Fixpoints for the Masses: Programming with First-Class Datalog Constraints

MAGNUS MADSEN, Aarhus University, Denmark
ONDŘEJ LHOTÁK, University of Waterloo, Canada

Datalog is a declarative logic programming language that has been used in a variety of applications, including big-data analytics, language processing, networking and distributed systems, and program analysis.

In this paper, we propose first-class Datalog constraints as a mechanism to construct, compose, and solve Datalog programs at run time. The benefits are twofold: We gain the full power of a functional programming language to operate on Datalog constraints-as-values, while simultaneously we can use Datalog where it really shines: to declaratively express and solve fixpoint problems.

We present an extension of the lambda calculus with first-class Datalog constraints, including its semantics and a type system with row polymorphism based on Hindley-Milner. We prove soundness of the type system and implement it as an extension of the Flix programming language.

CCS Concepts: • Software and its engineering → Functional languages; Constraint and logic languages.

Additional Key Words and Phrases: functional programming, logic programming, first-class datalog

1 INTRODUCTION

Datalog is a simple, yet surprisingly powerful, declarative logic programming language. A Datalog program is a collection of constraints. Each constraint is a fact or a rule. Together, the facts and rules imply a minimal model, a unique solution to any Datalog program [Ceri et al. 1989].

Datalog has roots in the database community [Ullman 1984] and has been used in a wide variety of applications [de Moor et al. 2011; Huang et al. 2011], including in bioinformatics [King 2004], big-data analytics [Halperin et al. 2014; Seo et al. 2013; Shkapsky et al. 2016], natural language processing [Mooney 1996], networking and distributed systems [Alvaro et al. 2010; Conway et al. 2012; Loo et al. 2009], program understanding [Hajiyev et al. 2006], and program analysis [Bravenboer and Smaragdakis 2009; Lam et al. 2005; Smaragdakis and Bravenboer 2011].

Datalog has several properties of theoretical and practical interest: (i) every Datalog program eventually terminates, (ii) every Datalog program has a unique solution, (iii) efficient and parallel evaluation strategies exist, and (iv) any polynomial time algorithm can be expressed in Datalog [Papadimitriou 1985]. For practical purposes, (i)–(iii) are very useful as they enable easy reasoning and debugging of Datalog programs.
Datalog programs are truly declarative: Unlike Prolog, the order of constraints and the order of predicates within a constraint is immaterial in Datalog; we may freely reorder the program without changing its semantics. The solution to a Datalog program, its minimal model, is defined without reference to any specific evaluation order or strategy. Thus, Datalog cleanly separates the what from the how, i.e. the specification is separate from its implementation.

Modern Datalog solvers are increasingly efficient [Jordan et al. 2016, 2018; Scholz et al. 2016; Subotic et al. 2018; Veldhuizen 2012], implementing many important optimizations, such as index selection, query planning, join-ordering, and parallel execution [Bancilhon et al. 1985; Graefe 1993; Gregory 1987; Ullman 1984]. In hand-crafted fixpoint computations, such optimizations must be re-discovered and re-implemented, a waste of programmer effort.

Despite these advantages, use of Datalog is not widespread. We speculate that there are at least two barriers to a broader adoption of Datalog: (i) the poor integration of Datalog into general-purpose programming languages, and (ii) the lack of mechanisms to construct modular Datalog programs. To expand on the former, use of Datalog today often involves generating constraints, storing them into a file, invoking the Datalog engine on the file, and reading the results back into memory for further processing. To expand on the latter, we are rarely interested in a specific Datalog program, but rather in a family of related Datalog programs. Since Datalog lacks a module system, programmers often resort to program generation. Such approaches, whether based on textual generation or template programming, are often inflexible and error-prone.

To overcome these issues, we propose Datalog programs as first-class values that can be constructed, composed, and solved within a functional programming language. We can leverage the features of functional programming to build modular and parametric families of Datalog programs. We can build pipelines of Datalog programs where the output facts of one program are passed as input facts to another program. We can define functions and refer to them from inside Datalog constraints. And finally, we can use parametric polymorphism to express Datalog constraints that are polymorphic in the types of their terms. In this paper, we show how to integrate first-class constraints into a statically-typed eager functional language. We describe the static and dynamic semantics, interaction with lexical scoping, and how to support stratified negation.

We identify and overcome two technical challenges in the presence of first-class constraints: (i) how can we design a modular type system in which reusable fragments of Datalog programs can be typed independently such that when they are composed, at run-time, the composition is type-safe? Specifically, the type system should support abstraction such that each module can be provided by multiple different implementations, and (ii) in the presence of negation, how can we ensure, at compile-time, that every such composition that could occur at run-time is stratified?

In summary, the contributions of this paper are:

- **(Calculus)** We show how to extend the lambda calculus with first-class Datalog constraints.
- **(Type System)** We show how a Hindley-Milner style type system for a functional programming language can be seamlessly extended to type Datalog program fragments as first-class values. Inspired by record calculi, the type system uses row polymorphism to precisely track the predicate symbols that occur in each Datalog fragment. The type system supports full type inference and we prove type safety of the system.
- **(Stratification)** We observe that there are trade-offs between modularity and type complexity in solutions for ensuring stratified negation. As a starting point, we propose one sound technique to compute stratification at compile-time.
- **(Implementation)** We implement our system in the Flix programming language.
- **(Evaluation)** We present case studies that illustrate the usefulness of our system.
2 MOTIVATION

We motivate the need for first-class Datalog constraints with several toy examples. In Section 6, we present real-world applications implemented in our system. All programs shown in this paper are valid and executable programs in our system.

2.1 Example I

As a student of history might know, Pompey the Great was the son of the wealthy equestrian Gnaeus Pompeius Strabo. Pompey had five wives with whom he had two sons and one daughter. We can capture these familial relations with the Datalog program:

```
ParentOf("Pompey", "Strabo").
ParentOf("Gnaeus", "Pompey").
ParentOf("Pompeia", "Pompey").
ParentOf("Sextus", "Pompey").
```

In ancient Rome, the lineage of a statesman was an important part of his stature. We can use Datalog to elegantly compute the ancestors of every person, i.e. the transitive closure of the ancestor relation:

```
AncestorOf(x, y) :- ParentOf(x, y).
AncestorOf(x, z) :- AncestorOf(x, y), AncestorOf(y, z).
```

The solution to this program, its minimal model, contains the fact AncestorOf("Sextus", "Strabo"), because Strabo was the grandfather of Sextus. We are now able to reason about family relations in ancient Rome. However, familial ties where not only by blood, as adoption was equally recognized.

With this in mind, we can extend the Datalog program with the facts:

```
AdoptedBy("Augustus", "Caesar").
AdoptedBy("Tiberius", "Augustus").
```

to record that Caesar adopted Augustus who himself later adopted Tiberius. We can easily include adoptions in the ancestor relation with the addition of a single rule:

```
AncestorOf(x, y) :- AdoptedBy(x, y).
```

This example demonstrates the power of Datalog: We can extend the semantics of a program by simply adding new facts and rules! But what if we had wanted to keep the original program? We would have to maintain two versions of our program: one with ancestry based on biological parents and one extended to include adoptions. We could store these two programs as copies in separate files, but what happens when we discover a bug in one program? Or when we want to extend the other? We quickly run into the multiple maintenance problem. What we want is the ability to construct, compose, and solve Datalog programs at run-time. What we need are first-class constraints.

Here is a reformulation of both programs as a single program in our extension of Flix:

```
def getParents[r](): #{ParentOf(String, String) | r} = #{
    ParentOf("Pompey", "Strabo").
    ParentOf("Gnaeus", "Pompey").
    ParentOf("Pompeia", "Pompey").
    ParentOf("Sextus", "Pompey").
}

def getAdoptions[r](): #{AdoptedBy(String, String) | r} = #{
    AdoptedBy("Augustus", "Caesar").
    AdoptedBy("Tiberius", "Augustus").
}
```
def heritage(withAdoptions: Bool): #{ParentOf(String, String),
AncestorOf(String, String), AdoptedBy(String, String) | r} =
let p1 = #{
  AncestorOf(x, y) :- ParentOf(x, y).
  AncestorOf(x, z) :- AncestorOf(x, y), AncestorOf(y, z).
};
let p2 = #{
  AncestorOf(x, y) :- AdoptedBy(x, y).
};
if (withAdoptions) p1 else (p1 <+> p2)

def ancestors(withAdoptions: Bool): #{AncestorOf(String, String) | r} =
let facts = getParents() <+> getAdoptions();
let model = solve (facts <+> heritage(withAdoptions));
project AncestorOf model

This program illustrates the key idea of our work: Datalog constraints are first-class values. We can pass them around, compose them with other Datalog values, and solve them.

The functions getParents and getAdoptions return sets of facts about biological parents and adoptions. The heritage function encapsulates the two variants of the Datalog program. The function constructs the two Datalog program values $p_1$ and $p_2$, and returns either $p_1$, corresponding to the program solely with biological parents, or the composition (i.e. union) of $p_1$ and $p_2$, corresponding to the program with biological and adoptive parents. The ancestor function assembles the facts and rules, computes their minimal model (with solve), and returns the AncestorOf facts. The getParents, getAdoptions, and ancestor functions have a row type parameter $r$. If the parameter were removed, each function would return a closed row which could not be combined with Datalog constraints sets containing other predicate symbols.

The program illustrates the following important principles of our design:

**Principle I**: Datalog constraints are first-class values that can be passed around.

**Principle II**: Datalog constraints can be composed with other Datalog constraints to form larger Datalog programs.

**Principle III**: The design preserves the essence of Datalog: the constraints are declarative, they look like ordinary Datalog clauses, and they are solvable by standard techniques.

