ_{DAT} type system is based on Hindley-Milner [12, 47] extended with row polymorphism [28]. Each row type tracks the predicate symbols (and the types of the term parameters of each predicate) of a Datalog expression. The type system is sound; satisfying the usual progress and preservation theorems [30].

The type system splits types into mono- and poly types as shown in Figure 5. The mono types consist of type variables α, a set of base types denoted by ι (e.g. `Bool`), function types $\tau_1 \to \tau_2$, and row types r. A row type is either a row type variable ρ, an empty row $\{\}$, or a row extension $\{p = (\tau_1, \cdots, \tau_n) \mid r\}$. A row type describes the type of a Datalog expression.

▶ **Example 5.** The following Datalog expression is typeable with the shown row type:

$$\#\{\mathsf{Bird}(\text{“Eagle”}).,\ \mathsf{Flying}(x) \Leftarrow \mathsf{Bird}(x), \mathsf{not}\ \mathsf{Penguin}(x).\} : \{\mathsf{Bird} = \mathsf{String} \mid \{\mathsf{Flying} = \mathsf{String} \mid \{\mathsf{Penguin} = \mathsf{String} \mid \rho\}\}\}$$

The order of predicate symbols within a row is immaterial. Hence the same row is equivalent to (written as $\cong$):

$$\{\mathsf{Penguin} = \mathsf{String} \mid \{\mathsf{Flying} = \mathsf{String} \mid \{\mathsf{Bird} = \mathsf{String} \mid \rho\}\}\}$$

Figure 5 shows the poly types (or type schemes) of λ_{DAT}. A poly type is of the form $\forall\alpha_1, \cdots, \alpha_n\, \forall\rho_1, \cdots, \rho_m.\, \tau$. Thus, the λ_{DAT} calculus separates regular type variables α from row type variables ρ.

▶ **Example 6.** The following Datalog expression is typeable with the shown poly type:

$$\#\{\mathsf{Path}(x, z) \Leftarrow \mathsf{Path}(x, y), \mathsf{Edge}(y, z).\} :\ \ \forall\alpha_1, \alpha_2\, \forall\rho.\, \{\mathsf{Path} = (\alpha_1, \alpha_2), \mathsf{Edge} = (\alpha_2, \alpha_2) \mid \rho\}$$

This expression is polymorphic in the types of the terms of the Edge and Path atoms (α_1 and α_2) and row polymorphic in the type of the rest of the row (ρ). As can be seen from the rule, the variables y and z must share the same type (α_2) because of their occurrences in the Path atoms. The two types of polymorphism serve two different purposes: the regular polymorphism allows the expression to be used with terms of different types (e.g. Edge and Path facts over integers, strings, etc) whereas the row polymorphism allows the expression to be composed with other Datalog expressions that may have additional predicate symbols.

The type system of λ_{DAT} has three mutually inductive typing judgments: one for expressions ($\Gamma \vdash e : \tau$), one for constraints ($\Gamma \vdash_c C : r$), and one for atoms ($\Gamma \vdash_p p(t_1, \cdots, t_n) : r$).

3.3.2.1 Type Rules

We briefly describe the (T-Head-Atom) and (T-Constraint) type rules of λ_{DAT}.

The typing judgement $\Gamma \vdash_p p(t_1, \cdots, t_n) : r$ captures that the head or body atom $p(t_1, \cdots, t_n)$ can be typed with row type r under the type environment Γ. In particular, the

$$\frac{\forall i.\, \Gamma \vdash t_i^h : \tau_i}{\Gamma \vdash_p p(t_1^h, \cdots, t_n^h) : \{p = (\tau_1, \cdots, \tau_n) \mid r\}} \quad \text{(T-Head-Atom)}$$

rule states that a head atom can be typed as a row type in which the predicate symbol p is mapped to a tuple type whose elements are the types of the head terms $t_1^h, \cdots, t_n^h$. The type rules for body atoms are similar. What is important, for our purposes, is that to type a head or body atom, its predicate symbol and term types must be part of the row type.

The typing judgement $\Gamma \vdash_c C : r$ captures that the constraint C can be typed as r under the type environment Γ. In particular, the

$$\frac{\Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p A_0 : r \quad \forall i.\, \Gamma, x_1 : \tau_1, \cdots, x_m : \tau_m \vdash_p B_i : r_i \quad \forall i.\, r \cong r_i}{\Gamma \vdash_c \forall(x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \Leftarrow B_1, \cdots, B_n. : r} \quad \text{(T-Constraint)}$$

type rule states that to type an entire constraint, the row type of the head atom and all the body atoms must be equivalent, i.e. contain the same predicate symbols mapped to the same term types, modulo the order of predicate symbols. In λ_{DAT}, the unbound Datalog variables are explicitly quantified.

We refer the reader to [30] for a complete description of the type system. We will use the type system when we define soundness of labelled dependency graphs in Section 5.

3.4 The Problem: Stratification and First-Class Constraints

We are now ready to define what it means for a λ_{DAT} program to be stratified:

▶ **Definition 7** (Stratification for λ_{DAT})**.** *A λ_{DAT} calculus program P is* stratified *if every Datalog value constructed during evaluation of P is stratified.*

The challenge is to statically determine when that is the case. Consider the λ_{DAT} program:

$$f = \lambda c_1.\, \lambda c_2.\, \mathsf{let}\, r = \#\{P(x) \Leftarrow A(x), \mathsf{not}\, Q(x).\}\, \mathsf{in}\, c_1 \texttt{ <+> } c_2 \texttt{ <+> } r$$

To determine whether f returns a stratified program we must know at least:

- whether the argument c_1 is itself stratified,
- whether the argument c_2 is itself stratified,
- whether the composition of c_1 and c_2 is stratified, and finally,
- whether the composition of c_1, c_2 with the rule r is stratified.

At run time, the values of c_1, c_2, and r are known and we can use Ullman's algorithm [45] to compute their stratification. But again, moving the stratification check to run time would force the program to crash if it ever encounters a Datalog value that cannot be stratified!

Before we explore the design space of techniques to ensure compile-time stratification of λ_{DAT} calculus programs, let us pause and reflect on what makes a design "good". As discussed, we are interested in techniques that are fully automatic, hence imposes no additional burden on the programmer. In addition, we want a system that: (i) has high precision (i.e. few programs are unfairly rejected), (ii) is fast (i.e. can be run continuously during program development), (iii) offers understandable error messages (i.e. does not require too much knowledge from the programmer "when things to wrong"), and (iv) is robust under refactoring (i.e. harmless refactorings should not break stratification). As is often the case, some of these goals are conflicting.

4 Dependency Graph Types: A Purely Type-based Approach

We now present a type system that captures the entire dependency graph in the type system itself. This type system is expressive, precise, and its types can be fully inferred. As we shall discuss, it is also impractical, since each type may be quadratic in the number of predicates.

