Flix: A Design for Language-Integrated Datalog

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

We present a comprehensive overview of the Datalog facilities in the Flix programming language. We show how programmers can write functions implemented as Datalog programs and we demonstrate how to build modular and reusable families of Datalog programs using first-class Datalog program values, rho abstraction, parametric polymorphism, and type classes. We describe several features that improve the ergonomics, flexibility, and expressive power of Datalog programming in Flix, including the inject and query program constructs, head and guard expressions, functional predicates, lattice semantics, and more.

We illustrate Datalog programming in Flix with several applications, including implementations of Ullman's algorithm to stratify Datalog programs, the Ford-Fulkerson algorithm for maximum flow, and the IFDS and IDE algorithms for context-sensitive program analysis. The implementations of IFDS and IDE fulfill a long-term goal: to have fully modular, polymorphic, typed, and declarative formulations of these algorithms that can be instantiated with any abstract domain.

CCS Concepts: • **Software and its engineering** → **Language features**.

Additional Key Words and Phrases: Datalog, logic programming, language design, Flix programming language

ACM Reference Format:
Magnus Madsen and Ondřej Lhoták. 2025. Flix: A Design for Language-Integrated Datalog. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 348 (October 2025), 29 pages. https://doi.org/10.1145/3763126

1 Introduction

"A programming language that doesn't affect the way you think about programming, is not worth knowing."

— *Alan Perlis*

Datalog is an elegant and powerful logic programming language that empowers programmers to write declarative logic constraints on relations and lattices and have them efficiently solved. Datalog is experiencing a renaissance in research and industry. Research on Datalog is blossoming in at least four directions: (i) new and emerging Datalog applications [Backes et al. 2019; Grech et al. 2019; Seo 2018], (ii) extensions to Datalog semantics [Alvaro et al. 2011; Bembenek et al. 2020; Madsen et al. 2016], (iii) techniques for debugging and provenance [Deutch et al. 2014; Pacak and Erdweg 2023; Zhao et al. 2020], and (iv) incremental evaluation and performance [Jordan et al. 2019b; Ryzhyk and Budiu 2019; Sahebolamri et al. 2023; Sun et al. 2023; Szabó et al. 2018, 2016].

In this paper, we focus on a fifth direction: bringing Datalog to the masses. We focus on *ergonomics* first and foremost. We believe that for Datalog to gain broader adoption, it must be integrated into general-purpose programming languages so that programmers can use it where it really shines: to declaratively express and solve fixpoint problems. In this line of work, other notable systems are Ascent [Sahebolamri et al. 2022], Datafrog [McSherry 2018], Datafun [Arntzenius and Krishnaswami 2016], and Functional incA [Pacak and Erdweg 2022]. We discuss them in Section 8.

Authors' Contact Information: Magnus Madsen, magnusm@cs.au.dk, Aarhus University, Aarhus, Denmark; Ondřej Lhoták, olhotak@uwaterloo.ca, University of Waterloo, Waterloo, Canada.

ACM 2475-1421/2025/10-ART348
https://doi.org/10.1145/3763126

This paper presents a "Grand Tour" of the Datalog facilities in the Flix programming language. As a ***programming language systems*** paper, it aims to explore all practical aspects required to build a useable programming language with Datalog as a first-class feature. It is the culmination of several years of design, development, and experience.

While bits and pieces of Flix have been described in the research literature as small calculi and type systems, in this paper, we take a step back and explain the entire system in one coherent narrative. In particular, we describe each Datalog program construct and how it increases the ergonomics, flexibility, or expressive power of Datalog programming in Flix, including:

- **(Datalog Program Values)** We describe how Flix supports Datalog programs as a *first-class values* which can be stored in local variables, passed as function arguments, returned from functions, composed with other Datalog values, and have their minimal model computed.
- **(Inject-Query Constructs)** We describe two programming constructs for working with Datalog program values. The `inject` construct allows programmers to convert any immutable data structure (e.g., a `List[t]` or `Set[t]`) to a Datalog relation. The `query` construct, inspired by Linq's language-integrated query [Meijer et al. 2006], allows programmers to compose a sequence of Datalog program values, compute their minimal model, and extract a relation as a collection of tuples. We illustrate how these constructs give rise to a common programming pattern, which we dub the *inject-program-query* pattern.
- **(Polymorphism and Type Classes)** We show how parametric polymorphism and type classes enable programmers to write reusable Datalog program values which abstract over the concrete types of their Datalog terms.
- **(Rho Abstraction)** We describe a mechanism, called *rho abstraction*, to construct modular Datalog program values with *local* predicate symbols hidden from the outside world.
- **(Functional Predicates)** We describe a mechanism, called *functional predicates*, which enables functions in the functional world to introduce *tuples* into the Datalog world during fixpoint computation.
- **(Lattice Semantics)** We describe how Flix supports not just *constraints on relations* as in classical Datalog, but also *constraints on lattices*. We illustrate how lattices are defined as type classes and show how the `fix` construct enables use of lattice values in relations.
- **(Provenance)** We describe how lattice semantics can be used to compute a limited form of provenance. In essence, each Datalog rule can be extended to capture information about the facts that it derives. While this "poor man's" provenance does not grant access to the full provenance proof, we argue that it is nevertheless useful.

We also touch on a few other Datalog aspects, including stratified negation [Apt and Bol 1994; Starup et al. 2023; Zaniolo et al. 1993].

The outcome is a ***cohesive programming language system*** that enables the ***construction of modular and reusable families of Datalog programs*** with flexibility across multiple dimensions, including: composition, encapsulation, polymorphism, and tight integration with type classes. Most importantly, Datalog is integrated smoothly into the general-purpose programming language, enabling a mix of programming styles. Lastly, but importantly, the system ensures traditional type safety and guarantees that every Datalog program constructed at runtime is stratified.

Applications. We present several applications that illustrate Datalog programming in Flix: We implement Ullman's algorithm for the stratification of Datalog programs [Ullman 1988], the Ford-Fulkerson algorithm for maximum flow problems [Cormen et al. 2022], the IFDS and IDE algorithms for context-sensitive program analysis [Reps et al. 1995; Sagiv et al. 1996], and more. The implementations of IFDS and IDE fulfill a long-term goal: to have fully modular, polymorphic, typed, and declarative formulations of these algorithms, which can be instantiated with any abstract domain.