2.2 Example II

We now consider a simple reachability problem: Given a road network with speed limits on each road, we want to determine if it is possible to drive from one city to another city going at least a certain speed. We can write a function $r$ that uses Datalog to compute if this is possible:

def r(g: #{Road(City, Int, City)}, src: City, dst: City, speed: Int): Bool =
let p = #{
  Path(x, y) :- Road(x, speedLimit, y), if speedLimit > speed.
  Path(x, z) :- Path(x, y), Road(y, speedLimit, z), if speedLimit > speed.
};
(solve g <+> p) |= #{ Path(src, dst). }

The function $r$ takes a road network in the form of a set of road facts $g$, a source city $src$, a destination city $dst$, and a minimum speed we want to go on each road. Each rule is equipped with a boolean-valued filter expression that must be true in the minimal model. The expression $e_1 |= e_2$ returns true if $e_2$ is a subset of $e_1$. In other words, if there is a path with the desired speed.
If we want a leisurely drive, we may wish to avoid roads which have a high speed limit. Instead of changing the $r$ function, we can generalize it by equipping it with a predicate $q$ that determines whether the speed limit on a road is acceptable:

```plaintext
def r(g: #{Road(City, Int, City)}, src: City, dst: City, q: Int -> Bool): Bool =
let p = #{
  Path(x, y) :- Road(x, speedLimit, y), if q(speedLimit).
  Path(x, z) :- Path(x, y), Road(y, speedLimit, z), if q(speedLimit).
};
(solve g <+> p) |= #{Path(src, dst).}
```

We can imagine that roads carry not only speed limits, but also other meta-data such as the presence of construction work, the current weather conditions, etc. We can further generalize $r$ to be polymorphic in the type of the information associated with a road:

```plaintext
def r[a](g: #{Road(City, a, City)}, src: City, dst: City, q: a -> Bool): Bool =
let p = #{
  Path(x, y) :- Road(x, metaData, y), if q(metaData).
  Path(x, z) :- Path(x, y), Road(y, metaData, z), if q(metaData).
};
(solve g <+> p) |= #{Path(src, dst).}
```

These programs illustrate the following important principles of our design:

**Principle IV:** Datalog constraints may refer to expressions in the functional language.

**Principle V:** The types of the predicates of a Datalog constraint may be polymorphic.

### 2.3 Benefits of Type Safety

We can imagine a programmer accidentally writing a program like:

```plaintext
let g = #{
  Edge(Paris, 120, Lyon).
  Edge(Lyon, 110, Rome).
};
let q = #{
  Path(x, y) :- Edge(x, y).
  Path(x, z) :- Path(x, y), Edge(y, z).
};
solve (g <+> q)
```

where the arity of the predicate symbol `Edge` is inconsistent. In the facts of `g`, the `Edge` relation is ternary, whereas in the rules of `q`, the `Edge` relation is binary. The composition of `g` and `q` is meaningless: In a Datalog program every predicate symbol must have a fixed arity. In the presence of first-class Datalog constraints, evaluation of such a program would get stuck at run-time. In this paper, we propose a static type system, based on row polymorphism, to reject such programs.

As a continuation of Example II, we can also imagine a programmer accidentally writing:

```plaintext
let g = #{ Road(Paris, true, Lyon). ... };
solve r(g, Paris, Rome, speed -> speed > 60)
```

This program would get stuck inside the Datalog engine, since when the function `speed -> speed > 60` is applied to the value `true`, we get the stuck term: `true > 60`. The proposed type system also rejects such programs.

**Principle VI:** Type safety: Well-typed programs do not get stuck.
P ∈ Programs = C₁, · · · , Cₙ
C ∈ Constraints = A₀ ⇐ B₁, · · · , Bₙ
A ∈ Head Atoms = p(t₁, · · · , tₙ)
B ∈ Body Atoms = p(t₁, · · · , tₙ) | not p(t₁, · · · , tₙ)
t ∈ Terms = x | c
c ∈ Literals = is a set of literal constants.
x, y ∈ VarSym = is a set of variable symbols.
p, q ∈ PredSym = is a set of predicate symbols.

Fig. 1. Datalog Syntax.

These examples demonstrate the benefit of a type system. A library based approach would suffer from such run-time errors. In addition, from a practical point of view, a language-based approach has several advantages: (i) we can determine whether a Datalog program is stratified (and compute its stratification) at compile-time (see Section 4.7), (ii) we can have proper syntax and integration with lexical scoping (as shown in the examples), and (iii) we can provide understandable type and stratification error messages.

3 THE \( \lambda_{\text{Dat}} \) CALCULUS

We now present \( \lambda_{\text{Dat}} \), a minimal lambda calculus with first-class Datalog constraints, which is the formal foundation of our implementation. We begin with a brief recap of Datalog, then discuss the syntax, semantics, and type system of \( \lambda_{\text{Dat}} \).

3.1 A Brief Recap of Datalog

We briefly discuss the syntax and semantics of Datalog. Readers who are already familiar with Datalog may wish to skip this subsection. A comprehensive introduction to Datalog is available in Ceri et al. [1989, 2012]. In this paper, we focus on stratified Datalog, a variant of Datalog with a restricted form of negation [Minker 1988].

Syntax. A Datalog program \( P \) is a collection of constraints \( C₁, · · · , Cₙ \). A constraint, also called a Horn clause, is of the form \( A₀ ⇐ B₁, · · · , Bₙ \) where \( A₀ \) is the head atom and each \( Bᵢ \) is a body atom. A head atom \( p(t₁, · · · , tₙ) \) consists of a predicate symbol \( p \) and a sequence of terms \( t₁, · · · , tₙ \). A body atom is similar to a head atom, except it can be negated, which is written with the not keyword in front of the predicate symbol. A constraint without a body is called a fact. Conversely, 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 1 shows the grammar of Datalog. As an example, the constraint:

\[
\text{Path}(x, z) ⇐ \text{Path}(x, y), \text{Edge}(y, z).
\]

is a rule with predicate symbols Path and Edge and variables \( x, y \) and \( z \). Datalog rules are implicitly universally quantified by their variables, hence the rule is formally written as:

\[
∀x, ∀y, ∀z. \text{Path}(x, z) ⇐ \text{Path}(x, y), \text{Edge}(y, z).
\]

In the calculus, we will write constraints with universal quantifiers. In the implementation, as shown in the previous section, the compiler will automatically determine which variables are implicitly quantified and which are bound by the lexical scope. Thus, like in traditional Datalog programs, the programmer never has to explicitly introduce variables.
$v \in \text{Val} = c \mid \lambda x.\ e \mid \# \{C_1, \ldots, C_n\}$

e \in \text{Exp} = x \mid v \mid e \mid \text{let } x = e \text{ in } e \\
\quad | \ e \leftrightarrow e \mid \text{solve } e \mid \text{subset } e \mid e \mid \text{project } p\ e \\
\quad | S(e) \mid H(A, e, e)$

$C \in \text{Constraints} = \forall(x_1 : \tau_1 \ldots x_n : \tau_n). A_0 \leftarrow B_1, \ldots, B_n$

$A \in \text{Head Atoms} = p(t^h_1, \ldots, t^h_m)$

$B \in \text{Body Atoms} = p(t^b_1, \ldots, t^b_n) \mid \neg p(t^b_1, \ldots, t^b_n)$

$t^h \in \text{Head Terms} = x \mid c \mid e$

$t^b \in \text{Body Terms} = x \mid c$

c \in \text{Literals} = \text{is a set of literal constants.}

x, y \in \text{VarSym} = \text{is a set of variable symbols.}

p, q \in \text{PredSym} = \text{is a set of predicate symbols.}$

Fig. 2. Syntax of $\lambda_{\text{Dat}}$.

**Semantics.** The model-theoretic semantics of Datalog describes the solution, i.e. minimal model, of any Datalog program independent of the mechanism used to compute it [Ceri et al. 1989, 2012; Fitting 2002; Gelfond and Lifschitz 1988, 1991; Kunen 1987]. Intuitively, a Datalog model is simply a set of facts. The minimal model is the smallest set of facts that satisfies all constraints. The model-theoretic semantics is what makes Datalog declarative: we can write a Datalog program, and someone else can write a Datalog solver, and we can independently agree on what the result ought to be. In this paper, we shall not be particularly concerned with how the minimal model is computed. Rather, we will model a generic Datalog solver with very mild assumptions about its behavior. Consequently, the calculus will work with any Datalog solver independent of its specific implementation details. We will model the solver as a black box that, when given a set of Datalog constraints, is permitted to repeatedly select any constraint, pick a type-safe valuation of its quantified variables, instantiate the constraint with the valuation, evaluate its terms (expressions), and add the head atom to the current constraint set.

### 3.2 Syntax of $\lambda_{\text{Dat}}$

We now turn to the syntax of $\lambda_{\text{Dat}}$ which extends the lambda calculus with first-class constraints. The grammar of $\lambda_{\text{Dat}}$ is shown in Figure 2. The language includes the usual expressions from the lambda calculus: constants, variables, lambda abstraction, function application, and let-bindings.

**Values.** The values of $\lambda_{\text{Dat}}$ include literal constants $c$, lambda abstractions $\lambda x.\ e$, and constraint sets $\# \{C_1, \ldots, C_n\}$. A constraint set is a set of enriched Datalog constraints. In our extension of Flix, a single fact or rule can be written without $\#\{\}$. We will use this in subsequent examples when there is no risk of confusion. The grammar of constraints mirrors that of Figure 1, but with three important differences: (i) the implicit universally quantified variables are made explicit along with their types, (ii) every constraint is extended with a filter expression on the implication arrow, and (iii) the terms of a head atom may now be expressions. The last two extensions enrich the expressive power of the constraints.