We extend the λ_{DAT} type system to track the *entire* dependence graph of every Datalog expression in the type system. The key idea is straightforward: We represent a dependency edge $p \leftarrow q$ or $p \leftarrow\!\!\!\times\!\!\!- q$ as a single "label" and then use row polymorphism to track a row of all these labels. The type system does not concern itself with the types of terms and should be understood as being in addition to the existing type system.

The new row types are given by the grammar:

$$r = \rho \mid \{\,\} \mid \{\, p \leftarrow q \mid r \,\} \mid \{\, p \leftarrow\!\!\!\times\!\!\!- q \mid r \,\}$$

The type rule for a Datalog constraint is straightforward:

$$\frac{A_0 = p_h(t, \cdots, t) \quad E = \{p_h \leftarrow p_b^i \mid B_i = p_b^i(t, \cdots, t)\} \cup \{p_h \leftarrow\!\!\!\!\times\; p_b^i \mid B_i = \mathsf{not}\, p_b^i(t, \cdots, t)\}}{\Gamma \vdash_c \forall(x_1 : \tau_1, \cdots, x_m : \tau_m).\, A_0 \Leftarrow B_1, \cdots, B_n. : \{E \parallel r\}}$$

where $\{E \parallel r\}$ is the row type with all be labels in the set E, e.g. $\{\{x, y\} \parallel r\} = \{x \mid \{y \mid r\}\}$. Intuitively, the type rule states that if we have a constraint $A_0 \Leftarrow B_1, \cdots, B_n$ where the head predicate symbol is p_h and there is a positive body atom $B_i = p_b^i(t, \cdots, t)$, then the row contains the positive edge $p_h \leftarrow p_b^i$. Similarly, if there is a negative body atom $B_i = \mathsf{not}\, p_b^i(t, \cdots, t)$, then the row contains the negative edge $p_h \leftarrow\!\!\!\!\times\; p_b^i$.

For this type system, we conjecture the important property:

▶ **Theorem 8** (Soundness). *Let $\Gamma \vdash e : r$ and define e' to be e where all the free variables have been substituted for values of the types given in Γ. If $e' \rightarrow^\star v$ then the dependency graph of v is a subset of g, i.e. $\mathcal{DG}(v) \subseteq g$, where g is the graph defined by the edges present in the row type of v.*

We now give two examples of how the type system works. We will use the abbreviations Warm (W), Kettle (K), Cold (C), and Off (O) moving forward:

▶ **Example 9.** The left expression is typeable with the abbreviated type on the right:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```

$$\forall\rho.\, \{\mathsf{C} \leftarrow \mathsf{K}, \mathsf{C} \leftarrow \mathsf{O}, \quad \mathsf{W} \leftarrow \mathsf{K}, \mathsf{W} \leftarrow\!\!\!\!\times\; \mathsf{C}, \mid \rho\}$$

▶ **Example 10.** Consider the expressions on the left and their abbreviated types on the right:

```
let p1 = #{
  Cold(x) :- Kettle(x), not Warm(x).
};
let p2 = #{
  Warm(x) :- Kettle(x), not Cold(x).
};
```

$$p_1 : \forall\rho_1.\, \{\mathsf{C} \leftarrow \mathsf{K}, \mathsf{C} \leftarrow\!\!\!\!\times\; \mathsf{W} \mid \rho_1\}$$

$$p_2 : \forall\rho_2.\, \{\mathsf{W} \leftarrow \mathsf{K}, \mathsf{W} \leftarrow\!\!\!\!\times\; \mathsf{C}, \mid \rho_2\}$$

The composition of p_1 and p_2 has the following row type which contains a negative cycle between W and C and is rejected.

$$\forall\rho_3.\, \{\mathsf{C} \leftarrow \mathsf{K}, \mathsf{C} \leftarrow\!\!\!\!\times\; \mathsf{W}, \mathsf{W} \leftarrow \mathsf{K}, \mathsf{W} \leftarrow\!\!\!\!\times\; \mathsf{C}, \mid \rho_3\}$$

4.1 Discussion

The type system has several advantages:

- it captures the dependency graph of each Datalog expression in its type,
- it supports row polymorphism, and
- it has complete type inference.

The type system is a simple and straightforward application of row types. But the strength of the type system is also its weakness: The type of each expression needs to store the whole dependency graph between all pairs of predicate symbols in the expression. The amount of information to be stored in each type is quadratic in the number of predicate symbols.

To understand why a type needs to store, for each pair of predicate symbols A and B, whether or not A is reachable from B in the dependency graph, consider the Datalog value $\#\{A(x) \Leftarrow \mathsf{not}\, B(x).\}$. If this value is composed with another Datalog value v, the resulting Datalog program is stratified if and only if B does not depend on A in v. Since A and B could be arbitrary predicate symbols, the type of v needs to store, for every possible pair

of predicate symbols (A, B), whether or not there is a dependence in v. While we have not implemented the system, this complexity leads to concerns about efficiency of type inference. In particular, the rows used to track all dependency edges are now very long. Instead, we want to explore the design space of a hybrid approach: We keep the original type system of [30] and we combine it with global information about the constraints in the entire program.

5 Labelled Dependency Graph: A Hybrid Approach

We now describe a hybrid approach that combines local information from the type system with global information about the λ_{DAT} program. As it turns out, the choice of what information to collect about the entire program opens up a large design space.

General Framework

We want to statically ensure that a λ_{DAT} program is stratified in the sense of Definition 7. To do so, we take the following overall approach:

> For each Datalog expression e in a well-typed λ_{DAT} program P (i.e. we have $\Gamma \vdash e : r$), we want to construct a dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ that soundly over-approximates the dependency graph(s) of every Datalog value v that e could evaluate to at runtime. In other words, define e' to be e where all the free variables have been substituted for values of the types given in Γ. These values must be chosen from compositions of the Datalog literals in the program. If $e' \rightarrow^\star v$ then $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ over-approximates $\mathcal{DG}(v)$. The role of the parameter $\mathcal{LG}(P)$ will be discussed shortly.

If the dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ for the expression e is stratified, then every Datalog value v that the expression e could evaluate to must also be stratified: if the over-approximate dependency graph does not contain a negative cycle then any sub-graph cannot contain a negative cycle. If this is true for every expression e in a program P, then the entire program must be stratified in the sense of Definition 7.

We construct the over-approximate graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ using two types of information:

- Local information about the expression e (the $e : r$ part).
- Global information about the entire program P (the $\mathcal{LG}(P)$ part).

We call the data structure that records the global information the *labelled dependency graph* $\mathcal{LG}(P)$. The graph records all (positive and negative) dependencies between predicate symbols in all Datalog rules appearing anywhere in λ_{DAT} the program. The dependency edges are annotated with labels that record various constraints about each dependency. When constructing a specific dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ for a specific expression e of type r, local information about the expression recorded in the type r will be combined with the constraints recorded in the edge labels to determine that certain edges represent global dependencies that are incompatible with some characteristics of the local expression e. These edges from the global dependency graph are removed to construct a more precise local dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ specific to the local expression e.