Language Design. We explore several programming language design choices made during the development of Flix. In particular, we consider the trade-offs of using row typing for Datalog values and expressions. We then discuss the roles of the `solve` and `query` constructs and explain why both are necessary. Next, we explore how Datalog expressions interact with lexical scope. We also briefly highlight some of the compile-time checks performed by the Flix compiler, including the range restriction and stratification. Finally, we present alternative design choices that may be of interest to implementers of similar systems.

Datalog Solver. The Flix Datalog solver is implemented as a just-in-time compiler that translates Datalog programs into a low-level intermediate representation known as the Relational Algebra Machine (RAM). The RAM IR is an imperative language equipped with relational algebra operators. The translation from Datalog to RAM elaborates the semi-naïve evaluation strategy. The RAM IR is subjected to a series of optimization passes, including the introduction of parallelism, before being executed by an interpreter. Flix represents relations and lattices using concurrent B+-trees which support efficient range queries. Although the Flix Datalog engine does not yet match the performance of the state-of-the-art Soufflé engine [Scholz et al. 2016], it implements many of the same techniques and its performance and scalability is practical for real-world use.

Teaching. We have been using Flix for teaching in two courses at the undergraduate and graduate levels at Aarhus University. Specifically, we have used Flix in an extracurricular course on logic programming for talented 1st year students. This course uses Datalog and Flix to illustrate the declarative logic programming paradigm. We have also used Flix as part of a 4th year master-level course on program analysis. Here, we use Flix to implement several program analyses, including dataflow and points-to analyses. These two courses have offered valuable insight into how beginners as well as experienced programmers use Flix, which has influenced the design of the language.

Audience and Objective. We intend for this paper to be accessible to a broad audience. The papers that describe Datalog enriched with lattice semantics [Madsen et al. 2016], first-class Datalog program values [Madsen and Lhoták 2020], and their stratification [Starup et al. 2023] are all technical in nature. Their focus is on syntax, semantics, and types. While such foundations are essential, this paper aims to take a step back, reorient its focus toward the programmer, and put everything into a larger context. ***We want to illustrate that a language design that combines functional programming with Datalog is powerful and elegant.***

We remark that *everything in the paper has been implemented and that all examples are runnable.* We want to emphasize that Flix is a full-blown programming language with a large standard library, package manager, and excellent Visual Studio Code and LSP support.

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

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

A fully functional and reusable artifact is available at: https://doi.org/10.5281/zenodo.15743442

Organization. The paper is organized as follows: Section 2 provides background information on Datalog and briefly explains the history of the Flix programming language. Section 3, which is the main body of the paper, describes programming with Datalog in Flix. We discuss all the outlined features and motivate them with examples. We illustrate how they help bridge the gap between the functional and the Datalog worlds. We also discuss some programming patterns that we have discovered. Section 4 discusses several design choices made in Flix. Section 5 presents the Flix Datalog solver. Section 6 presents several applications illustrating Datalog programming in Flix. Section 7 briefly discusses our experience with using Flix in teaching two courses. Lastly, Section 8 presents related work and Section 9 concludes.

2 Background

We begin with a brief introduction to Datalog and to the history of the Flix programming language.

2.1 A Brief Introduction to Datalog

A Datalog *program* P is a collection of constraints $C_1, \cdots, C_n$. A *constraint*, also called a Horn clause, 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 it can be negated, written with the not keyword before 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 a variable x or a literal constant c.

Every Datalog program has a unique solution called the *minimal model*. Informally, the minimal model is the smallest set of facts that can be derived from the constraints of the program. Importantly, the minimal model is defined independently of how it is computed. This 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 the result. Here is a small example of a Datalog program:

```
Edge(1, 2). Edge(2, 3). Edge(3, 4).
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
```

This Datalog program contains three `Edge` facts and two rules for inferring `Path` facts. Notably, the last rule is recursive: the predicate `Path` depends on itself. The minimal model of the above program contains nine facts, including the `Path(1, 4)` fact which captures that there is a path from 1 to 4.

For more information on Datalog, we recommend [Ceri et al. 1989] and [Greco and Molinaro 2016].

2.2 A Brief History of Flix: From Datalog Dialect to Full-Blown Programming Language

Flix was originally introduced as a Datalog dialect which enriched Datalog from *constraints on relations* to *constraints on lattices*. Programmers would write a *single* Datalog program, which could refer to lattice operations defined in a small functional language [Madsen et al. 2016]. The original work used Flix to present concise and declarative versions of the IFDS and IDE algorithms.

Allowing programmers to define their own lattices and monotone functions meant they might make mistakes breaking the properties needed to ensure the existence of a least fixed-point. To ensure safety, subsequent work proposed using dynamic symbolic execution and SMT solving to verify the required lattice and monotonicity properties [Madsen and Lhoták 2018].

A significant step forward was the introduction of *first-class* constraints where Datalog programs became first-class values [Madsen and Lhoták 2020]. With this change, the roles of Datalog and the functional language were reversed: Datalog was now deeply embedded inside the functional language, not vice versa. Instead of writing a single Datalog program, programmers would now write a functional program with multiple Datalog fragments [Madsen et al. 2022].

Introducing first-class Datalog constraints presented the challenge that stratification was no longer a simple syntactic check. To overcome this problem, Flix adopted a lightweight context– and flow-insensitive whole program analysis to ensure safety [Starup et al. 2023].

While the Datalog aspect of Flix grew, other aspects of the Flix programming language grew even more. A new strand of research became its polymorphic type and effect system [Lutze et al. 2023; Madsen and van de Pol 2020]. Another strand became its use of type-level Booleans and Boolean unification [Madsen et al. 2023b] to support relational nullability [Madsen and van de Pol 2021] and restrictable variants [Madsen et al. 2023a].

Today, Flix is a modern, fully-featured programming language with two notable features: A polymorphic type and effect system and first-class Datalog program values.

3 Datalog Programming in Flix

We now present a comprehensive overview of Datalog programming in Flix. We have written a single self-contained narrative that progresses from simple uses of Datalog to the more complex.

3.1 A Brief Overview of Flix

Flix is a functional, imperative, and logic programming language that supports algebraic data types, pattern matching, extensible records, higher-order functions, parametric polymorphism, type classes, higher-kinded types, associated types and effects, user-defined effects and handlers, structured concurrency with channels and light-weight processes, and has a Hindley-Milner style polymorphic type and effect system. Flix comes with a package manager with Maven integration and a fully-featured Visual Studio Code extension. The Flix compiler project, including the standard library and tests, is approximately 251,000 lines of code.