A constraint set $\# \{C_1, \ldots, C_n\}$ is a set of the constraints $C_1, \ldots, C_n$. We define two Datalog constraints $C_1$ and $C_2$ to be equal when they are syntactically identical.
A key insight is that these properties enable a design of first-class constraints which supports modular and local reasoning. A similar design for Prolog falls apart because the order of clauses is significant and simply changing the order of two clauses may cause non-termination.

1Under the assumption that e₁, e₂, and e₃ are pure (i.e. have no side-effects) and total (i.e. always terminate).

Expressions. The expressions of λ_dat include variables x, values v, function application e e, and let-bindings let x = e in e. The calculus has four expressions for working with first-class constraints: (i) a composition expression e₁ ↔ e₂ to compute the union of two constraint sets, (ii) a subset expression subset e₁ e₂ to determine if one constraint set is a subset of another constraint set, (iii) a project expression project p e to extract all facts of a given predicate symbol from a constraint set, and (iv) a solve expression solve e to compute the minimal model of a constraint set. Finally, the calculus has two internal constructs: S(e) and H(A, e) to model the execution of the Datalog solver. The two expressions are not considered part of the surface syntax of the language. Figure 2 shows the grammar of the expressions in λ_dat.

Composing Constraints. We can compute the union of two constraints sets with the composition expression e ↔ e. Unlike in Prolog, the order of constraints in Datalog is immaterial. Hence, composition is both commutative and associative¹:

\[
(e₁ ↔ e₂) ↔ e₃ = e₁ ↔ (e₂ ↔ e₃)
\]

Furthermore, repeating a Datalog constraint has no effect, so composition is idempotent:

\[
e ↔ e = e
\]

¹Under the assumption that e₁, e₂, and e₃ are pure (i.e. have no side-effects) and total (i.e. always terminate).
Computing the Minimal Model. We can compute the minimal model of a constraint set with the solve expression. The expression evaluates to a new constraint set that consists of all the facts in the minimal model of e. The constraint set does not contain any rules.

The solve expression is idempotent since the minimal model of a set of facts is the set itself:

\[
\text{solve } (\text{solve } e) = \text{solve } e
\]

(Solve-Idempotent)

Why is Solve Explicit? The solve operation must be explicit because in the presence of composition of constraint sets, we cannot identify a constraint set with its minimal model. If we compute the minimal model eagerly, immediately when two constraints are composed, then the order of composition determines the minimal model. For example, if we have the program:

\[
\begin{align*}
\text{let } & a = A(1).; \\
\text{let } & b = B(1).; \\
\text{let } & q = R(x) :- A(x), \text{ not } B(x).
\end{align*}
\]

Then the eager computation of \((a <+> b) <+> q\) is not equivalent to the eager computation of \((a <+> q) <+> b\). The minimal model of the former does not contain an \(R\) fact, but the latter does. We view such order dependence as completely antithetical to the declarative nature of Datalog. Consequently, our design requires explicit solve expressions.

Comparing Constraint Sets. We can compute whether one constraint set is a subset of another with the subset expression \(\text{subset } e_1 e_2\). The expression evaluates to \(\text{true}\) if all constraints in \(e_1\) appear in \(e_2\). The expression does not compute the minimal model of \(e_1\) nor \(e_2\). By lifting a fact into the singleton set, the subset expression can be used to determine if the minimal model of a program contains that fact.

Projecting Constraint Sets. We can use the project expression \(\text{project } p e\) to select the facts in \(e\) which share the predicate symbol \(p\). The project expression is useful when we have computed the minimal model of a program, and we want to select a certain type of facts for further computation.

Enriched Datalog Constraints. We enrich Datalog constraints in two ways: (i) we introduce a filter expression as a guard on a constraint, and (ii) we allow arbitrary expressions as head terms.

For (i), a constraint with a filter expression \(e\) is written as: \(\forall(x_1 : \tau_1, \ldots, x_n : \tau_n). A_0 \leftarrow B_1, \ldots, B_n.\) The expression \(e\) may refer to any of the quantified variables \(x_1, \ldots, x_n\) of the constraint. Filter expressions enrich the semantics of a Datalog constraint by allowing an arbitrary expression to determine when a ground instance of the constraint holds. But the cost is decidability: the programmer must take adequate measures to ensure that the expression \(e\) always terminates.

For (ii), a constraint may now have arbitrary expressions as terms in its head atom. This increases the expressiveness at the cost of decidability. For example, we can now perform addition from within constraints:

\[
\text{Var } (r, v1 + v2) :- \text{Add } (r, x, y), \text{ Var } (x, v1), \text{ Var } (y, v2).
\]

But the consequence is that the Herbrand Base is no longer finite, and as such we have lost most of the important properties of Datalog. For example, we can now write a program with a constraint that never terminates:

\[
\text{P}(0). \text{ P}(x + 1) :- \text{P}(x).
\]

With great power comes great responsibility: If the programmer chooses to use the enriched Datalog constraints, then he or she must ensure termination and that the Herbrand base remains finite.
We define evaluation of λ (or Church-style) semantics where the evaluation rules depend on the type of the expressions: in states that to combine two constraint sets, we compute their union. The evaluation rule for solve:

\[
\text{solve } E \rightarrow \text{subset } E \rightarrow \text{project } E \rightarrow \text{constraint composition:} \quad S(E) \mid \mathcal{H}(A, e, E) \mid \mathcal{H}(A, E, v)
\]

Fig. 4. Evaluation Contexts of λ\textsubscript{Dat}.

**Internal Solver Expressions.** We model the semantics of the Datalog solver with two internal solver expressions. The two internal expressions are not part of the surface syntax of the language, but they are expressions that appear during evaluation.

The select expression \( S(\#\{C_1, \cdots, C_n\}) \) represents an internal state where the Datalog solver is in the process of computing the fixed point of the constraint set \( \#\{C_1, \cdots, C_n\} \) and needs to select and instantiate a constraint. The extend expression: \( \mathcal{H}(A, e, \#\{C_1, \cdots, C_n\}) \) represents an internal state where the Datalog solver is in the process of evaluating the head terms of an atom and its filter expression. If the filter expression evaluates to \( \text{true} \), then the evaluated head atom is added to the constraint set, and the solver continues. If the filter expression evaluates to \( \text{false} \), then the solver continues with the original constraint set. The purpose of the two internal solver expressions is to mimic the most fundamental operations of a Datalog solver: rule selection, rule instantiation, and evaluation of head terms. The idea is that these two expression can be substituted for an actual implementation while preserving the type safety of the calculus.

### 3.3 Evaluation of λ\textsubscript{Dat}

We define evaluation of λ\textsubscript{Dat} as a small-step operational semantics. The evaluation relation \( e \rightarrow e' \) describes when the expression \( e \) can reduce to the expression \( e' \) in a single step. We define an intrinsic (or Church-style) semantics where the evaluation rules depend on the type of the expressions: in other words, we assign no meaning to untyped programs. This is required, since for the Datalog solver to correctly instantiate a Datalog constraint, it must know the types of its quantified variables. The evaluation rules of λ\textsubscript{Dat} are shown in Figure 3. We now discuss the most important rules:

The evaluation rule for constraint composition:

\[
\text{E-Compose} \quad \frac{v_1 = \#\{C_1^1, \cdots, C_1^n\} \quad v_2 = \#\{C_2^1, \cdots, C_2^m\}}{v_1 \leftrightarrow v_2 \rightarrow \#\{C_1^1, \cdots, C_1^n, C_2^1, \cdots, C_2^m\}}
\]

states that to combine two constraint sets, we compute their union. The evaluation rule for solve:

\[
\text{E-Solve} \quad \frac{\text{solve } \#\{C_1, \cdots, C_n\} \rightarrow S(\#\{C_1, \cdots, C_n\})}{v_1 \subseteq v_2}
\]

states that a solve expression reduces to the internal select expression.

The first evaluation rule for the subset expression:

\[
\text{E-Subset-T} \quad \frac{v_1 = \#\{C_1^1, \cdots, C_1^n\} \quad v_2 = \#\{C_2^1, \cdots, C_2^m\}}{v_1 \subseteq v_2}
\]

states that a constraint set \( v_1 \) is a subset of \( v_2 \) if all constraints in \( v_1 \) also occur in \( v_2 \). The (E-Subset-F) rule is similar.

The evaluation rule for projection:

\[
\text{E-Project} \quad \frac{v = \{p(v_1, \cdots, v_m) \mid p(v_1, \cdots, v_m) \in \#\{C_1, \cdots, C_n\}\}}{\text{project } p \#\{C_1, \cdots, C_n\} \rightarrow v}
\]
states that given the predicate symbol \( p \) and the constraint set \( \#\{C_1, \ldots, C_n\} \), the expression reduces to those ground facts in \( \#\{C_1, \ldots, C_n\} \) that share the same predicate symbol. Note that the project expression returns ground facts, and implicitly strips out non-ground facts and rules.