Formally, we define the labelled dependency graph as:

▶ **Definition 11** (Labelled Dependency Graph). *The* labelled dependency graph $\mathcal{LG}(P)$ *of a* λ_{DAT} *program* P *is a directed graph between predicate symbols where each edge is labelled with information that is used to determine if that edge is possible w.r.t. to a specific row type.*

and we require that the $\mathcal{LG}$ and $\widehat{\mathcal{DG}}$ functions satisfy the following important property:

▶ **Definition 12** (Soundness Criterion). *Given a well-typed λ_{DAT} program P, assume that e is a sub-expression of P and that $\Gamma \vdash e : r$. Define e' to be e where all the free variables have been substituted for values of the types given in Γ. These values must be chosen from compositions of the Datalog literals in P. If $e' \rightarrow^\star v$ then the dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ is a sound over-approximation of the dependency graph of v, i.e. $\mathcal{DG}(v) \subseteq \widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$.*

The choice of what information to record in the labelled dependency graph opens a large design space. For example, it could include information about the constraints that occur in the program, their predicate symbols, and the types of their terms. The design space contains various choices of possible constraints that can be recorded in the labels of the global dependency graph.

5.1 Design Choice 1: Granularity of the Labelled Dependence Graph

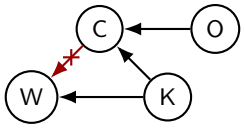
We now turn to the choice of which labels to use on the labelled dependency graph $\mathcal{LG}(P)$.

5.1.1 Degenerate

The simplest choice is to leave the labelled dependency graph unlabelled. This is a degenerate choice which corresponds to the most pessimistic assumption: that all Datalog values could be composed into one big Datalog value. We include it for completeness.

▶ **Example 13.** Consider the Datalog expression on the left:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



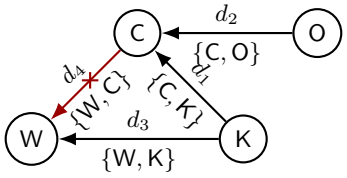
which gives rise to the labelled dependency graph on the right. The dependency edges carry no additional information and hence any local information about the type of a specific Datalog expression cannot help narrow down the set of possible edges. We have to consider all edges as possible.

5.1.2 Source and Destination Granularity

A straightforward improvement is to label each dependency edge with its source and destination predicate. This is information that is already represented by the graph itself.

▶ **Example 14.** Consider again the Datalog expression:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



which now has the labelled dependency graph on the right. Each edge is now labelled with its source and destination predicate symbols. We can use this information as follows.

Suppose we are given an expression e with type r:

$$r = \{\cdots \mid \mathsf{Cold} = \cdots \mid \mathsf{Off} = \cdots \mid \mathsf{Warm} = \cdots \mid \cdots\}$$

where r does not contain the Kettle predicate symbol. If so, we know that e *cannot* evaluate to a Datalog value v which would give rise to the dependency edges d_1 and d_3 (because these

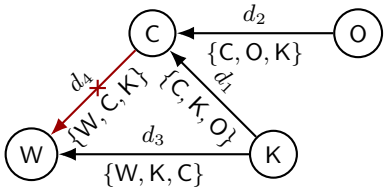
edges are labelled with the Kettle predicate symbols and the type system guarantees that e cannot evaluate to a Datalog value with a Kettle predicate). On the other hand, we cannot exclude the d_2 and d_4 edges because their labels occur in the type.

5.1.3 Rule-level Granularity

A more interesting design choice is to label each dependency edge with *all* predicate symbols that occur in the rule from which the edge originates.

▶ **Example 15.** Consider again the Datalog expression:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



which now has the labelled dependency graph on the right. Each dependency edge is now labelled with *all the predicate symbols that occur in the rule* from where the edge originates.

For example, the edge $C \overset{\{C,K,O\}}{\leftarrow} K$ represents that Cold depends on Kettle but it can only occur if Cold, Kettle, *and* Off can occur in the Datalog value.

Suppose, as before, that we are given an expression e with type r:

$$r = \{\cdots \mid \mathsf{Cold} = \cdots \mid \mathsf{Off} = \cdots \mid \mathsf{Warm} = \cdots \mid \cdots\}$$

where r does not contain the Kettle predicate. We are now able to exclude *all* dependency edges because they are all labelled with Kettle. Intuitively, the Datalog expression on the left uses the Kettle predicate in both rules. Thus, if a Datalog expression does not contain the Kettle predicate then neither rule can contribute to its dependency graph.

Suppose, on the other hand, that we are given an expression e_2 with type r_2:

$$r_2 = \{\cdots \mid \mathsf{Cold} = \cdots \mid \mathsf{Kettle} = \cdots \mid \mathsf{Warm} \mid \cdots\}$$

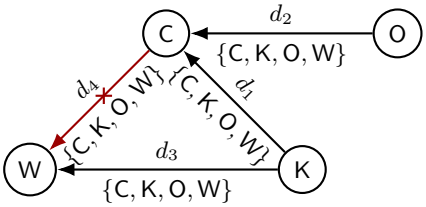
where r_2 does not contain the Off predicate. If e_2 evaluates to a Datalog value v_2, we can exclude the dependency edges d_1 and d_2 from its dependency graph, but we cannot exclude d_3 nor d_4. This is because we are able to exclude the dependency edges from the first rule, but not from the second.

5.1.4 Datalog Value-level Granularity

The last and most powerful option is to label each dependency edge with *all* predicate symbols that occur within the same Datalog literal.

▶ **Example 16.** Consider one last time the Datalog expression:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



which now has the labelled dependency graph on the right. Each dependency edge is labelled with *all the predicate symbols that occur in the same Datalog literal*. For example, the edge $C \overset{\{C,K,O,W\}}{\leftarrow} K$ represents that Cold depends on Kettle but only if *all* the predicates Warm, Cold, Kettle, *and* Off can occur in the Datalog value.

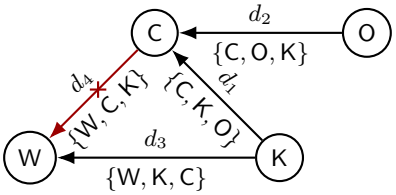
Suppose we are given an expression e with type r:

$$r = \{\cdots \mid \mathsf{Cold} = \cdots \mid \mathsf{Kettle} = \cdots \mid \mathsf{Warm} = \cdots \mid \cdots\}$$

which does not contain the Off predicate symbol. If so, we know that e cannot evaluate to a Datalog value v with *any* of the four dependency edges. Note, in particular, that we are able to exclude the dependency edge $W \nleftarrow C$ even though the predicate symbol Off has nothing to do with Warm or Cold or even the rule from which the edge arises.

While this design choice is very powerful, it suffers from the problem that a simple refactoring can break stratification. For example, if we take the same program and refactor it to:

```
let p1 = #{
  Cold(x) :- Kettle(x), Off(x).
};
let p2 = #{
  Warm(x) :- Kettle(x), not Cold(x).
};
let pr = p1 <+> p2
```



then it is no longer the case that the rule in p_1 must occur together with p_2. Consequently, if we have an expression e with the type r (as above), we can no longer exclude the dependency edges d_3 and d_4. Thus, while this design choice is powerful, it is also brittle under refactoring.