We remind the reader that Flix is open source, ready for use, and freely available at:

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

A fully functional and reusable artifact is available at: https://doi.org/10.5281/zenodo.15743442

3.2 Datalog Programs as First-class Values

In Flix, Datalog programs are *first-class values*: they can be stored in local variables, passed as arguments to functions, returned from functions, and stored in data structures. We can compose two Datalog program values to compute their union. We can compute the minimal model of a Datalog value which is itself a Datalog value. For example, we can write:

```
let db = #{
    Edge(1, 2). Edge(2, 3). Edge(3, 4).
};
let pr = #{
    Path(x, y) :- Edge(x, y).
    Path(x, z) :- Path(x, y), Edge(y, z).
};
let mm = solve (db <+> pr)
```

Here we have three local variables: `db`, `pr`, and `mm`. Each local variable holds a Datalog program value. The `mm` variable evaluates to the minimal model of the Datalog expression `db <+> pr` which is the union of the two Datalog values `db` and `pr`. In general, Datalog values contain both facts and rules, but no Datalog evaluation happens until `solve` (or `query`) is called. Datalog values and expressions are typed with row types [Madsen and Lhoták 2020].

For example, the type of `db` is:

$$\forall \rho.\{\mathsf{Edge} = (\mathsf{Int32}, \mathsf{Int32}) \mid \rho\}$$

and the type of `pr` is:

$$\forall \alpha, \rho.\{\mathsf{Path} = (\alpha, \alpha), \mathsf{Edge} = (\alpha, \alpha) \mid \rho\}$$

and the type of `mm` is:

$$\forall \rho.\{\mathsf{Edge} = (\mathsf{Int32}, \mathsf{Int32}), \mathsf{Path} = (\mathsf{Int32}, \mathsf{Int32}) \mid \rho\}$$

The Flix type system ensures that the arity and terms of each predicate symbol are consistent. The type system is sound, i.e. it satisfies progress and preservation [Madsen and Lhoták 2020].

3.3 Injecting Facts and Querying the Minimal Model

We now present two programming constructs for getting facts into and out of the Datalog world. In the original Flix implementation, Datalog facts could only be constructed by capturing variables from the lexical scope. Worse, facts could not be extracted back into the functional world [Madsen and Lhoták 2020]. We overcome both issues with the `inject` and `query` constructs.

3.3.1 Injecting Facts. Flix, like most functional programming languages, support a rich collection of immutable data structures such as `List`s, `Set`s, and `Map`s, but also `Option`s, `Result`s, `Validation`s, `Chain`s, `Nel`s (non-empty lists), `Nec`s (non-empty chains), and several more. Programming with these data structures is the bread and butter of functional programming. In contrast, Datalog has a single data structure: predicates, i.e., named relations. What we need is a mechanism to bridge the gap between the two worlds. Specifically, we need a feature to associate a collection of tuples with a predicate symbol and to transform such a collection into an internal representation that is suitable for Datalog evaluation. In Flix, we overcome both issues with the `inject` construct.

We can translate a list of tuples:

```
let edges = (1, 2) :: (2, 3) :: (3, 4) :: Nil
```

into a Datalog relation, i.e. a set of facts, using `inject`:

```
inject edges into Edge/2
```

which conceptually evaluates to the Datalog program value[1]:

```
#{ Edge(1, 2). Edge(2, 3). Edge(3, 4). }
```

The type of the `inject` expression is, as one would expect, the row type: $\{\mathsf{Edge}(\mathsf{Int32}, \mathsf{Int32}) \mid r\}$. We can use `inject` to translate multiple heterogeneous collections into relations:

```
let nodes = Set#{1, 2, 3, 4};
let edges = (1, 2) :: (2, 3) :: (3, 4) :: Nil;
inject nodes, edges into Node/1, Edge/2
```

which conceptually evaluates to:

```
#{ Node(1). Node(2). Node(3). Node(4). Edge(1, 2). Edge(2, 3). Edge(3, 4). }
```

The general form of `inject` is:

```
  inject e1, e2, ... into P1/a1, P2/a2, ...
```

where e_i is an expression and P_i/a_i is a predicate symbol with its arity.[2] The `inject` construct works with any data type that implements the `Foldable[t]` type class. In particular, it works for `List[t]`, `Set[t]` and `Map[k, v]`, and can be extended to any programmer-defined data type.

3.3.2 Querying the Minimal Model. We can construct Datalog program values, but how do we solve them? Flix supports the `query` construct to compute the minimal model of a Datalog program value and to extract a collection of tuples from it. Given the Datalog program values:

```
let db = #{ Edge(1, 2). Edge(2, 3). Edge(3, 4). };
let pr = #{
    Path(x, y) :- Edge(x, y).
    Path(x, z) :- Path(x, y), Edge(y, z).
 };
```

[1]An `inject` expression does not have to immediately evaluate to a set of Datalog facts. The Flix Datalog solver can defer this transformation until later, given that the input is immutable. In particular, the Flix implementation is free to choose how to represent the facts in the Edge relation.
[2]A list of pairs – e.g. `List[(Int32, Int32)]` – can be interpreted in two ways: (a) as a unary relation whose elements are pairs, or (b) as a binary relation. To avoid ambiguity, `inject` requires the programmer to specify the expected arity.

We can write:

```
query db, pr select (x, y) from Path(x, y)
```

to compute the minimal model of the composition of the Datalog values `db` and `pr`, and to extract all pairs (x, y) from the `Path` relation. Here the type of the `query` expression is `Vector[(Int32, Int32)]`. In Flix, `Vector[t]` is the type of immutable arrays, i.e. a space-efficient data structure with its elements laid out consecutively in memory.

The `query` construct is implemented by a call to a full-blown Datalog solver, implemented in Flix, based on semi-naïve evaluation [Greco and Molinaro 2016]. The Datalog solver is implemented as a JIT compiler from Datalog program values, with extensions, to the RAM IR [Scholz et al. 2016]. The translation embodies the semi-naïve evaluation strategy. An interpreter, also written in Flix, evaluates the RAM IR to compute the minimal model, and then the result relation is extracted as a collection of tuples. In Section 5, we discuss the Flix Datalog solver in greater detail.

Returning to `query`, its general form is:

```
query e, ... select (e, ...) from P(e, ...), ... [where e]
```

If we squint, we can see that `query` is very similar to a Datalog rule, except the head atom has no associated predicate symbol. Since the return type is a `Vector[t]`, it is easy to perform various post-processing steps that are best done outside of Datalog. Here are a few common examples:

```
query db, pr select (x, y) from Path(x, y) |> Vector.isEmpty
query db, pr select (x, y) from Path(x, y) |> Vector.sort
query db, pr select (x, y) from Path(x, y) |> Vector.toMap
query db, pr select (x, y) from Path(x, y) |> Vector.foreach(println)
```

Here we take advantage of Flix's support for pipelines and partial application.[3] Using `toMap` is common since working with a map is more pleasant than working with a vector of tuples.