The first evaluation rule for the internal select expression:

\[
C_i = \forall(x_1 : \tau_1, \ldots, x_n : \tau_n). A \xleftarrow{e} B_1, \ldots, B_n' \\
v \text{ is a primitive valuation of } x_i \text{ s.t. } \text{typeOf}(v(x_i)) = \tau_i \\
S(\#\{C_1, \ldots, C_n\}) \rightarrow \mathcal{H}(v(A), v(e), \#\{C_1, \ldots, C_n\}) \tag{S-Rule}
\]

states that given a constraint set \( \#\{C_1, \ldots, C_n\} \), the Datalog solver may non-deterministically select any constraint \( C_i \) and any primitive valuation \( v \) of the quantified variables \( x_1 : \tau_1, \ldots, x_m : \tau_m \). The valuation must assign a value of type \( \tau_i \) to each quantified variable \( x_i \) and the value must be a primitive value. We do not allow complex types, such as functions and constraint sets, as terms. The evaluation rule is in Church-style since evaluation depends on the types of the quantified variables \( x_1, \ldots, x_n \). Given a primitive valuation \( v \), the expression applies it to the head atom \( A \) and the filter expression \( e \), and reduces to the extend expression with these components together with the original constraint set. Intuitively, the (S-Rule) models the Datalog solver when it selects a Datalog rule for evaluation, binds its quantified variables, and proceeds to evaluate the head atom.

The second evaluation rule for the internal select expression:

\[
v = \{p(v_1, \ldots, v_n) \mid p(v_1, \ldots, v_n) \in \#\{C_1, \ldots, C_n\}\} \\
S(\#\{C_1, \ldots, C_n\}) \rightarrow v \tag{S-Finish}
\]

states that the select expression \( S(\#\{C_1, \ldots, C_n\}) \) may immediately evaluate to the facts of \( \#\{C_1, \ldots, C_n\} \). Together, the (S-RULE) and (S-Finish) evaluation rules model a non-deterministic execution where constraints are repeatedly instantiated, their head atoms become part of the constraint set (if the filter expression reduces to \( \text{true} \)), and at some point the facts of the constraint set are returned. This over-approximates the concrete evaluation steps of any reasonable Datalog engine without modeling its exact semantics.

The extend expression has two evaluation rules, (H-TRUE) and (H-FALSE), depending on the value of the filter expression. The first rule:

\[
A = p(v_1, \ldots, v_n) \\
\mathcal{H}(A, \text{true}, \#\{C_1, \ldots, C_n\}) \rightarrow S(\#\{A, C_1, \ldots, C_n\}) \tag{H-True}
\]

states that the expression \( \mathcal{H}(A, \text{true}, \#\{C_1, \ldots, C_n\}) \) reduces to the select expression where the constraint set has been extended with the fact \( A \). Note that the side-condition requires that all terms of \( A \) are values, i.e. that \( A \) is a ground fact.

Substitution. As is standard, we use capture avoiding substitution to implement beta reduction for \( \lambda_{\text{DAT}} \). The substitution rules are available in the technical report [Madsen and Lhoták 2020].

Evaluation Contexts. As is also standard, we use evaluation contexts to allow reduction of sub-expressions [Felleisen et al. 2009]. Figure 4 shows the evaluation contexts of \( \lambda_{\text{DAT}} \). We use \( \square \) to represent the hole in the expression tree. If \( E \) is an evaluation context, then \( E[e] \) is an expression in which the hole \( \square \) has been replaced by the expression \( e \). We assume that the semantics are extended with the rule:

\[
e \rightarrow e' \\
E[e] \rightarrow E[e'] \tag{E-Context}
\]
In the proofs, we will sometimes refer to a specific instantiation of this meta-rule in combination with a specific evaluation context. For example, we might refer to the context rule in reference to the first evaluation context for function application, i.e. the context rule with $E = E' e$ for some $E'$.

The evaluation contexts for $\lambda_{\text{Dat}}$ are straightforward. In the case of the extend expression, the context enforces that we evaluate the constraint set first, then the filter expression, and finally the terms (expressions) of the head atom itself.

4 TYPING OF $\lambda_{\text{Dat}}$

We now discuss the type system for $\lambda_{\text{Dat}}$. The type system is based on Hindley-Milner [Damas and Milner 1982; Wright and Felleisen 1994] extended with row types. We use row types to precisely track the types of predicate symbols in constraint sets. In this way, the type system is reminiscent of type systems for extensible records [Leijen 2005]. We have proved soundness of the type system. The detailed proofs are available in the technical report [Madsen and Lhoták 2020].

4.1 Mono and Poly Types

The type system, as is standard for Hindley-Milner style type systems, splits types into mono- and poly types. Figure 5a shows the mono types of $\lambda_{\text{Dat}}$. The mono types consist of type variables $\alpha$, a set of base types denoted by $i$ (e.g. $\text{Bool}$), function types $\tau_1 \to \tau_2$, tuple types $(\tau_1, \cdots, \tau_n)$, and row types $r$. A row type is either a row type variable $\rho$, an empty row $\{\}$, or a row extension $\{p = (\tau_1, \cdots, \tau_n) | r\}$. A row type describes the types of the predicate symbols of a constraint set. For example, the constraint set:

$$\#\{A(123)., B("Hello World")\}$$

may be assigned the row type:

$$\{A = (\text{Int}) | \{B = (\text{Str}) | \{\}\}\}$$

For brevity, we will write such types as: $\{A = (\text{Int}), B = (\text{Str})\}$. In our extension of Flix both constraint sets and row types are written using $\#\{\}$. In the calculus, for clarity, we write constraint sets as $\#\{\cdots\}$ and row types as $\{\cdots\}$ without the hash $\#$.

Figure 5b shows the poly types (or type schemes) of $\lambda_{\text{Dat}}$. A poly type is of the form:

$$\forall \alpha_1, \cdots, \alpha_n \forall \rho_1, \cdots, \rho_m. \tau$$

As can be seen, we separate regular type variables $\alpha$ from row type variables $\rho$. We use poly types to type polymorphic constraint sets. For example, the constraint set:

$$\#\{\text{Path}(x, z) \leftarrow \text{Path}(x, y), \text{Edge}(y, z)\}$$

is given the polymorphic type:

$$\forall \alpha_1, \alpha_2 \forall \rho. \{\text{Path} = (\alpha_1, \alpha_2), \text{Edge} = (\alpha_2, \alpha_2) | \rho\}$$

The type is polymorphic in the terms of the Edge and Path atoms. Specifically, the variable $x$ must have type $\alpha_1$ whereas the variables $y$ and $z$ must have type $\alpha_2$, because the two occurrences of the
Path atom force the variables y and z to have the same type. But, the type is also polymorphic in the rest of the row ρ, since the constraint set can be composed with other constraint sets that (potentially) contain additional predicate symbols.

As is standard, we introduce a partial order on types [Damas and Milner 1982; Wright and Felleisen 1994]. Given two poly types σ₁ and σ₂, we say that σ₁ is more general than σ₂, written as σ₁ ⊑ σ₂, if there is a substitution s of quantified variables of σ₁ such that s(σ₁) = σ₂. For example:

∀α.α → α ⊑ Int → Int
∀ρ.{A = (Int) | ρ} ⊑ {A = (Int), B = (Int)}
∀α, ρ.{A = α, B = (Int) | ρ} ⊑ {A = (Int), B = (Int), C = (Bool)}

The least element of the partial order is ∀α.α.

4.2 Type Equality

Consider the two constraint sets:

# {A(123).B(456).} and # {B(456).A(123).}

We can assign them the two types:

{A = (Int), B = (Int)} and {B = (Int), A = (Int)}

These two types are not equivalent, since the order of the predicate symbols is different! But clearly, from a Datalog perspective, the two constraints sets are equivalent.

To fix this, we introduce an equivalence relation on types. Specifically, we consider two row types to be equivalent up to permutation of distinct labels following Leijen [2005]. Figure 6 specifies this equivalence relation ≡ on types. The [Eq-Refl] and [Eq-Trans] rules specify that the relation is reflexive and transitive. In the technical report [Madsen and Lhoták 2020], we prove that the relation is symmetric, hence it is an equivalence relation. The [Eq-Arrow] and [Eq-Tuple] rules specify that function and tuple types are equivalent if their constituents are. The [Eq-Head] rule specifies that two row types are equivalent if their first predicate symbols are the same, they have equivalent types, and the rest of the two rows are equivalent. The [Eq-Swap] rule specifies that two row types are equivalent if they are equivalent when the first two predicates of one of them are swapped, provided that the predicate symbols are distinct. With this in place, we have that:

{A = (Int), B = (Int)} ≡ {B = (Int), A = (Int)}

The type system permits rows with duplicate predicate symbols. In the context of extensible records with scoped labels such types make sense and can be useful, but in the context of our work they are more of an artifact. We tolerate them because they simplify the formalism and the implementation of type inference. We refer the reader to Leijen [2005] for more details.
4.3 Examples

Before we present the type system in detail, we give some examples of how it works.