5.1.5 Summary

In summary, the four design choices are:

- **Choice 1a**: The degenerate case where $\mathcal{LG}(P)$ is unlabelled.
- **Choice 1b**: Label the $\mathcal{LG}(P)$ with the source and destination predicate symbols from which the dependency edge arises.
- **Choice 1c**: Label the $\mathcal{LG}(P)$ with *all the predicate symbols that occur in the same rule* from where the dependency edge originates.
- **Choice 1d**: Label the $\mathcal{LG}(P)$ with *all the predicate symbols that occur in the same Datalog literal* from where the dependency edge originates.

For the purpose of exposition, we assume **choice 1c** for the next subsections.

5.2 Design Choice 2: Enriched Labelling and Type Filtering

We can increase precision by including information about the types of the predicate symbols in the labelled dependency graph.

5.2.1 Predicate Symbol Arity

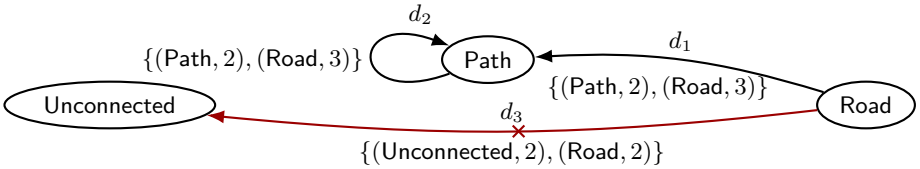
We can include the arity of the predicate symbols in the labelled dependency graph to add precision. We define the labels to be a set of pairs (p, n) of predicate symbols and their arity. We then define $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ to include an edge $a \xrightarrow{\ell} b$ when the arities in ℓ agree with the arities in the row type r.

▶ **Example 17.** Consider a λ_{DAT} program P that contain the two Datalog expressions:

```
#{Path(x, y) :- Road(x, 1, y).
  Path(x, z) :- Path(x, y), Road(y, 1, z).}
```

```
#{Unconnected(x, y) :-
  ..., not Road(x, y).}
```

In the Datalog expression on the left, the Road predicate symbol has three terms whereas in the Datalog expression on the right, the Road predicate has two terms. The labelled dependency graph, $\mathcal{LG}(P)$, is:



which includes the arity of each predicate in the labels on the edges. Suppose the program contains an expression e with the row type r:

$$r = \{\mathsf{Road} = (\mathsf{Int32}, \mathsf{Int32}) \mid \mathsf{Unconnected} = (\mathsf{Int32}, \mathsf{Int32}) \mid \cdots\}$$

If the e evaluates to a Datalog value v, then the type system guarantees that every atom in v with the predicate symbol Road must have two terms, both of which are of type Int32. Thus we can exclude the two dependency edges d_1 and d_2 because the arities on the labels do not match.

5.2.2 Predicate Term Types

We can increase precision even further by including the term types of predicate symbols in the labelled dependency graph. We define labels to be pairs $(p, \overline{\tau})$ of a predicate symbol and the types of its terms, also written $p(\overline{\tau})$ on the labelled graphs. We then define $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ to include an edge $a \xrightarrow{\ell} b$ when the term types in ℓ unify with term types in the row r.

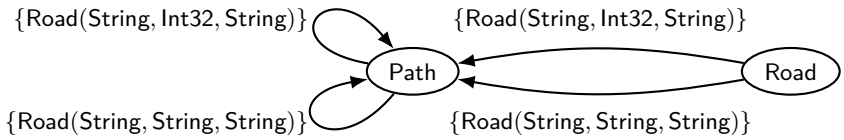
▶ **Example 18.** Consider a λ_{DAT} program P that contain the two Datalog expressions:

```
#{Road("Lyon", 120, "Paris").
  Path(x, z) :- Path(x, y),
                Road(y, l, z).}
```

```
#{Road("Lyon", "Icy", "Paris").
  Path(x, z) :- Path(x, y),
                Road(y, l, z).}
```

In each Datalog expression, the Road predicate has arity two. In the expression on the left, the label on a Road fact represents the current weather (type String), whereas in the expression on the right, the label represents the current speed limit (type Int32).

The labelled dependency graph, omitting the Path predicate in labels, is:



5.2.3 Relational and Lattice Predicate Symbols

In Flix, as we shall discuss further in Section 6, every predicate symbol is given either a relational or a lattice interpretation. The type system ensures that relational and lattice predicate symbols cannot be mixed. That is, within a Datalog value every predicate symbol has exactly one interpretation. Similarly to how we extended the labelled dependency graph with arity and term types, we can also extend it to account for this information.

In summary, the four options are:

- **Option 2a**: Enrich $\mathcal{LG}(P)$ to track the arity of the predicates.
- **Option 2b**: Enrich $\mathcal{LG}(P)$ to track term types of the predicates.
- **Option 2c**: Enrich $\mathcal{LG}(P)$ to distinguish between relations and lattices.
- **Option 2d**: Do not enrich $\mathcal{LG}(P)$.

These options are not mutually exclusive and can be combined.

5.3 Choice 3: Stratify With or Without Monomorphization

We have shown how local information, i.e. information about the row type of an expression, enables us to filter the labelled dependency graph. An orthogonal design choice, which affects precision, is whether to perform stratification with or without monomorphization.

Monomorphization is a compile-time transformation that replaces polymorphic functions by copies that are specialized to their concrete type arguments. For example, if List.map is used with both integer and string lists, then monomorphization generates two copies of List.map: one specialized to integers and one specialized to strings. While the primary purpose of monomorphization is to avoid boxing, monomorphization can also be used to improve the precision of stratification.

Monomorphization improves precision in two ways:

(i) by specializing polymorphic Datalog expressions to concrete types, boosting the precision of type-based filtering, and

(ii) by eliminating unreachable Datalog expressions.

▶ **Example 19.** Consider the Flix program fragment:

```
def f(): #{A(t), B(t)} =
  #{ A(x) :- B(x). }

def g(): #{A(t), B(t)} =
  # { B(x) :- B(x), not A(x). }
```

```
def main(): Unit =
  let c1 = f() <+> A(123).;
  let c2 = g() <+> A("hello").;
  solve c1
```

The functions f and g return Datalog values with row types that contain the predicate symbols A and B, which are polymorphic in the type parameter t. The two Datalog constraints form a negative cycle between A and B. Because the types of the two expressions are

$$\forall\alpha_1, \forall\rho_1.\, \{A = \alpha_1 \mid \{B = \alpha_1 \mid \rho_1\}\} \quad \text{and} \quad \forall\alpha_2, \forall\rho_2.\, \{A = \alpha_2 \mid \{B = \alpha_2 \mid \rho_2\}\}$$

we cannot exclude that these two types could occur in the same Datalog value and consequently we cannot exclude that the overall program might construct a non-stratified Datalog program at runtime. However, if we monomorphize the program first, i.e. specialize f and g to their concrete type arguments whenever they are used in the program, then we obtain the program:

```
def f1(): #{A(Int32), B(Int32)} =
  #{ A(x) :- B(x). }

def g1(): #{A(String), B(String)} =
  # { A(x) :- A(x), not B(x). }
```

```
def main(): #{A(Int32), B(Int32)} =
  let c1 = f1() <+> A(123).;
  let c2 = g1() <+> A("hello").;
  solve c1
```

where $f1$ and $g1$ are no longer polymorphic and the types of their Datalog expressions are:

$$\{A = \mathsf{Int32} \mid B = \mathsf{Int32}\} \quad \text{and} \quad \{A = \mathsf{String} \mid B = \mathsf{String}\}$$

We can now use the types to determine that the two rules cannot occur in the same Datalog value and consequently the program is stratified.