We remark that in the context of the Flix type and effect system, `inject` and `query` are *pure*. In particular, the evaluation of a Datalog value with `query` is guaranteed to produce a deterministic unique result: the minimal model. Hence, we can use `query` to implement pure functions.

3.3.3 *Inject-Program-Query.*

We can use `inject` and `query` to write a function that computes the transitive closure of a graph where the graph is represented as a list of edges:

```
def closure(edges: List[(Int32, Int32)]): Vector[(Int32, Int32)] =
    let db = inject edges into Edge/2;
    let pr = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
    };
    query db, pr select (x, y) from Path(x, y)
```

This function illustrates the power of Flix. We have implemented the closure *function* as a short, concise, and elegant *Datalog program*. To the outside, `closure` is a pure function like any other, but on the inside, it takes advantage of Datalog where it really shines: to express and solve fixpoint constraints on relations. Moreover, the `closure` function is likely shorter and faster than any transitive closure computation we could have written by hand.

The `closure` function illustrates a common programming pattern: We write a function that takes some immutable data structure, injects that data into a collection of Datalog relations, performs some complex computation in Datalog, and returns the result as an immutable data structure. We dub this style of programming the *inject-program-query* pattern.

[3]The pipeline `|>` operator is simply function application written in reverse. For example, `x |> f` is `f(x)`.

3.4 Polymorphism

A limitation of the `closure` function is that it only works for edges where the vertices are integers. What if we have a graph where the vertices are strings? Fortunately, Flix supports parametric polymorphism and type classes, which we can take advantage of. Without any change to the function body of `closure`, we can update its signature to:

```
def closure(edges: List[(t, t)]): Vector[(t, t)] with Order[t]
```

Two things have changed: (i) we have introduced a polymorphic type parameter `t`[4], and (ii) we have added a type class constraint on `t` for the `Order` type class. We require an `Order[t]` instance because the Datalog solver must be able to build search trees on the term constants[5]. In the original version of `closure`, we did not need the `Order` constraint because `Int32` already implements `Order`. In Flix, most types implement `Order`, and programmers can derive or define their own instances.

3.5 Higher-Kinded Polymorphism

A limitation of the updated `closure` function is that it requires a list of edges. What if we have a set of edges? Fortunately, Flix supports higher-kinded polymorphism. Again, without any changes to the function body of `closure`, we can update its signature to its final form:

```
def closure(edges: f[(t, t)]): Vector[(t, t)] with Foldable[f], Order[t]
```

The `closure` function now works for any higher-kinded type `f[t]` which has a `Foldable` instance. This includes data types such as `List[t]`, `Set[t]`, and `Chain[t]`, but also all their non-empty variants.

We illustrate the usefulness of parametric and higher-kinded polymorphism in Section 6.2 when we present the implementation of a fully generic and modular IDE framework [Sagiv et al. 1996]. For IDE, the primary analysis function is parameterized by *five* type parameters, *four* relations, *six* transfer functions, and *twelve* type class instances.

3.6 Back to First-Class Constraints

We do not use the *inject-program-query* programming pattern when we want to construct modular and reusable Datalog building blocks. For example, we may want to write a polymorphic function that returns a Datalog program value which captures graph reachability:

```
def closure(): #{Edge(t, t), Path(t, t) | r} with Order[t] = #{
    Path(x, y) :- Edge(x, y).
    Path(x, z) :- Path(x, y), Edge(y, z).
}
```

The return type of the `closure` function is an open row type and is polymorphic in the term types of `Edge` and `Path`. Note that we must still require an `Order[t]` instance. We can reuse the closure program to define a larger program that computes cycles:[6]

```
def cycle(): #{Cycle(), Edge(t, t), Path(t, t) | r} with Order[t] =
    let pr1 = closure();
    let pr2 = #{ Cycle() :- Path(x, x). };
    pr1 <+> pr2
```

Here `pr1 <+> pr2` combines the two Datalog program values which simply means to take their union. We can reuse `cycle` in an even bigger Datalog program or we can `query` it with some facts.

[4]In Flix, type parameters are introduced implicitly. For example, the signature `List.map` is declared as `def map(f: a -> b)`.
[5]A Datalog solver based on hashing could require `Hash[t]` instead. A Datalog engine based on both could require both.
[6]Here `Cycle()` is a nullary predicate, i.e. a single fact that is either absent or present.

3.7 Rho Abstraction

We can write a Datalog program value that computes the unreachable vertices in a graph:

```
def unreachable(): #{Vertex(t), Origin(t), Edge(t, t), Reachable(t),
                     Unreachable(t) | r} with Order[t] = #{
    Vertex(x) :- Edge(x, _).
    Vertex(y) :- Edge(_, y).
    Reachable(o) :- Origin(o).
    Reachable(y) :- Reachable(x), Edge(x, y).
    Unreachable(x) :- Vertex(x), not Reachable(x).
}
```

The row type of `unreachable` mentions *all* predicate symbols and their term types used in the Datalog program value: `Vertex(t)`, `Origin(t)`, `Edge(t, t)`, `Reachable(t)`, and `Unreachable(t)`. This presents two problems: First, row types can quickly grow large and unwieldy. Second, row types may expose internal implementation details. To expand on the latter, the predicate symbols `Vertex(t)` and `Reachable(t)` are neither part of the "input" nor the "output", instead they are an implementation detail that we would like to encapsulate. From a compositional view, the input to the above program are the `Origin(t)` and `Edge(t, t)` relations, and the output is the `Unreachable(t)` relation.

What we want is to *hide* the `Vertex(t)` and `Reachable(t)` relations and ensure that they stay local. We can do that with *rho abstraction*. We modify the `unreachable` function to:

```
def unreachable(): #{Origin(t), Edge(t, t), Unreachable(t) | r} with Order[t] =
    #(Origin, Edge, Unreachable) -> #{ // Rho Abstraction
        Vertex(x) :- Edge(x, _).
        Vertex(y) :- Edge(_, y).
        Reachable(o) :- Origin(o).
        Reachable(y) :- Reachable(x), Edge(x, y).
        Unreachable(x) :- Vertex(x), not Reachable(x).
    }
```

Now, the row type of `unreachable` no longer mentions the `Vertex(t)` and `Reachable(t)` predicates.