Example I. Consider the program fragment:

```plaintext
let p1 = Edge("a", "b").
let p2 = Path("a", "b").
let p3 = p1 <+> p2;
```

The type system will assign the two local variables `p1` and `p2` the row polymorphic types:

- `p1 : \forall r_1.\{\text{Edge} = (\text{Str}, \text{Str}) | r_1\}`
- `p2 : \forall r_2.\{\text{Path} = (\text{Str}, \text{Str}) | r_2\}`

The type rule for composition:

```
\Gamma \vdash e_1 : r_1 \quad \Gamma \vdash e_2 : r_2 \quad r_1 \equiv r_2
----------------------------- (T-Compose)
\Gamma \vdash e_1 <+> e_2 : r
```

requires that the types of `p1` and `p2` are equivalent. We can instantiate the two polymorphic types using the type rule for variables:

```
(x, \sigma) \in \Gamma \quad \sigma \subseteq \tau
--------------- (T-Var)
\Gamma \vdash x : \tau
```

which allow us to replace `r_1` by `{\text{Path} = (\text{Str}, \text{Str}) | r_3}` and `r_2` by `{\text{Edge} = (\text{Str}, \text{Str}) | r_3}`. We pick the same type variable `r_3` for both instantiations; otherwise the composition rule would be inapplicable. The type rule for composition now allows us to conclude that `p_1 <+> p_2` has type:

```
\{\text{Edge} = (\text{Str}, \text{Str}), \text{Path} = (\text{Str}, \text{Str}) | r_3\}
```

Using the type rule for let:

```
\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2
------------------ (T-Let)
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2
```

We obtain the polymorphic type:

```
\forall r_3.\{\text{Edge} = (\text{Str}, \text{Str}), \text{Path} = (\text{Str}, \text{Str}) | r_3\}
```

for the local variable `p_3`.

Example II. Consider the program fragment for some labelled graph:

```plaintext
let p1 = Edge("a", 123, "b").
let p2 = Path(x, l, z) :- Path(x, l, y), Edge(y, l, z).
let p3 = p1 <+> p2;
```

The types of the three local variables are:

- `p_1 : \forall r_1.\{\text{Edge} = (\text{Str}, \text{Int}, \text{Str}) | r_1\}`
- `p_2 : \forall \alpha_1, \alpha_2, \alpha_3, \forall r_2.\{\text{Edge} = (\alpha_1, \alpha_2, \alpha_3), \text{Path} = (\alpha_1, \alpha_2, \alpha_3) | r_2\}`
- `p_3 : \forall r_3.\{\text{Edge} = (\text{Str}, \text{Int}, \text{Str}), \text{Path} = (\text{Str}, \text{Int}, \text{Str}) | r_3\}`

Note how the type of `p_2` is polymorphic in terms of `Edge` and `Path`, but once `p_1` and `p_2` are composed in `p_3`, the types of both `Edge` and `Path` become `(\text{Str}, \text{Int}, \text{Str})`.
Example III. If we modify the previous program to the following:

```plaintext
let p1 = Edge("a", 123, "b").;
let p2 = Path(x, 11, z) :- Path(x, 11, y), Edge(y, 12, z).;
let p3 = p1 <+> p2;
```

We get the more interesting types:

```plaintext
p1 : ∀r1. {Edge = (Str, Int, Str) | r1}
p2 : ∀a1, a2, a2', a3, ∀r2.
    {Edge = (a1, a2, a3), Path = (a1, a2', a3) | r2}
p3 : ∀a2', ∀r3.
    {Edge = (Str, Int, Str), Path = (Str, a2', Str) | r3}
```

Note how the type of p2 is now polymorphic in both labels l1 (with type a2') and l2 (with type a2). Consequently, p3 becomes polymorphic in the label l1 (a2').

Ill-typed Example I. The program fragment:

```plaintext
let p1 = Edge("a", "b").;
let p2 = Edge("a", 42, "b").;
let p3 = p1 <+> p2;
```

cannot be typed since the types of p1 and p2:

```
∀r1. {Edge = (Str, Str) | r1}  and  ∀r2. {Edge = (Str, Int, Str) | r2}
```

cannot be unified.

Ill-typed Example II. The program fragment:

```plaintext
let p1 = Edge("a", 12345, "b").;
let p2 = Edge("a", "abc", "b").;
let p3 = p1 <+> p2;
```

cannot be typed since the types of p1 and p2:

```
∀r1. {Edge = (Str, Int, Str) | r1}  and  ∀r2. {Edge = (Str, Str, Str) | r2}
```

cannot be unified.

Ill-typed Example III. Similarly, the single constraint:

```
Path(x, 42, z) :- Path(x, "foo", y), Edge(y, z).
```

cannot be typed since the types of the Path atom in the head and body cannot be unified.

4.4 Type Rules

We now present the type rules of ≡DAT. The type rules are a syntax-directed formulation of the Hindley-Milner type system where instantiation occurs in the type rule for variables [T-V AR] and generalization occurs in the type rule for let-bindings [T-Let] [Damas and Milner 1982; Wright and Felleisen 1994]. Figure 7 shows the type rules of ≡DAT. The type system has three type judgements: Γ ⊢ e : τ for expressions, Γ ⊢ C : r for constraints, and Γ ⊢ A : r for atoms. We assume that there is a function typeof : Literal → Type that assigns a primitive type to every constant literal.

We now discuss the most important type rules:

The type rule for composition was shown in Example I. It states that e1 and e2 must have row types r and r', and they must be equivalent r ≡ r':

```
Γ ⊢ e1 : r  Γ ⊢ e2 : r'  r ≡ r'  (T-Compose)
```

The type rule for project states that e must have a row type where the predicate p has some type (τ1, · · · , τn), and then the result is a new row type with the same type for the predicate p:
The type rule for the internal extend expression states that $A$ must have row type $r$ under the typing judgement for atoms $r_p$, the filter expression $e_1$ must have type $\text{Bool}$, and the constraint set $e_2$ must have row type $r'$. Finally, $r$ and $r'$ must be equivalent, i.e., $r \equiv r'$:

\[
\frac{\Gamma \vdash A : r \quad \Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : r' \quad r \equiv r'}{\Gamma \vdash \mathcal{H}(A, e_1, e_2) : r}
\] (T-Extend)

The type rule for the internal select expression states that $e$ must have a row type $r$ and the result has type $r$:

\[
\frac{\Gamma \vdash e : r \quad r \equiv \{ p = (\tau_1, \cdots, \tau_n) \mid r' \}}{\Gamma \vdash \text{project } p e : \{ p = (\tau_1, \cdots, \tau_n) \mid r'' \}}
\] (T-Project)

where $\{\tau_1, \cdots, \tau_n, \rho_1, \cdots, \rho_n, r\} = \text{ftv}(r) \setminus \text{ftv}(\Gamma)$.

\[
\frac{\Gamma \vdash e : r}{\Gamma \vdash \mathcal{S}(e) : r}
\] (T-Select)
The type rule for a constraint set states that the type of a constraint set \( \#\{C_1, \ldots, C_n\} \) is a row type \( r \) if each of the constraints \( C_i \) are typeable under the \( \vdash_c \) typing judgement with some row type \( r_i \) and all of the row types are equivalent to \( r \):

\[
\forall i. (\Gamma \vdash_c C_i : r_i \land r_i \equiv r) \\
\Gamma \vdash \#\{C_1, \ldots, C_n\} : r
\]

(T-Constraint-Set)

Intuitively, the rule ensures that the types of the predicates within multiple constraints have consistent types.

The type rule for a single constraint is:

\[
\Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash_p A_0 : r \\
\forall i. \Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m \vdash_p B_i : r_i \\
\forall i. r \equiv r_i \\
\Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m. A_0 \leftarrow B_1, \ldots, B_n : r
\]

(T-Constraint)

It states that a constraint \( \forall(x_1 : \tau_1, \ldots, x_m : \tau_m). A_0 \leftarrow B_1, \ldots, B_n \) has row type \( r \) under a type environment \( \Gamma \) if we can type its constituents as follows: (i) the head atom \( A_0 \) must have type \( r \) under the extended environment \( \Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m \), (ii) each body atom \( A_i \) must have type \( r_i \) under the extended environment \( \Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m \), (iii) all the row types, \( r \) and each \( r_i \) must be equivalent, and (iv) the filter expression \( e \) must have type \( \text{Bool} \) under the extended environment \( \Gamma, x_1 : \tau_1, \ldots, x_m : \tau_m \). Intuitively, the rule ensures that the types of the atoms within a single constraint have consistent types.

The type rule for a head atom is:

\[
\forall i. \Gamma \vdash t^h_i : \tau_i \\
\Gamma \vdash p(t^h_1, \ldots, t^h_n) : \{p = (\tau_1, \ldots, \tau_n) \mid r\}
\]

(T-Head-Atom)

It states that the type of an atom \( A = p(t^h_1, \ldots, t^h_n) \) is a row type of the form \( \{p(\tau_1, \ldots, \tau_n) \mid r\} \) for some row type \( r \) provided that each expression term \( t^h_i \) has type \( \tau_i \) according to the expression typing judgement \( \vdash \). The type rules [T-Body-Atom-1] and [T-Body-Atom-2] are similar.

### 4.5 Type Safety

We have established type soundness for \( \lambda\text{Dat} \). We state the most important theorems here:

**Theorem 4.1 (Progress).** Suppose \( e \) is a closed, well-typed expression (that is, \( \vdash e : \tau \) for some \( \tau \)). Then either \( e \) is a value or else there is some expression \( e' \) such that \( e \rightarrow e' \).

**Theorem 4.2 (Preservation).** If \( \Gamma \vdash e : \tau \) and \( e \rightarrow e' \), then \( \Gamma \vdash e' : \tau' \) where \( \tau = \tau' \).

The detailed proofs are available in the technical report [Madsen and Lhoták 2020].