Monomorphization increases precision, but has two practical downsides: the labelled dependency graphs are larger and consequently more costly to stratify, and intertwining monomorphization and stratification may make it difficult for programmers to understand why or when a program fails to be stratified.

In summary, the two design choices are:

- **Choice 3a**: Stratify *without* monomorphization.
- **Choice 3b**: Stratify *with* monomorphization.

6 The Fix Modifier, Lattice Semantics, and Stratification

We now explain the role of the fix modifier and its semantics. Intuitively, the fix modifier enables us to safely use lattice values in relations. Given the rule:

$$A(x) \Leftarrow \mathsf{fix}\, B(x), C(x).$$

The use of "fix" in front of the atom $B(x)$ forces the relation B to be fully computed before the rule is applied. Therefore, A must belong to a stratum strictly greater than B. Thus, the fix modifier has the same effect on stratification as negation. For a normal Datalog program, fix does not change the minimal model, only the evaluation order. However, for Datalog programs with lattice semantics [31], the fix construct solves a long-standing problem.

In Flix, every predicate symbol is associated with a *relational* or *lattice* interpretation. We will write p for a predicate symbol that has a relational interpretation and p_ℓ for a predicate symbol that has a lattice interpretation. We syntactically distinguish between relational and lattice predicates by writing a relational predicate as $A(t_1, \cdots, t_n)$, whereas we write a lattice predicate as $A(t_1, \cdots, t_n; t)$, with a semi-colon before the last term.

▶ **Definition 20** (Key and Lattice Positions). *Given an atom $p(t_1, \cdots, t_n)$ where p has* relational *interpretation, we say that the terms $t_1, \cdots, t_n$ are in* key position. *Given atom $p_\ell(t_1, \cdots, t_n; t)$ where p_ℓ has a* lattice *interpretation, we say that the terms $t_1, \cdots, t_n$ are in* key position *and t is in* lattice *position.*

▶ **Definition 21** (Key and Lattice Variables). *We split variables into* key *and* lattice *variables. A variable is a* key variable *if all its occurrences are in key positions. Otherwise it is a* lattice variable.

As the definition states, a variable that occurs in *both* a key and lattice position is considered a lattice variable. The original version of Flix disallows such "dual-use" of variables; enforcing that lattice variables cannot be used in key position.

▶ **Example 22.** To better understand key and lattice variables, consider the Datalog rules:

```
A(k1, k2; l) :- B(kl, k2; l), C(k1, k2; l). // legal
A(k1, k2, l) :- B(kl, k2; l), C(kl, k2; l). // illegal
A(k1, k2; l) :- B(kl, k2; l), C(k1, k2, l). // illegal
A(k1, l; k2) :- B(kl, k2; l), C(kl, k2; l). // illegal
```

In each rule, the variable l is a lattice variable because it occurs in at least one lattice position. The first rule is legal since the lattice variable l only occurs lattice positions. The second rule is illegal since the lattice variable l occurs in a key position in the head of the rule (where A has a relational interpretation). The third rule is illegal since the lattice variable l occurs in a key position in the body of the rule (where C has a relational interpretation). The fourth rule is illegal since the lattice variable l occurs in a key position in the head of the rule.

Formally, the original version of Flix enforces the *lattice range restriction*:

▶ **Definition 23** (Lattice Range Restriction). *Every lattice variable must occur in a lattice position.*

To understand the lattice range restriction, let us revisit the example from Section 2.

▶ **Example 24.** Figure 2 contains the Datalog rule:

```
Edge(c1, c2) :- fix Component(_; c1), fix Component(_; c2).
```

Assume that the rule did not use fix and that we ignore the lattice range restriction. Suppose we have the following lattice facts:

$$\mathsf{Component}(7; \{1\}). \qquad \mathsf{Component}(7; \{2\}). \qquad \mathsf{Component}(7, \{7\}).$$
$$\mathsf{Component}(5, \{3\}). \qquad \mathsf{Component}(5, \{5\}).$$

The minimal model has two facts: $\mathsf{Component}(7; \{1, 2, 7\})$ and $\mathsf{Component}(5; \{3, 5\})$. We would thus assume that the above rule would compute the undirected edge fact: $\{1, 2, 7\} \leftrightarrow \{3, 5\}$. But this is not what the program computes! It derives nonsensical facts such as $\{1\} \leftrightarrow \{3\}$, $\{1\} \leftrightarrow \{3, 5\}$, and all other edges between intermediate lattice values. The lattice range restriction avoids this problem by banning the program. We propose to allow the program as long as the lattice is fix'ed, i.e. fully computed, before it is used in a relation.

As the example shows, the lattice range restriction is overly strict. We can allow lattice variables to be used in key positions in head atoms provided that we ensure that the head predicate symbol occurs in a strictly higher stratum than every body atom in which the lattice variable occurs. This ensures that the lattice predicates are fully computed before they are used as keys. We introduce the extended dependency graph to capture this notion:

▶ **Definition 25** (Extended Dependency Graph). *The* extended dependency graph *of a Datalog program D with lattice semantics is a directed graph of predicate symbols that contains:*

- *a* weak edge *$a \leftarrow b$ if D contains a rule where a is the predicate symbol of the head atom and b is a predicate symbol of a positive body atom, and*
- *a* strong edge *$a \leftarrow\!\!\!\!\times\; b$ if D contains a rule where a is the predicate symbol of the head and b is a predicate symbol of a* fixed *or* negative *body atom.*

We use the word *strong* to represent either a *fixed* or a *negative* dependency.