In general, a rho abstraction is of the form:

```
#(P1, ..., Pn) -> e
```

where `e` should be a Datalog expression. A rho abstraction evaluates `e` to a Datalog value `v` and alpha-renames every predicate symbol in `v` that is *not* one of the listed predicate symbols `P1, ..., Pn`. In the type system, the row type of a rho abstraction only mentions the predicate symbols listed. Suppose we compose the Datalog value returned by `unreachable` with another Datalog value which defines `Vertex` as a binary relation. Normally, this would be a type error, since we cannot have both `Vertex/1` and `Vertex/2` in the same program. However, with rho abstraction, the original `Vertex` predicate symbol has been alpha-renamed to a fresh name and there is no collision.

Alpha-Renaming. The reader may wonder if the alpha-renaming can be done statically, i.e. at compile-time. This is not the case. Consider the program:

```
def f(x: t): #{A(Int32)} with Order[t] = #(A) -> #{ A(21). R(x). }
def g(): #{ A(Int32) } = f(123) <+> f("hello")
```

Suppose we statically rename `R` to `R17`. Now the Datalog value returned by `g` contain two facts with different term types: `R17(123)` and `R17("hello")`! This breaks type safety. Instead, like capture avoidance in the lambda calculus, we must alpha-rename every time a rho abstraction is evaluated. In this case, each invocation of `f` would return a Datalog value with a fresh name for `R`, hence `g` would return a Datalog value with two facts, `R17(123)` and `R18("hello")`, where there is no collision.

3.8 Expressions in Atoms and Guards

We have seen that the functional language embeds Datalog programs as values, but what about the other way around? Flix allows expressions to appear in two places in Datalog programs: As terms in head atoms and as guards in rule bodies.

3.8.1 Expressions in Head Atoms. We can use expressions to compute with integer arithmetic:

```
Generation("Mitochondrial Eve", 0).
Generation(x, n + 1) :- Parent(x, y), Generation(y, n).
```

Here, given some `Parent` relation, we compute the generation of every person going back to Eve. However, we must be careful: if the `Parent` relation has a cycle, the Herbrand base becomes infinite, and the program no longer has a finite model, i.e., evaluation will diverge. Flix programmers may use any expression as a term in a head atom, but they are responsible for ensuring termination. As we shall see in Sections 3.11 and 3.12, expressions in head atoms are often used in combination with lattice semantics.

3.8.2 Expressions as Guards. We can use expressions as guards in rule bodies. For example, we can write a function to compute the transitive closure of a graph subject to some predicate function:

```
def closure(edges: List[(t, l, t)], pred: l -> Bool): Vector[(t, t)] with ... =
    let db = inject edges into Edge/3;
    let pr = #{
        Path(x, l, y) :- Edge(x, l, y), if (pred(l)).
        Path(x, l, z) :- Path(x, l, y), Edge(y, l, z), if (pred(l)).
    };
    query db, pr select (x, y) from Path(x, _, y)
```

The `closure` function takes a labeled graph of `edges` and a predicate `pred`. The predicate determines whether an edge in the graph is active. For example, suppose the graph represents a road network labeled with the forecasted weather. We can compute its transitive closure using only non-icy roads with the expression: `closure(g, weather -> weather != Icy)`.

3.8.3 Purity. Datalog is declarative: the minimal model is specified without reference to any specific algorithm or evaluation order. However, if we allow arbitrary expressions in Datalog rules, including expressions with side effects, we would expose the underlying evaluation order of the Datalog solver. To avoid this, we use the Flix type and effect system to ensure that every head and guard expression is pure, i.e., they cannot have any side effects.[7]

3.9 Functional Predicates

We have seen how `inject` can translate a collection of tuples into a Datalog program value. However, we have found that for some applications, we need the ability to introduce tuples *during the fixpoint computation*. For example, we can define the Fibonacci function:

```
def fibonacci(e: Int32): Vector[(Int32, Int32)] =
    /* efficiently computes all fibonacci numbers from zero up to e */
```

which returns all Fibonacci numbers up to the given `e`. For example, `fibonacci(7)` returns the vector:

```
Vector#{(0, 0), (1, 1), (2, 1), (3, 2), (4, 3), (5, 5), (6, 8), (7, 13)}
```

[7]The Flix type and effect system is rich enough to capture purity, region-based memory, user-defined effects and handlers, and the uninterpretable IO effect. However, it does not capture *termination*. Hence, an expression that diverges could reveal the internal Datalog evaluation order. It is the programmer's responsibility to ensure that all expressions terminate.

we can then use a *functional predicate* to write a Datalog rule that calls the fibonacci function and iterates through all the returned tuples:

```
R(i, f) :- A(x), let (i, f) = fibonacci(x).
```

Importantly, each call to fibonacci generates *a sequence of tuples* which is more efficient than computing every fibonacci number independently. While this example is contrived, the IFDS and IDE algorithms rely heavily on this feature.

3.10 Stratified Negation

The expressiveness of Datalog is significantly increased with the addition of negation [Apt and Bol 1994; Bárány et al. 2012]. However, general negation poses semantic problems. Stratified negation has emerged as a practical and useful form of negation [Zaniolo et al. 1993]. In Flix, as we have already seen, we can use stratified negation to write a function that computes the set of vertices that are unreachable from a given origin:

```
def unreachable(origin: t, edges: List[(t, t)]): Set[t] with Order[t] =
    let db = inject edges into Edge/2;
    let pr = #{
        Vertex(x) :- Edge(x, _).
        Vertex(y) :- Edge(_, y).
        Reachable(origin).
        Reachable(y) :- Reachable(x), Edge(x, y).
        Unreachable(x) :- Vertex(x), not Reachable(x).
    };
    query db, pr select x from Unreachable(x) |> Vector.toSet
```

We inject the edges into the Edge relation. We then define a Datalog program that computes the unreachable vertices as follows: First, we compute all vertices in the graph from the edges. Second, we compute all vertices reachable from the origin. Third and finally, we compute the set of unreachable vertices using negation. Note that we *must* compute the set of vertices because, in the last rule, we must constrain x such that it is *positively bound*. The Flix compiler enforces this so-called *range restriction* at compile-time.

As discussed, unrestricted negation poses semantic issues. For example, the program:

```
Husband(x)  :- Man(x), not Bachelor(x).
Bachelor(x) :- Man(x), not Husband(x).
```

has no unique minimal model. In a nutshell, the problem is that the Husband relation depends negatively on itself. More formally, the dependency graph (sometimes called the *precedence graph*) contains a cycle with a negative edge. Stratified negation restricts negation to those Datalog programs where the dependency graph has no negative cycles. This is sometimes colloquially expressed as "*no negation through recursion*". For a specific Datalog program, it is straightforward to check if it is stratified and to compute its stratification, i.e., an evaluation order where a relation is fully determined before it is used in a negation.