### 4.6 Type Inference

The type system, as an instance of Hindley-Milner, supports full type reconstruction. The type system for \( \lambda\text{Dat} \) is based on that for polymorphic extensible records [Leijen 2005]. We leave it as future work to prove its correctness, but we believe that the proofs should be easily adaptable. The relationship between records and first-class constraints is as follows: Given the Datalog constraint:

\[
p_0(t^0_1, \ldots, t^0_l) \iff p_1(t^1_1, \ldots, t^1_{l_1}), \ldots, p_n(t^n_1, \ldots, t^n_{l^n})
\]

we can view it as a record in which the fields are the predicate symbols \( p_0, \ldots, p_n \) and the types of the fields are the types of the terms, i.e. the "field" \( p_0 \) would have the tuple type \( (\tau_1, \ldots, \tau_l) \), where \( \tau_i \) is the type of the term \( t^i_l \).
4.7 Stratification

Unrestricted use of negation poses problems for Datalog. Consider the following program:

\[ P(x) :- \text{not } Q(x). \quad Q(x) :- \text{not } P(x). \]

If we assume that the program contains the constant \( c \), then the program has two minimal models:

\[ m_1 = \{ P(c) \} \quad \text{and} \quad m_2 = \{ Q(c) \} \]

This is undesirable because the minimal model is no longer unique. We can ensure a consistent semantics by restricting ourselves to stratified Datalog [Minker 1988]. Stratification requires that the predicate symbols of a Datalog program can be partitioned into a sequence of strata such that a positive predicate symbol in a stratum only depends on predicate symbols in the same or lower strata and a negative predicate symbol only depends on predicate symbols in strictly lower strata. Stratification splits a Datalog program \( P \) into a sequence of sub-programs \( P_1, \ldots, P_n \) such that the output of \( P_i \) becomes the input of \( P_{i+1} \).

**Definition 4.3.** (Precedence Graph) The precedence graph of a program \( P \) is a graph that contains:

- a positive edge \( a \leftarrow b \) if \( P \) contains a rule where \( a \) is the head predicate and \( b \) is a positive body predicate, and similarly
- a negative edge \( a \nleftrightarrow b \) if \( P \) contains a rule where \( a \) is the head predicate and \( b \) is a negative body predicate.

We can use the precedence graph to determine whether a Datalog program is stratified:

**Definition 4.4.** (Stratified) A program \( P \) is stratified if its precedence graph contains no cycles with a negative edge.

Given a complete Datalog program, we can determine if it is stratified and compute its stratification using Ullman’s Algorithm [Ullman 1988]. For a language like \( \lambda_{\text{Dat}} \), we can identify three levels of modularity of a type system or static analysis. At the least modular level, we require the complete Datalog program for analysis of stratification. At the second level, we require the complete \( \lambda_{\text{Dat}} \) program, which contains Datalog constraint sets that will be composed at run time. Finally, at the most modular level, we may be given fragments of \( \lambda_{\text{Dat}} \) programs, and be required to construct a summary of each fragment such that stratification of a complete \( \lambda_{\text{Dat}} \) program can be computed using the summaries alone.

For the first level, stratification of the complete Datalog program can be computed using Ullman’s Algorithm [Ullman 1988]. In the context of \( \lambda_{\text{Dat}} \), however, such an approach would only be applicable at runtime when the complete Datalog programs are known, so stratification errors would be detected only at runtime. The strategy that we will describe in the rest of this section supports the second level of modularity, which allows analyzing Datalog constraint sets at compile time. The analysis can therefore detect stratification errors at compile time. The analysis in this section does not support the third level of modularity. At this level, there is a range of design choices in how detailed and complex the summaries of the \( \lambda_{\text{Dat}} \) fragments should be. An analysis is deemed modular if it reads the summaries instead of the code. At the extreme, if the full details contained in the code are recorded in the summaries, then any analysis can be made modular in name, but then analyzing the summaries will be equivalent to analyzing the original code. At this level, there is a rich design space with trade offs between precision and complexity of the summaries which we leave for exploration in future work.

**Definition 4.5 (Stratification for \( \lambda_{\text{Dat}} \)).** Given a complete \( \lambda_{\text{Dat}} \) program \( e \), determine at compile-time whether all Datalog constraint sets that \( e \) may construct at run-time are stratified.
We propose a simple and sound algorithm that computes stratification at compile time using information from the type system. We view this as an important first step towards stratification of Flix programs, but there is still a rich design space to be explored.

**Definition 4.6.** (First-Class Constraint Stratification) For a whole \( \lambda_{\text{Dat}} \) or Flix program:

1. Collect all constraints into a set \( G \), regardless of where they occur in the program.
2. Compute the global dependency graph \( D = \text{dg}(G) \) of the constraint set \( G \).
3. For each expression \( e \) with a row type \( r = \{ p_1 = (\cdots), \ldots, p_n = (\cdots) \} \), compute a subgraph \( R \) of the dependency graph with only those edges \( a \leftarrow b \) or \( a \leftrightarrow b \) where both predicates \( a \) and \( b \) occur in the row type. Finally, use Ullman’s Algorithm to determine if the dependency graph \( R \) is stratified, and to compute its stratification.

We do not need to consider row type variables because stratification occurs after monomorphization.

For example, consider the following program fragment:

```plaintext
let r1 = B(x) :- A(x).;
let r2 = C(x) :- B(x).;
let r3 = K(x) :- A(x).;
let r4 = A(x) :- \text{not} C(x), R(x).;
```

The dependency graph \( G \) of this program fragment is:

\[
\{ B \leftarrow A, C \leftarrow B, K \leftarrow A, A \leftrightarrow C, A \leftarrow R \}
\]

The dependency graph is not stratified, since there is a cycle with a negative edge:

\[ C \leftarrow B \leftarrow A \]

However, if we consider the expression: \( r3 \leftrightarrow r4 \) which has the row type:

\[ \{ A = (\cdots), C = (\cdots), K = (\cdots), R = (\cdots) \} \]

we see that its restricted dependency graph \( R \) does not have any edges for \( B \):

\[ \{ K \leftarrow A, A \leftrightarrow C, A \leftarrow R \} \]

and hence the constraint sets that the expression \( r3 \leftrightarrow r4 \) may evaluate to must be stratified and we can compute the stratification at compile-time. We now prove correctness of the algorithm.

**Lemma 4.7.** Given any subset of constraints \( U \subseteq G \), the dependency graph of \( U \) is a subgraph of the dependency graph of \( G \), i.e. \( \text{dg}(U) \subseteq \text{dg}(G) \) by the definition of the dependency graph.

**Theorem 4.8 (Correctness).** If algorithm 4.6 reports that an expression \( e \) is stratified, then all constraint sets that \( e \) may evaluate to must be stratified.

**Proof.** An expression with row type \( \{ p_1 = (\cdots), \ldots, p_n = (\cdots) \} \) may evaluate to some constraint set \( Q \) which can be partitioned into facts \( F \) and rules \( C \) such that \( Q = F \cup C \) and we have that \( C \subseteq G \). By inversion of the typing derivation of \( e \), the constraint set \( C \) can only contain constraints with predicate symbols drawn from \( p_1, \ldots, p_n \). By definition, the precedence graph \( \text{dg}(C) \) can then only contain edges with predicates drawn from \( p_1, \ldots, p_n \). By the previous lemma, the dependency graph \( \text{dg}(C) \) must be a subgraph of \( \text{dg}(G) \), but the largest subgraph of \( \text{dg}(G) \) with edges where the predicates are drawn from \( p_1, \ldots, p_n \) is \( R \). Hence we must have that \( \text{dg}(C) \subseteq R \). Now, if \( R \) is stratified, then \( \text{dg}(C) \) must also be stratified, since if a graph has no negative cycles then a subgraph cannot have any negative cycles. \( \square \)

The algorithm is an over-approximation, and it will sometimes unfairly reject programs that can never fail at run-time. For example, the program:
def f(b: Bool): #{...} =
  let r1 = P(x) :- A(x), not Q(x);.
  let r2 = Q(x) :- A(x), not P(x);.
  solve (if (b) r1 else r2)

is unfairly rejected, even though it is always stratified.

5 IMPLEMENTATION

We have implemented $\lambda_{\text{Dat}}$ as an extension of the Flix programming language. Flix is a functional, imperative, and logic programming language. Flix supports algebraic data types, pattern matching, currying, higher-order functions, extensible records, channel and process-based concurrency, and now first-class Datalog constraints.

The extension required 5,000 lines of code in addition to the 55,000 lines of code already present in the compiler. The implementation effort was significant, but not difficult, as the constructs of $\lambda_{\text{Dat}}$ integrate seamlessly with an ML-style language. Most changes were to the frontend of the compiler and to the intricate details of the type inference algorithm. We reuse the Datalog engine that already comes with Flix to solve Datalog constraints at run-time.

Flix is open source, ready for use, and freely available at: flix.dev

6 EVALUATION: CASE STUDIES

To demonstrate the practical value of the $\lambda_{\text{Dat}}$ calculus and its implementation in Flix, we present a series of case studies. We report on programs that have already been implemented in Flix and on one program that could benefit from being implemented in Flix (or in a system based on $\lambda_{\text{Dat}}$).

6.1 Case Study: Koans

We have implemented eleven “koans” which are small programs that demonstrate how to program with first-class constraints. The koans solve practical computational problems such as: (i) Given a Git commit graph, find the pair of commits where a bug was introduced and where the bug was merged into the master branch. (ii) Given a social network graph, compute a set of friend suggestions based on the friends of my friends. (iii) Given a train and bus network, compute if there is a route from one city to another with a preference for the train. (iv) Given a list of graphs, find all pairs of graphs whose union is acyclic. The koans illustrate the interplay between functional and logic programming: the overall program is constructed from smaller functions that use first-class constraints. The koans are available in the technical report [Madsen and Lhoták 2020].