We also update the range restriction:

▶ **Definition 26** (Extended Lattice Range Restriction). *If every occurrence of a lattice variable is under a fix in the body of a rule, then the variable may be treated as a key in the head of the rule.*

Finally, we update our definition of stratification for the λ_{DAT} calculus and for Flix:

▶ **Definition 27** (Extended Stratification). *A Datalog program D_L, enriched with lattice semantics, is stratified if the extended dependency graph does not contain a cycle with a strong edge. A λ_{DAT} calculus program P_L, enriched with lattice semantics, is stratified if every Datalog value constructed during evaluation of P_L is stratified.*

▶ **Example 28.** We conclude with a small example of how the fix construct can be used[1]:

```
Degree("Kevin Bacon"; Down(0)).
Degree(x; n + Down(1)) :- Degree(y; n), StarsWith(y, x).
Layer(n; Set#{x}) :- fix Degree(x; n).
Count(n, Set.size(s)) :- fix Layer(n; s).
```

This program computes the number of people who are separated by n-degrees from Kevin Bacon. For example, the 2nd-degree is *the number of people* who have starred in a movie where someone in that movie has also starred in another movie with Kevin Bacon.

[1] Here, the Down constructor defines a lattice with the reverse ordering of the underlying type.

7 Implementation

We now describe where the original version of Flix, and our proposed future version of Flix, reside in the design space.

7.1 The Original Flix Implementation

The original Flix implementation and its associated paper [30] do not use the terminology of this paper. Nevertheless, we can recast their design choices in our framework.

Flix uses a labelled dependency graph constructed from the entire program and filtered based on the type of each Datalog expression, the hybrid approach. The labels are predicate symbols where each dependency edge is annotated with its source and destination (**choice 1b**). No enriched labelling or types are used (**choice 2d**). Stratification is performed *without* monomorphization (**choice 3a**).

7.2 Our Flix Extension

While all design choices are valid, in our view, **choice 1b** and **choice 2d** are sub-optimal.

- For **choice 1b**, by only using the predicate symbols that are the source and destination of each dependency edge, we lose the important information that most dependency edges arise from rules where multiple predicate symbols are involved and thus where *all* of these predicate symbols must be present for the edge to be relevant. Instead, **choice 1c** or **choice 1d** offer increased precision with little downside.
- For **choice 2d**, ignoring the arity and term types of each predicate symbol loses important contextual information. In other logic programming languages, such as Prolog, predicate symbols are often overloaded and use the arity as part of their name, e.g. `spawn/2` and `spawn/3`. Given that λ_{DAT} and Flix are statically typed, it seems like a missed opportunity not to use types to distinguish predicate symbols, e.g. `Path(Int32, Int32)` vs. `Path(String, String)`.

For these reasons, in our proposed extension, we settled on **choice 1c** and **choice 2b** combined with **choice 2c**. We chose **choice 1c** because of its increased precision while still remaining explainable to the programmer. For **choice 2b** and **choice 2c**, we think that incorporating types and the distinction between relational and lattice predicates into the dependence graph increases precision significantly while also remaining understandable to the programmer.

We keep the rest of the design choices the same. We explored the idea of moving the stratifier after monomorphization **choice 3b**, which would boost precision. However, monomorphization is a relatively expensive compiler phase that is traditionally not part of the Flix compiler frontend. Thus, if stratification depends on monomorphization, then it becomes part of the frontend and must be run whenever the program is "type checked" by an IDE. We were worried that this would have unacceptable performance implications[2].

In summary, our Flix extension makes the following design choices:

- We use the hybrid approach based on the *labelled dependency graph*.
- **Choice 1c**: We use *rule-level* granularity.
- **Choice 2b**: We enrich the graph with *term types*.
- **Choice 2c**: We enrich the graph with *relation and lattice* information.
- **Choice 3a**: We stratify *without* monomorphization.

[2] This frontend versus backend problem is not unique to stratification. For example, many C or C++ compilers will report additional compilation warnings or errors when expensive backend optimizations are enabled. This might seem counter-intuitive, but the reason is that expensive program analysis enables the compiler to know more about the program and thus to report more warnings or errors.

7.3 Implementation Details

We have implemented the above design choices in an extension of the Flix compiler.

Flix is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, polymorphic effects, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [30, 31].

The Flix compiler project, including the standard library and tests, consists of 161,000 lines of Flix and Scala code. We re-wrote the Stratifier compiler phase which required ~1,000 lines of code and added support for the "fix" construct which required ~700 lines of code.

Flix, with our extensions, is ready for use, open source, and freely available at:

`https://flix.dev` and `https://github.com/flix/flix`

7.4 When a Program Does Not Stratify

When Flix programmers encounter a stratification error, there are essentially two possibilities:

- The program contains an actual stratification error.
- The type system is too imprecise to rule out the possibility of a stratification error.

We want to support the programmer in both scenarios. First, this means giving the programmer useful error messages such that they can accurately identify which of the two cases is applicable. Second, we want to give the programmer the ability to refactor their program such that it passes the stratification.

If a programmer should encounter a stratification error due to imprecision, they can:

1. Rename a predicate symbol to avoid a clash with a conceptually different predicate symbol. For example, instead of Node, a more suitable name could be City.
2. Introduce an extra predicate symbol in a rule to exclude the clash.
3. Change the arity of a predicate symbol, for example by recording more or less information.
4. Enrich the types of the terms in a predicate; for example, the programmer could introduce a new type Celsius instead of Int32 or a new type City instead of String.

We think these are flexible and reasonable strategies that a Flix programmer will be able to use. Of course these strategies cannot necessarily resolve *all* stratification issues, but the space of accepted programs is much larger than in the original version of Flix where only strategy (1) is available.

7.5 The Motivating Example, Revisited

We now revisit the motivating example from Section 2. Figure 6 shows the labelled dependency graph for the program in Figure 2. The graph reflects our design choices:

(i) the dependency edges are labelled with all predicate symbols from the rule that gives rise to the dependency, and

(ii) the labels carry the term types of each predicate symbol.

The braces and commas of the label sets are omitted. Inspecting the graph, we see two potential negative cycles. However, no expression in the program in Figure 2 has a type where the cycles cannot be excluded.

As an example, on line 26, the composition of the expressions `ns`, `es`, and `r` has the type:

$$\{\mathsf{ComponentRep} = (\mathsf{Int32}, \mathsf{Int32}), \mathsf{Reachable} = (\mathsf{Int32}, \mathsf{Int32}),$$
$$\mathsf{Node} = (\mathsf{Int32}), \mathsf{Edge} = (\mathsf{Int32}, \mathsf{Int32}), \mathsf{ReachUp} = (\mathsf{Int32})\}$$

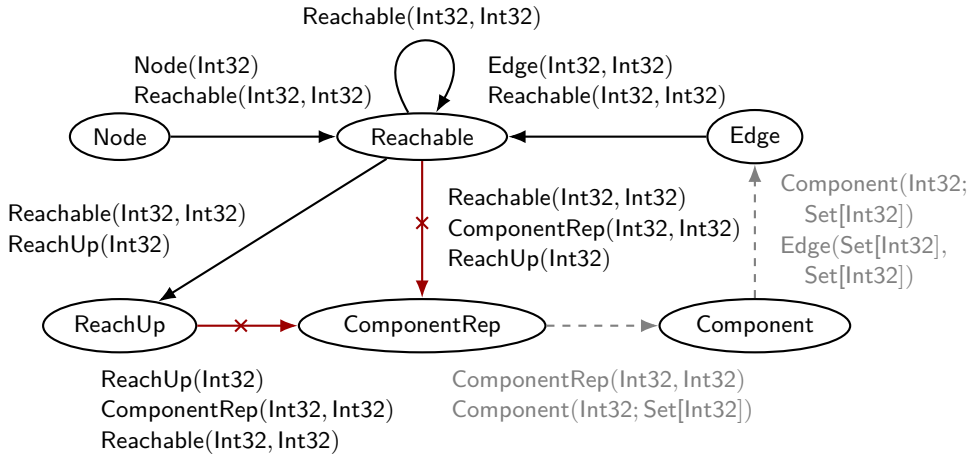


Figure 6 The labelled dependency graph of the program in Figure 2 based on the design decisions of Section 7.2. The gray dashed lines are the edges that are filtered out when looking at the union of `ns`, `es`, `r` on line 26.