In Flix, the situation is more complex: Datalog programs are values, constructed and composed at runtime. How can we know that the Datalog value passed to query is stratified? We can defer stratification to runtime, but then we would have to accept that evaluation of a Flix program may crash with a "non-stratified error". Instead, Flix adopts a lightweight context– and flow-insensitive whole program analysis enriched with type system information to statically ensure that every Datalog value constructed at runtime is stratified [Starup et al. 2023]. Thus Datalog programs in Flix may use negation, and the Flix compiler ensures that no stratification error can occur.

3.11 Lattice Semantics

We have seen how to compute graph reachability, but what if we wanted to compute single-source shortest distances (SSSD), i.e., the distance to every vertex from some origin? Here, *constraints on relations* are insufficient; we need *constraints on lattices.*

To compute SSSDs, we can define a function:[8]

```
def sssd(origin: t, edges: Set[(t, Int32, t)]): Map[t, Int32] with Order[t] =
    let db = inject edges into Edge/3;
    let pr = #{
        Dist(origin; Down(0)).
        Dist(y; add(d1, d2)) :- Dist(x; d1), Edge(x, d2, y).
    };
    query db, pr select (x , coerce(d)) from Dist(x; d) |> Vector.toMap
```

The `sssd` function takes a graph and an origin vertex. Each edge in the graph is labeled with its distance. The function returns a map with the total distance from the origin to each reachable vertex. The function is implemented as one fact and one rule that both use lattice semantics. Concretely, the use of the semicolon ; signifies that `Dist` is a map lattice.[9]

The fact `Dist(origin; Down(0))` captures that the distance to the origin is *at least* `Down(0)`. We say *at least* because `Down(0)` is a *lower bound* on the value `Dist(origin)`. But since we are interested in *smallest* distances, we use the `Down` constructor, which *reverses* the order on `Int32`. Hence, the lower bound is an upper bound. Thus, as one would expect, the distance to the origin is *at most* zero.

The rule captures that if the distance to `x` is `d1` and there is an edge from `x` to `y` with distance `d2`, then the distance to `y` is at least (i.e. at most) the distance `d1 + d2`. The `add` function has the type `Down[Int32] -> Int32 -> Down[Int32]` and is strict and monotone.

We can understand the program as follows: Initially, the distance to every vertex is conceptually infinite because the smallest value, i.e., the bottom value, of the type `Down[Int32]` is `Int32.maxValue()`. The fixpoint computation iteratively decreases the distance to every reachable vertex by moving upwards in the `Dist` map lattice.

To better understand how joins and meets are used in Flix, consider the following example:

Example 3.1 (Joins and Meets). Assume we have the standard Sign lattice with the elements $\{\bot, \top, \text{Neg}, \text{Zer}, \text{Pos}\}$. Then the Datalog program with lattice semantics:

```
A(1; Pos). B(1; Neg). C(1; Top).
R(1; x) :- A(1; x).             // Intuitively, R(1) is at least A(1).
R(1; x) :- B(1; x).             // Intuitively, R(1) is at least B(1).
R(2; x) :- A(1; x), C(1; x).    // Intuitively, R(2) is at least A(1) meet C(1).
```

has a minimal model where the map lattice `R` has the two mappings: `R(1; Top)` and `R(2; Pos)` and is bottom everywhere else. Specifically, the first two rules force `R(1)` to be at least `Pos` and `Neg`, i.e. we have to compute their join which is `Top`. In the last rule, the variable `x` is bounded by `A(1)` and `C(1)`, i.e. it is the meet of `Pos` and `Top` which is `Pos`. Hence `R(2)` is at least `Pos`.

In both examples, the programmer must be careful and ensure the lattice components actually form a lattice. Moreover, every head and guard expression must be strict and monotone. If these properties are violated, the fixpoint computation may produce wrong results or diverge.

[8]The `coerce` function unboxes a singleton enum, i.e. `coerce(Down(0)) == 0`.
[9]The choice of the semicolon has been controversial. On the one hand, using semicolons means that the syntax retains the same aesthetics. On the other hand, it is easy to overlook a semicolon compared to a comma. However, the Flix type system ensures that *both* predicate symbols and variables are assigned consistent meaning. Hence, in practice, any confusion between relations and lattices will likely result in a type error.

3.12 Polymorphic Lattice Semantics

We can refactor the previous example to take advantage of polymorphism and type classes:

```
def sssd(origin: t, edges: Set[(t, l, t)], sum: (l, l) -> l): Map[t, l]
    with LowerBound[l], JoinLattice[l], UpperBound[l], MeetLattice[l], ... =
    let db = inject edges into Edge/3;
    let pr = #{
        Dist(origin; UpperBound.maxValue()).
        Dist(y; sum(d1, d2)) :- Dist(x; d1), Edge(x, d2, y).
    };
    query db, pr select (x , d) from Dist(x; d) |> Vector.toMap
```

The updated function is identical except for three details: (i) the type parameter `t` must have type class instances for the required lattice components, (ii) the `add` function has been replaced with a function argument `sum` to compute the sum of two distances, and (iii) the distance to the origin is defined as `maxValue` (i.e. the top element) of the lattice on `l`. The upshot is that we can reuse the `sssd` function with several lattices.

3.13 Lattice Stratification

In Flix, every predicate symbol has either a relational or lattice interpretation. The Flix type system ensures that the two cannot be confused. Furthermore, every Datalog variable has a relational or lattice interpretation. The rules for variables are more involved, but the main point is that it would be unsafe to use a lattice variable where a relational value is expected. Doing so could lead to a program with overall infinite lattice height. However, in some situations, we *want* to use a lattice variable in a relation. Fortunately, this can be made safe with *lattice stratification*.

For example, we can compute with Kevin Bacon numbers[10]:

```
def degreesOfKevinBacon(starsWith: List[(String, String)]): Map[Int32, Int32] =
    let db = inject starsWith into StarsWith/2;
    let pr = #{
        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).
    };
    query db, pr select (coerce(n), m) from Count(n, m) |> Vector.toMap
```

Kevin Bacon has the number zero. If an actor has starred in a movie with Kevin Bacon, their number is one. If an actor has starred in movie with an actor who has starred in a movie with Bacon, then their number is at most two, *and so on*. We want to compute the *number of actors with Bacon number n*. We compute the Bacon number of each star in `Degree` using the `Down` lattice, as before. We then compute all actors in layer *n* and finally we compute the size of each layer. Notably, we must have fully computed `Degree` before we compute `Layer`. Similarly, `Layer` must be fully computed before we compute `Count`. The `fix` construct ensures this stratification and moreover permits the *lattice variable n* to be used in a *relational position* in the third rule, and similarly for *s* in the last rule.