6.2 Case Study: PuppetMaster an Actor Library with Declarative Actor Supervision

PuppetMaster is an actor library for Flix. An actor is a light-weight process that executes concurrently with—and in isolation from—other processes. Every actor has its own unique mailbox, an unbounded queue of incoming messages. Actors communicate by sending immutable messages to each other’s mailboxes. An actor system is a collection of actors together with policies that govern how actors are started, how actors are stopped, and how to respond when an actor crashes. In PuppetMaster, startup, shutdown, and supervision policies are expressed as first-class Datalog constraints. The “input” (i.e. ground facts) of a policy is the state of every actor in the system and the “output” (i.e. minimal model) of a policy is a set of actions to be executed by the actor system.

For example, the very simple policy (where ActorPolicy is a type alias omitted for brevity):

```f
  def immediatelyStartAllPolicy(): ActorPolicy =#
    Start(actor) :- Actor(actor).
  }
```
specifies that every actor should be started immediately. A more sophisticated policy is:

```python
def defaultStartPolicy(): ActorPolicy = 
    Waiting(x) :- DependsOn(x, y), not ActorState(y, Running).
    Start(x) :- Actor(x), not Waiting(x).
```

which allows an actor to start once all its dependencies have entered the Running state. An even more sophisticated policy is:

```python
def defaultSupervisionPolicy(): ActorPolicy = 
    Path(x, y) :- DependsOn(x, y).
    Path(x, z) :- Path(x, y), DependsOn(y, z).
    Waiting(x) :- Path(x, w), ActorState(w, NonResumablyCrashed).
    Resume(x) :- ActorState(x, ResumablyCrashed), not Waiting(x).
```

which allows a resumably crashed actor \( x \) to continue execution unless one of its transitive dependencies have non-resumably crashed. The PUPPETMASTER library ships with a collection of such startup, shutdown, and supervision policies, but the programmer may also define his or her own policies using first-class constraints.

### 6.3 Case Study: A Prototype Points-To Analysis for Python

We have implemented a prototype points-to analysis for Python with a special focus on precise object initialization. The points-to-analysis is parameterized by the choice of context- and heap-sensitivity in the form of two functions, `merge` and `record`, following Smaragdakis et al. [2011]:

```python
def analysis(merge: (String, octx, String, ctx) -> ctx,
              record: (String, ctx) -> octx, empty_ctx : ctx): 
    #{...}
```

where the analysis function is polymorphic in the type of the call context `ctx` and the type of the heap context `octx`. The `merge` and `record` functions are used within the analysis rules:

```plaintext
Reachable(attrObject, merge(baseObject, baseOCtx, invo, callerCtx)) :-
    VCall(base, attr, invo, inMeth),
    VarPointsTo(base, callerCtx, baseObject, baseOCtx),
    AttrPointsTo(baseObject, baseOCtx, attr, attrObject, octx),
    ObjectIsFunction(attrObject),
    Reachable(inMeth, callerCtx).
```

This design allows us to easily change and experiment with different types of context- and heap sensitivity. We can instantiate the analysis with no context or heap sensitivity by simply using the Unit type and passing in two constant functions. We can also instantiate the analysis with 2-call-1-heap sensitivity by defining appropriate algebraic data types and passing in appropriate record and merge functions.

**Case Study: The IFDS & IDE Program Analyses.** The Interprocedural Finite Distributive Subset (IFDS) algorithm [Reps et al. 1995] solves context-sensitive interprocedural dataflow analysis problems by computing reachability in a graph. The input to the algorithm is a graph called the exploded supergraph, and the algorithm works by constructing two sets of additional edges, called path edges and summary edges, according to rules about existing edges. For example, whenever there is a path edge from node \( n_1 \) to node \( n_2 \) and a supergraph edge from node \( n_2 \) to node \( n_3 \), the algorithm adds a new path edge from \( n_1 \) to \( n_3 \). There are additional rules, some of them more complicated, but they all have a similar form. Thus, it appears natural to express the rules of the IFDS algorithm as Datalog rules and to use a Datalog solver to compute the solution.
However, the exploded supergraph is generally very large and the IFDS algorithm explores only a subgraph that is reachable from the entry point of the program. It would be expensive to realize the whole exploded supergraph as a data structure, including unreachable parts of the graph. Instead, the graph is usually specified implicitly as a successor function that can compute, for each node $n$ that is found to be reachable, the set of edges leading out from $n$ to its successor nodes. The exploded supergraph would be too large to realize as a Datalog relation, so it is impractical to implement the IFDS algorithm in pure Datalog. In our hybrid language, the exploded supergraph can be expressed using functions in the functional fragment of the language, which can be called on demand by the rules of the IFDS algorithm expressed declaratively in the relational fragment of the language. Since the functions can be first-class this allows us to implement a generic IFDS framework once, with parameters that can later be instantiated to specific program analyses.

The Interprocedural Distributive Environment (IDE) algorithm [Sagiv et al. 1996] extends IFDS by adding representations of functions on a lattice to the edges of the exploded supergraph. Whenever a rule of the algorithm creates a new path edge or summary edge, it decorates the new edge with a new function created by composing functions on the edges that triggered the rule. When the sets of edges are represented as Datalog relations, it is easy to decorate each edge with a function by adding a new attribute to the relation to store the function. However, the rules for creating new edges no longer map to pure Datalog: each rule needs to perform a general computation to determine the function for the new edge from the functions of the existing edges. Our language makes it possible to express this by allowing a Datalog rule to call into the functional fragment of the language, which can express the general computations needed to determine the function for the new edge. As with IFDS, we can implement a single parametric IDE framework and later instantiate it with multiple IDE analyses.

6.4 Case Study: The Doop Program Analysis Framework

Doop\(^2\) is a fast, scalable, context-sensitive, subset-based points-to analysis for Java implemented in Datalog [Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2013, 2014]. Doop models many features of the Java programming language, including reflection and exceptions. The Doop Framework comes with a wide variety of configuration options for tuning the precision and performance of the points-to analysis, e.g. the choice of context sensitivity, heap sensitivity, reflection support, exception support, and a multitude of other options. The implementation uses the C preprocessor to selectively include (or exclude) rules based on the configuration options.

For example, if the FEATHERWEIGHT\_ANALYSIS option is disabled, the C preprocessor excludes the following rule that models static field accesses:

```c
#ifdef FEATHERWEIGHT\_ANALYSIS
StaticFieldPointsTo(?hctx, ?value, ?signature) :-
    ReachableStoreStaticFieldFrom(?from),
    OptStoreStaticField(?signature, ?from),
    VarPointsTo(?hctx, ?value, _, ?from).
#endif
```

As another example, if the REFLECTION option is enabled, the C preprocessor includes the following rule that models reflective calls:

```c
#ifdef REFLECTION
AnyCallGraphEdge(?from, ?to) :- ReflectiveCallGraphEdge(_, ?from, _, ?to).
#endif
```

\(^2\)https://bitbucket.org/yanniss/doop/src/master/
Similarly, the REFLECTION_DYNAMIC_PROXIES option controls the modelling of reflective calls through proxy objects:

```
#ifdef REFLECTION_DYNAMIC_PROXIES
AnyCallGraphEdge(?from, ?to) :- ProxyCallGraphEdge(_, ?from, _, ?to).
#endif
```

These examples demonstrate that it is common to selectively include or exclude rules in Doop. In fact, a search across the Doop repository reveals 268 uses of ifdef and 189 uses of ifndef.

An interesting flag is EXCEPTIONS_CS, which controls whether context sensitivity is used when modelling the data flow of exceptions:

```
#ifdef EXCEPTIONS_CS
ThrowPointsTo(?hctx, ?heap, ?ctx, ?method) <-
      Throw(?ref, ?var),
      VarPointsTo(?hctx, ?heap, ?ctx, ?var),
#else
ThrowPointsTo(?hctx, ?heap, ?method) <-
      Throw(?ref, ?var),
      VarPointsTo(?hctx, ?heap, _, ?var),
#endif
```

Note that in the first rule, the ThrowPointsTo predicate symbol takes four arguments whereas in the second rule it takes only three arguments. Thus it is critical, whether ThrowPointsTo is enabled or not, that any use of ThrowPointsTo is also guarded by EXCEPTIONS_CS to ensure that the arities always match up. Another example of the same pattern is:

```
#ifdef EXCEPTIONS_CS
ThrowPointsTo(?hctx, ?heap, ?callerCtx, ?callerMethod) <-
      CallGraphEdge(?callerCtx, ?invocation, ?calleeCtx, ?tarmethod),
      ThrowPointsTo(?hctx, ?heap, ?calleeCtx, ?tarmethod),
#else
ThrowPointsTo(?hctx, ?heap, ?callerMethod) <-
      CallGraphEdge(_, ?invocation, _, ?tarmethod),
      ThrowPointsTo(?hctx, ?heap, ?tarmethod),
#endif
```

These examples illustrate that there is a need to selectively include or exclude certain rules where even the arity of the predicate symbols might differ. The Flix type system ensures that for any first-class constraint set constructed at run-time the arities of predicate symbols always match.

Doop uses macros to parameterize the choice of context- and heap sensitivity:

```
#define MergeMacro(callerCtx, invocation, hctx, value, calleeCtx) \
  Context(calleeCtx), \ 
  CtxFromRealCtx[RealContext2FromContext[calleeCtx], invocation] = calleeCtx
#define MergeMacro(callerCtx, invocation, hctx, heap, calleeCtx) \
  Context(calleeCtx), \ 
  CtxFromRealCtx[RealHContext1FromHContext[hctx], \ 
    DeclaringClass:Allocation[RealHContext2FromHContext[hctx]], \ 
    heap] = calleeCtx
```

// .. and many more ...

Here the syntax $A[x] = y$ can be understood as a function call that is true if $A(x)$ evaluates to $y$.