This excludes the dashed dependency edges shown in gray in Figure 6. For example, the edge between Component and Edge is ruled out since the term types of Edge do not match and since Component is not present. The resulting graph, shown with solid arrows, does not contain any negative cycles. Hence the expression is stratified. This is true for all expressions in the program, hence Flix is able to statically determine that the program is stratified.

8 Case Study: A Small Graph Library in Flix

We have implemented a small open-source graph library for Flix[3]. The graph library provides a range of queries on graphs, as shown in Table 1. Each query is implemented as a Flix function that internally uses Datalog. The purpose of the case study is to explore how a graph library can be implemented using first-class Datalog constraints in Flix and to check that the proposed stratification strategy is practical.

Table 1 shows an overview of the functions that we have implemented in the graph library. In total, we have implemented 24 functions in 450 lines of code. The columns of the table are as follows: The Function and Lines columns give the name and number of source code lines for a specific function. The Lattices column indicates whether the Datalog program uses lattice semantics. The Stratified Negation column indicates whether the Datalog program uses stratified negation. The Fix Construct column indicates whether the Datalog program uses the fix construct.

To look at a specific example, the table shows that the stronglyConnectedComponents function consists of 26 lines of code, it uses lattice semantics and the fix construct, but it does not use stratified negation. To give an idea of how the library is implemented, the complete source code for the stronglyConnectedComponents function is shown in Figure 7. As the code shows, each query is implemented using first-class Datalog constraints and there is some code-reuse in the form of the nodes and reachability functions which return Datalog program values.

[3] `https://github.com/flix/flix/blob/master/main/src/library/Graph.flix`

Table 1 Overview of the Flix Graph Library. The line count includes the number of lines of helper functions. The features used in a function also include features used in helper functions.

		Features Used		
Function	**Lines**	**Lattices**	**Stratified Negation**	**Fix Construct**
boundedDistances	20	**Y**	N	**Y**
closure	17	N	N	N
cutPoints	30	**Y**	**Y**	**Y**
degrees	19	**Y**	N	**Y**
distance	16	**Y**	N	**Y**
distances	49	**Y**	N	**Y**
distancesFrom	15	**Y**	N	**Y**
flipEdges	10	N	N	N
frontiersFrom	29	**Y**	N	**Y**
inDegrees	18	**Y**	N	**Y**
invert	12	N	**Y**	N
isCyclic	15	N	N	N
outDegrees	18	**Y**	N	**Y**
reachable	16	N	N	N
reachableFrom	15	N	N	N
stronglyConnectedComponents	26	**Y**	N	**Y**
toGraphviz	11	N	N	N
toGraphvizWeighted	11	N	N	N
topologicalsort	28	**Y**	**Y**	N
toUndirected	4	N	N	N
toUndirectedWeighted	4	N	N	N
unreachableFrom	20	N	**Y**	N
withinDistanceOf	14	**Y**	N	**Y**
withinEdgesOf	14	**Y**	N	**Y**

In total, the library contains 31 distinct predicate symbols and 64 rules. The library was developed using our extended Flix compiler. During development, we never encountered a spurious stratification error. However, if we compile the library with the original Flix compiler, it is unfairly rejected due to a spurious negative cycle.

In summary, Table 1 shows that: (i) we have 24 functions that co-exist, using overlapping predicate names from the same domain, without spurious stratification errors, (ii) the majority of functions (15/24) require stratification via not, fix, or both, (iii) many functions use lattice semantics (13/24), (iv) the fix construct is used more often than the not construct, and (v) the library is accepted by our extended Flix compiler, but is rejected by the original Flix compiler due to a spurious negative cycle in the dependency graph.

Flix is a whole-program optimizing compiler. When the graph library is compiled together with the standard library, the stratification is computed in 0.16 seconds whereas the total compilation time is 7.3 seconds. In particular, compilation time is dominated by the cost of type inference. In conclusion, we find the stratification does not unfairly reject our library and that the cost of computing the stratification is low.

```
/// Returns the strongly connected components of the directed
/// graph `g`. Two nodes are in the same component if and only
/// if they can both reach each other.
pub def stronglyConnectedComponents(g: m[(t, t)]): Set[Set[t]]
    with Foldable[m], Boxable[t] = {
  let edges = inject g into Edge;
  let connected = #{
    // If `n1` can reach `n2` and `n2` can reach `n1` then they are
    // part of the same strongly connected component.
    Connected(n1; Set#{n2}) :- Reachable(n1, n2), Reachable(n2, n1).
  };
  let components = #{
    // After the full computation of `Connected`, duplicates are
    // removed by checking that `n` is the minimum in the strongly
    // connected component.
    Components(s) :- fix Connected(n; s), if Some(n) == Set.minimum(s).
  };
  let res = query edges, nodes(), reachability(), connected, components
            select x
            from Components(x);
  List.toSet(res)
}
```

Figure 7 The stronglyConnectedComponents function from the graph library case study.

9 Related Work

9.1 First-class Datalog

Magnus and Lhoták present the λ_{DAT} calculus which is the foundation for the current work [30]. The authors briefly discuss stratified negation and propose a simple solution based on type filtering similar to **choice 1b** without any information on the labels (**choice 2d**). As we have seen, some of these choices are sub-optimal.

9.2 Negation and Aggregation Semantics

There are many proposed semantics for Datalog with negation but stratified negation is the most prevalent one. Kolaitis and Papadimitriou present inflationary semantics that produce facts in such a way that a fixpoint exists for all programs using negation. The fixpoint is not guaranteed to be minimal [26]. There are also variations in the realm of stratification. Negation can be restricted to guarded negation, which in broad terms means that all first-order variables in negated atoms exist in a single atom. This makes additional questions like query containment decidable [5]. Local stratification stratifies the program on instances of rules instead of the quantified rules. This is a property that is hard to verify, but in its most expressive form, it allows deducing $even(i+1)$ from $\neg even(i)$, since this is not circular reasoning for any instantiations of i [36].