3.14 Poor Man's Provenance with Lattice Semantics

We have seen how to compute graph reachability. We have also seen how to compute single-source shortest *distances*. What if we wanted to compute single-source shortest *paths*? For example, if we have calculated that the shortest distance between Paris and Berlin is 1,112 km, how can

[10] https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

we know what roads to follow if we want to drive the route? In Datalog terminology, we are interested in the *provenance* [Deutch et al. 2014; Green et al. 2007; Zhao et al. 2020] of the fact `Dist("Paris", "Berlin", 1112)`, i.e. the proof used to establish this fact. We show how lattice semantics can sometimes be used as a light-weight or "Poor Man's" technique to compute provenance.

Returning to the example of shortest paths, we want to define a lattice of *paths*:

```
enum Path with ToString {
    case Path(List[Int32])
    case Bot
}
```

We define a lattice on `Path` by defining instances for all the required type classes, i.e. instances for `LowerBound`, `PartialOrder`, `JoinLattice`, and so on. We define the bottom element ⊥ as `Bot` which conceptually represents a path of infinite length.

We define the least upper bound as:

```
def leastUpperBound(x: Path, y: Path): Path = match (x, y) {
    case (Bot, p)              => p
    case (p, Bot)              => p
    case (Path(xs), Path(ys)) => if (length(xs) <= length(ys)) x else y
}
```

The partial order can be derived from the least upper bound. Note that two lattice elements are considered equal if their paths have the same length. The other lattice operations are defined accordingly. Next, we define two helper functions:

```
def init(y: Int32, x: Int32): Path = Path(y :: x :: Nil)
```

which constructs a path of length one and

```
def cons(z: Int32, p: Path): Path = match p {
    case Bot      => Bot
    case Path(xs) => Path(z :: xs)
}
```

which extends an existing path by one vertex. We can now put everything together:

```
def sssp(src: Int32, dst: Int32, edges: List[(Int32, Int32)]): Option[Path] =
    let db = inject edges into Edge/2;
    let pr = #{
        Reachable(x, y; init(y, x)) :- Edge(x, y).
        Reachable(x, z; cons(z, p)) :- Reachable(x, y; p), Edge(y, z).
    };
    query db, pr select p from Reachable(src, dst; p) |> Vector.head

def main(): Unit \ IO =
    let edges = List#{(1, 2), (2, 3), (3, 4), (3, 8), (4, 5), (3, 5)};
    println(sssp(1, 5, edges))
```

This program prints `Some(Path(5 :: 3 :: 2 :: 1 :: Nil))` which is the shortest path, in reverse, from `1` to `5`. Extending this program to support different distances on each edge is straightforward: We track the total path length as part of every lattice element.

Minimal Model and Correctness. While the above program works correctly, we must be careful. We have defined two paths as equivalent if they have the same length. For example, the paths `a :: b :: c :: Nil` and `a :: b :: d :: Nil` are equivalent, even though they do not end with the same vertex! Specifically, the *declarative* reading of the constraint:

```
Reachable(x, z; cons(z, p)) :- Reachable(x, y; p), Edge(y, z).
```

allows a situation where the path from `a` to `c` is e.g., `a :: b :: c :: Nil`, but we are allowed to conclude `Reachable(a, c, a :: b :: d :: Nil)` according to the declarative semantics! The problem is that the equivalence relation on paths permits minimal models with "wrong" solutions. However, *in practice*, the Flix Datalog solver will never conjure lattice elements out of thin air, and hence, in this case, will compute actual paths in the graph. If there are multiple shortest paths, it is *implementation defined* which path is returned. If we are uncomfortable with this gap between theory and practice, we *can* define a lattice where we track the *lexicographically smallest* path leading to each vertex. On this lattice, the minimal model uniquely determines the shortest path to each (reachable) vertex.

Light-Weight vs. Full Provenance. In the above example, we extended the predicate `Reachable` from `Reachable(x, y)` to `Reachable(x, y; p)`, where `p` captures the shortest path from `x` to `y`. In general, given a predicate `P(...)`, we can extend it to `P(...; l)` where `l` is an element of a lattice used to track information about the derivation of facts in `P`. We can capture different kinds of information, e.g., shortest distances, shortest paths, rules used in the derivation, and so on. Is this the same as provenance? Yes and no. In full generality, the provenance for a fact `P` gives us the entire deriviation tree – that is, a tree where every node represents the derivation of an intermediate fact, the rule used to derive that fact, and the instantiation of its quantified variables. We cannot readily and ergonomically encode all that information with lattices. However, full provenance is often far too much information. For example, for shortest paths, we do not care about the intermediate facts or which rules were used to derive them. We only want the shortest distance and its path.

3.15 When to use First-Class Datalog Constraints?

An important question is: *when should programmers use first-class Datalog constraints?* We believe that Datalog is not a silver bullet. Flix is primarily an *effect-oriented* functional and imperative programming language; therefore we recommend that programmers mainly write in the functional part of the language. On the other hand, many problems inherently involve fixpoint computations and are natural to express using Datalog.

We propose two criteria which can inform when to use Datalog:

- **(Simplicity)** Programmers should use Datalog if the problem domain is a natural fit, i.e., if the problem is simple and elegant to express using Datalog constraints. The most common use case is for graph queries where functional programming is inelegant or cumbersome.
- **(Performance)** Programmers should use Datalog if (a) performance is important, and (b) the evaluation strategies of Datalog are applicable to the problem domain, i.e., the computation naturally benefits from semi-naïve and parallel evaluation.

3.16 Why not a Library?

A common question is: Why not implement Datalog support as a library in Flix? While almost any programming language feature can be implemented as library, in the case of Datalog, *we would lose*:

- the elegance and conciseness of Datalog syntax. Instead, programmers would have to construct Datalog ASTs by hand which is messy and error-prone.
- well-formedness; in particular, programmers might accidentally construct malformed Datalog programs with unbound variables or other constructs in illegal positions.
- the structural row type system, which ensures that predicate symbols are used with consistent arity and term types and that relations and lattices are not confused.
- the compile-time checks for the range restriction properties.
- the compile-time checks for stratified negation.
- several IDE features, including support for type-aware autocompletion of predicates and automatic renaming of variables and predicate symbols.