The two macro definitions above are examples of different context sensitivity policies. Similarly, there are different definitions of the `MergeStartupMacro` macro:

```c
#define MergeStartupMacro(hctx, heap, calleeCtx)
    Context(calleeCtx),
    ContextFromRealContext[RealHContextFromHContext[hctx], heap] = calleeCtx
```

```c
#define MergeStartupMacro(hctx, heap, calleeCtx)
    Context(calleeCtx),
    ContextFromRealContext[heap, heap] = calleeCtx
```

and of the `MergeThreadStartMacro` macro:

```c
#define MergeThreadStartMacro(hctx, heap, callerCtx, newCtx)
    Context(newCtx),
    ContextFromRealContext[RealHContextFromHContext[hctx], heap] = newCtx
```

```c
#define MergeThreadStartMacro(hctx, heap, callerCtx, newCtx)
    Context(newCtx),
    ContextFromRealContext[heap] = newCtx
```

In Flix, we can express such macros as arguments to higher-order functions that return constraint sets parameterized by these functions, as seen in Example II. The `commonMacros` file contains 11 such macros and the repository has over 800 uses of `#define`.

The Doop Framework demonstrates that there is a need for parametricity of Datalog programs. However, there are at least three problems with a meta-programming approach based on textual generation: (i) we cannot be sure that all of the programs output by the C preprocessor system are valid Datalog programs, (ii) we cannot change any option at runtime; we have to recompile the entire program from scratch, and (iii) we cannot determine if the program is stratified without actually constructing a specific program for a specific configuration. The $\lambda_{Dat}$ calculus (and its implementation in Flix) overcomes these issues. We can use first-class Datalog constraints to model the selective inclusion of rules. We can model macro functions, such as `MergeMacro`, using ordinary functions. And finally, the type system of $\lambda_{Dat}$ ensures that any Datalog program constructed at run-time is well-formed and stratified.

7 RELATED WORK

**Datalog Integration.** Arntzenius and Krishnaswami [2016] present Datafun, a typed functional programming language with constructs for fixpoint computations. The key feature of Datafun is to track monotonicity with types. If a function $f$ is deemed monotone by the type system, this guarantees that the fixpoint of $f$ exists and can be computed by the `fix` operator. Datafun is closer to a functional language than $\lambda_{Dat}$. In Datafun, the programmer writes functions and computes with these, whereas in $\lambda_{Dat}$, the programmer writes constraints, composes them, and solves them. The integration in Datafun is tight: functions such as `map` and `filter` are given monotonicity types. It is also expressive: the cross product and transitive closure can be expressed as generic functions. However, Datafun programs are not solvable by standard techniques, such as semi-naive evaluation. Consequently, the extra power of Datafun comes at a cost. Most recently Arntzenius and Krishnaswami [2019] have studied semi-naive evaluation in the context of Datafun.

Madsen et al. [2016] present Flix, a programming language that extends the Datalog semantics from *constraints on relations* to *constraint on lattices*. Unlike Datalog relations, which are always finite, lattices may have an infinite number of elements. Termination is still ensured, provided that the lattices have finite height. To define the components of the lattices, e.g. the least upper bound and greatest lower bound, Flix allows Datalog predicates to refer to functions defined in a functional language. While this allows the logic part of Flix to refer to the functional part of Flix, the opposite direction is not possible. In the current work, $\lambda_{\text{DAT}}$ allows integration in both ways: Datalog constraints can refer to expressions and expressions can evaluate to constraints.

**Template Programming.** Programmatic generation of Datalog programs is commonplace, whether based on simple string concatenation or with the use of a macro pre-processors. Recently, several macro-based meta programming languages for Datalog have appeared.

Souffle is an efficient and scalable Datalog engine that comes with its own template programming language [Scholz et al. 2016; Souffle Authors 2018]. In Souffle, a Datalog program can be organized as a set of *components* which are collections of Datalog constraints. A component can be instantiated, which copies all constraints and predicates within it, giving fresh names to all its predicate symbols. The fresh predicate symbols are then accessible through a handle to the instance. Components also support a simple form of inheritance that allows reuse of constraints and gives the ability to override (i.e. remove) constraints from a super-component. In $\lambda_{\text{DAT}}$, the combination of first-class constraints and predicate symbols allows us to emulate components, if so desired.

Template programming is a powerful technique, but it has at least two downsides: (i) programming with templates is difficult and error-prone, and (ii) template expansion can only depend on information that is available at compile time.

**Logic Programming.** Mercury is a strongly typed functional and logic programming language [Henderson et al. 1996; Somogyi et al. 1996, 1995]. In Mercury, a predicate has a *mode* that determines which arguments must explicitly be passed to it and how many times the predicate can succeed, i.e. whether it is deterministic or non-deterministic. This enables a mix of functional and logic programming: A function requires all its parameters and returns a deterministic result, whereas a Prolog-style goal requires only some of its arguments and may return a non-deterministic result. In $\lambda_{\text{DAT}}$, the integration between functions and predicates is less tight: We are free to call functions inside Datalog constraints, but we cannot call a Datalog predicate as an expression. Instead, we must put that predicate into a Datalog program and explicitly solve it. In comparison to Mercury, $\lambda_{\text{DAT}}$ is closer to a meta-language for Datalog.

**Constraint Logic Programming.** Constraint Logic Programming (CLP) extends logic programming with a decidable background theory, such as lists, trees, or linear arithmetic [Cohen 1990; Jaffar and Lassez 1987; Jaffar and Maher 1994; Li and Mitchell 2003]. A CLP(X) program is a set of Horn clauses with formulae over the background theory X. During evaluation, term unification is augmented with a decision procedure for the underlying theory, e.g. an SMT solver or other specialized solver.

**Datalog Extensions.** Many Datalog extensions have been proposed. We detail some of them below.

Alvaro et al. [2010] present Dedalus, an extension of Datalog with time. In Dedalus, every fact is equipped with a timestamp $T$ and said to hold at that instant. Dedalus rules come in two types. A *deductive* rule derives a new fact at timestamp $T$ from facts already established at timestamp $T$. An *inductive* rule, on the other hand, derives a new fact at timestamp $T + 1$ from facts already established at timestamp $T$. An important use case for Dedalus is to describe distributed systems.

Alvaro et al. [2011] present the Bloom programming language, a distributed programming language based on Datalog and built on the ideas of Dedalus. A Bloom program is a collection of rules that operate on facts with timestamps. Bloom programs are re-evaluated whenever new
messages arrive, i.e. over the network or due to some local event (such as a timeout). In later work, Conway et al. [2012] extend Bloom with predicate symbols that are given a lattice interpretation, similar to the first work on Flix [Madsen et al. 2016].

Avgustinov et al. [2016] present QL, a programming language that combines logic programming with elements of object-oriented programming. QL supports classes and methods, but recast in logical terms. For example, a class is simply a set of values described by a collection of logic formulae. Sub-typing between classes is then logical implication between sets. While the QL language is rich in features, it still compiles to plain Datalog and is solvable using standard Datalog techniques.

Bembenek and Chong [2018] present FormuLog, an extension of Datalog with logical formulae. FormuLog permits terms to be constructed from boolean connectives and using functions from some underlying theory. Using FormuLog, it becomes possible to express program analyses such as symbolic execution and model checking. To solve FormuLog programs, an SMT solver is used to reason about the specific logical formulae.

While all of these extensions increase the power of Datalog, the primary contribution of the current work is an embedding of Datalog as first-class values within a functional programming language. The extension of the expressive power of Datalog by allowing expressions as head terms is a secondary contribution. In addition, we can imagine situations where one would want the expressive power of Dedalus or FormuLog combined with first-class constraints.

Type Systems for Datalog. Zook et al. [2009] present a type system for LogicBlox, a Datalog-based platform for enterprise planning. The type system is based on the notion of integrity constraints. An integrity constraint is a special form of rule that if ever instantiated indicates an error. For example, an integrity constraint might demand that every Child(x) is also a Person(x): if there is a child that is not a person then there is a violation. Such integrity constraints can be viewed as specifying a type system. It is straightforward to check such constraints at run-time, i.e. during the fixpoint computation, and to raise an error, corresponding to a form of dynamic type checking. The authors present a static type system that can eliminate many of these run-time checks.

Schäfer and de Moor [2010] propose a type system for statically checking integrity constraints in the style of Zook et al., but for a richer language of type constraints, and with a type inference algorithm that is sound and optimal.

The work of Zook et al. [2009] and Schäfer and de Moor [2010] is orthogonal to our work: We embed Datalog programs inside a functional programming language and we want to ensure that composition of such programs is well-typed w.r.t. the arity and type of the terms. We can imagine an extension of our type system that also takes integrity constraints into account. We leave it as interesting future work to explore this direction.

8 CONCLUSION

We have proposed the idea of first-class Datalog constraints to enable the construction, composition, and evaluation of Datalog programs within a functional language.

We have proposed a modular type system, based on Hindley-Milner, in which reusable fragments of Datalog programs can be typed independently while guaranteeing that their composition is type-safe. The type system allows reuse and abstraction via polymorphism. We have proven safety of the system. We have also proposed a sound technique for computing stratification of \( \lambda_{\text{Datalog}} \) programs at compile-time. The technique ensures that every Datalog program constructed at run-time is stratified. Our implementation is freely available as as part of the Flix programming language.

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada.
REFERENCES


Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What You Always Wanted to Know About Datalog (and Never Dared to Ask). IEEE Transactions on Knowledge and Data Engineering (TKDE) (1989). https://doi.org/10.1109/69.43410