Aggregation is non-monotone like negation, which is why it has been studied using many of the same ideas. Aggregation can naturally be stratified like negation but another option is group-stratification based on the standard group-by operation. This means that a group of the predicate should not depend on itself [35]. Zaniolo et al. studies both negation and aggregation with a syntactically restricted form of local stratification that essentially tracks the dynamic strata on the facts [48].

9.3 Datalog Extensions

There have been many efforts to increase the expressive power and usability of Datalog while maintaining practical feasibility. One extension is the existential quantification of variables in the rule head. This was motivated by the ontological reasoning that is needed in web-standards for databases [19]. Datalog$^{\exists}$ is undecidable, so a family of languages called Datalog$^{\pm}$ [8] makes restrictions that reduce the complexity to classes ranging from AC_0 to EXPTIME. One of these is Warded Datalog$^{\pm}$ [19], which has a syntactic restriction on the usage of variables that may be bound to non-constant variables in evaluation and flow into the rule head. They must be within a single predicate, the "ward". It uses stratified negation and negative constraints that restrict the inclusion of certain facts.

10 Conclusion

Flix is a functional, imperative, and logic language with support for first-class Datalog constraints. In Flix, Datalog constraints are values that can be constructed, passed as arguments to functions, returned from functions, composed with other Datalog values, and solved. Flix is based on the λ_{DAT} calculus which itself builds on the Hindley-Milner type system extended with row polymorphism. A significant part of the expressive power of Datalog comes from the use of negation. Stratified negation is a particular simple form of negation that prohibits recursion through negation and is easily accessible to ordinary programmers. While it is straightforward to determine if a Datalog program is stratified, it is much more difficult to statically determine if a λ_{DAT} program is stratified. In this paper, we have explored the design space of stratification for λ_{DAT}. We have proposed several improvements to stratification in Flix and we have implemented these. With our extension, Flix accepts a much broader range of programs that use stratified negation. Finally, we have also extended Flix with a new "fix" construct that enables lattice values to be used as relational values.

References

1 Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, 2011. Springer Berlin Heidelberg.

2 Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive Datalog system DLV. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 282–301, Berlin, 2011. Springer Berlin Heidelberg.

3 Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2723372.2742796.

4 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proc. in Informatics (LIPIcs)*, pages 2:1–2:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.2.

5 Vince Bárány, Balder ten Cate, and Martin Otto. Queries with guarded negation. *Proc. VLDB Endow.*, 5(11):1328–1339, July 2012. doi:10.14778/2350229.2350250.

6 Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428209.

7 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1640089.1640108.

8 Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. In *Proc. of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, pages 77–86, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1559795.1559809.

9 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases: An Overview*, pages 1–15. Springer Berlin Heidelberg, Berlin, 1990. doi:10.1007/978-3-642-83952-8_1.

10 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proc. of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2391229.2391230.

11 Luis Damas. *Type assignment in programming languages*. PhD thesis, The University of Edinburgh, 1984.

12 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/582153.582176.

13 Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22(3):364–418, September 1997. doi:10.1145/261124.261126.

14 Melvin Fitting. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, 278(1):25–51, 2002. Mathematical Foundations of Programming Semantics 1996. doi:10.1016/S0304-3975(00)00330-3.

15 Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092, Berkeley, California, USA, August 2020. USENIX Association.

16 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proc. of International Logic Programming Conference and Symposium*, pages 1070–1080, Cambridge, MA, USA, 1988. MIT Press.

17 Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3):365–385, August 1991. doi:10.1007/BF03037169.

18 G. Gottlob, S. Ceri, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(01):146–166, 1989. doi:10.1109/69.43410.

19 Georg Gottlob and Andreas Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *Proc. of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 2999–3007, Palo Alto, California, USA, 2015. AAAI Press.

20 Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the Myria big data management service. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2588555.2594530.

21 R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. doi:10.2307/1995158.

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

23 Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, Boris Motik, and Ian Horrocks. Foundations of declarative data analysis using limit Datalog programs. In *Proc. of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1123–1130, California, USA, 2017. International Joint Conferences on Artificial Intelligence. doi:10.24963/ijcai.2017/156.

24 Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. Distribution policies for Datalog. In Benny Kimelfeld and Yael Amsterdamer, editors, *21st International Conference on Database Theory (ICDT 2018)*, volume 98 of *Leibniz International Proc. in Informatics (LIPIcs)*, pages 17:1–17:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICDT.2018.17.

25 Ross D. King. Applying inductive logic programming to predicting gene function. *AI Mag.*, 25(1):57–68, March 2004.

26 Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '88, pages 231–239, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/308386.308446.

27 Kenneth Kunen. Negation in logic programming. *The Journal of Logic Programming*, 4(4):289–308, 1987. doi:10.1016/0743-1066(87)90007-0.

28 Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.

29 Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, November 2009. doi:10.1145/1592761.1592785.

30 Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: Programming with first-class Datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428193.

31 Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, PLDI '16, pages 194–208, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2908080.2908096.

32 Hongyuan Mei, Guanghui Qin, Minjie Xu, and Jason Eisner. Neural Datalog through time: Informed temporal modeling via logical specification. In *International Conference on Machine Learning*, pages 6808–6819, Madison, WI, USA, 2020. PMLR, Omnipress.

33 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. doi:10.1016/0022-0000(78)90014-4.

34 Raymond J. Mooney. Inductive logic programming for natural language processing. In Stephen Muggleton, editor, *Inductive Logic Programming*, pages 1–22, Berlin, 1997. Springer Berlin Heidelberg.

35 Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proc. of the Sixteenth International Conference on Very Large Databases*, pages 264–277, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

36 Teodor C. Przymusinski. Chapter 5 - on the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, USA, 1988. doi:10.1016/B978-0-934613-40-8.50009-9.

37 Kenneth A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proc. of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 161–171, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/298514.298558.

38 Jiwon Seo. Datalog extensions for bioinformatic data analysis. *Annu Int Conf IEEE Eng Med Biol Soc*, 2018:1303–1306, July 2018.

39 Jiwon Seo, Stephen Guo, and Monica S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289, Manhattan, New York, USA, 2013. IEEE. doi:10.1109/ICDE.2013.6544832.

40 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed Socialite: A Datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013. doi:10.14778/2556549.2556572.

41 Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with Datalog queries on Spark. In *Proc. of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2882903.2915229.

42 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 245–251, Berlin, 2011. Springer Berlin Heidelberg.

43 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276509.

44 Petar Tsankov. Security analysis of smart contracts in Datalog. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 316–322, Cham, 2018. Springer International Publishing.

45 Jeffrey D. Ullman. Principles of database and knowledge-base systems, 1988.

46 Mario Wenzel and Stefan Brass. Declarative programming for microcontrollers - Datalog on Arduino. In *Declarative Programming and Knowledge Management: Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9–12, 2019, Revised Selected Papers*, pages 119–138, Berlin, 2019. Springer-Verlag. doi:10.1007/978-3-030-46714-2_9.

47 Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

48 Carlo Zaniolo, Natraj Arni, and Kayliang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Deductive and Object-Oriented Databases*, pages 204–221, Berlin, 1993. Springer Berlin Heidelberg.