Another way to think of the design choice – "library versus language?" – is the following: *If we were to design a programming language in which we could embed Datalog programs as ergonomically as in Flix, what language features would be required?* Viewed through this lens, we can identify the following questions and challenges:

- **(Representation and Typing)** We would need a way to construct data structures that represent Datalog program values. These should be structurally typed, with a type system that supports type inference. We can build on Flix's row typing, but we would also need *first-class labels* [Leijen 2004; Paszke and Xie 2023], since the predicate symbols cannot be known by the library. Depending on how Datalog values are merged, we might also need *row concatenation* [Morris and McKinna 2019], which significantly complicates the implementation of type inference.
- **(Syntax)** With a specific representation in place, the question becomes how to elegantly express Datalog facts and rules. A major challenge is that the Horn clause syntax `A :- B, C.` does not really match the syntax used by imperative, object-oriented, or functional programming languages. In Scheme, an S-expression style syntax might be workable; in other languages, some additional syntactic overhead will almost certainly be required.
- **(Variables)** Another major question is how to represent variables. To a first approximation, variables are simply unique symbols with an associated type that must be used consistently. However, in both Datalog and Flix, Datalog variables are *implicitly introduced*: a universally quantified Datalog variable has no binder and no declaration site—it is simply used. How can a library support that?
- **(Static Checks)** The hypothetical row type system described above would ensure that predicate symbols are used with consistent arity and term types. But how can a library enforce the many other properties guaranteed by Flix, such as the *range restriction* and the *lattice range restriction*? How can it ensure the *positively bound restriction* – that every variable occuring in a negative atom also appears in a positive body atom? And what about stratification – that every Datalog value constructed at runtime has no cycles with a negative edge in its dependency graph?

We believe that deeply integrating Datalog inside a general-purpose programming language, such as Flix, is an essential step towards broader adoption of Datalog, since it enables programmers to use Datalog where it really shines: to declaratively express and solve fixed-point computations on relations and lattices with all the advantages of a real programming language.

4 Language Design

We now discuss several important design choices made during the development of Flix.

4.1 Typing Discipline

We want type safety to ensure that Datalog programs "cannot go wrong" at runtime. Specifically, we must ensure that the arity and term types of every predicate in every Datalog value constructed at runtime are consistent. For example, the same Datalog program value must not contain two atoms `Q(42)` and `Q(1, 2, 3)`, where the arity is mismatched. Similarly, it must not contain two atoms `P(123)` and `P("Hello")`, where the term types are mismatched.

During the development of Flix, we experimented with three typing disciplines:

- **(Nominal Typing)** We require that every predicate symbol is *named*, i.e., globally declared, together with its arity and term types. For example, the declaration `rel Edge(Int32, Int32)` specifies that the `Edge` predicate is a relation with two terms that are `Int32`s. Similarly, the declaration `lat Dist(String, Int32)` specifies that the `Dist` predicate is a lattice where the first term is a `String` and the second term is an `Int32`. Since every predicate is globally declared, the type of Datalog values and expressions are simply a singleton type `Datalog`.

- **(Structural Typing)** We structurally type Datalog values and expressions using row types. A row type assigns arity and term types to every predicate. For example, the row type `({Edge(Int32, Int32), Path(Int32, Int32)})` specifies that the `Edge` predicate is a relation with two terms that are `Int32`s. This is the type system used in Flix today.
- **(Structural Typing with Data Dependencies)** We extend the row typing to include data dependencies. That is, we type Datalog values and expressions using a *pair* of row types where the first row is as above and the second row models the dependency graph. For example, the type: `({Edge(Int32, Int32), Path(Int32, Int32)}, {Path :- Edge})` specifies not only the type of the `Edge` and `Path` predicates, but also that `Path` (positively) depends on `Edge`.

Today, Flix uses *structural typing*. The problem with nominal typing is that it is too limiting in practice: We cannot reuse the same name, e.g. `Edge`, with different arities or types at different points in the program. Moreover, having to globally declare predicate symbols is tedious and goes against the "Datalog spirit" where there are no declarations. Including data dependencies in the type system is theoretically appealing: it allows us to ensure stratification through types. However, it suffers from poor ergonomics. First, dependency graphs can become large, and consequently, the types can also grow significantly. Second, exposing the dependency graph breaks abstraction: it implies that changes to the internals of a Datalog program may also change its type.

4.2 Explicit Constructs for Solving and Querying the Minimal Model

In Flix, `solve` is an explicit construct. An alternative design choice would be that whenever two Datalog values are composed, i.e. `e1 <+> e2`, we immediately compute their fixpoint. That is simpler; now a Datalog value is always fully evaluated. However, when negation is involved, implicitly and eagerly computing the minimal model becomes problematic. With negation, the meaning of `(solve (e1 <+> e2)) <+> e3` is *not* necessarily the same as the meaning of `solve (e1 <+> e2 <+> e3)`, since facts in `e3` may affect rules in `e1` and `e2` that use negation. As a result, *the order of composition would influence semantics*. By making `solve` an explicit construct, we avoid this issue: Datalog programs can be freely composed and we only compute their minimal model when `query` is invoked.

We have established the need for an explicit `solve` construct, but do we need both `query` and `solve`? Or could they be combined? We argue that we need both, as there are cases where we want to solve a program *once* and then query the minimal model *multiple* times. If `query` and `solve` were combined, then every call to `query` would redo the fixpoint computation, which is inefficient.

4.3 Implicit Variable Binders and Integration with Lexical Scope

In Datalog, quantified variables are *implicitly* introduced without explicit binders:

```
Path(x, z) :- Path(x, y), Edge(y, z). // x, y, and z are universally quantified.
```

Flix uses the same syntax, but also adds integration with lexical scope:

```
def reach(o: t): #{Reachable(t), Edge(t, t)} with Order[t] = #{
    Reachable(o). // o is bound by the lexical scope.
    Reachable(y) :- Reachable(x), Edge(x, y). // x and y implicitly quantified.
}
```

We find this to be natural and concise, but it can also lead to bugs:

```
def reach(o: t): #{Reachable(t), Edge(t, t)} with Order[t] = #{
    Reachable(o).
    Reachable(y) :- Reachable(o), Edge(o, y). // Oops-- 'o' is not quantified.
}
```

A more verbose, but less error-prone syntax would require *explicit capture*:

```
def reach(o: t): #{Reachable(t), Edge(t, t)} with Order[t] = #{
    Reachable($o). // We must write $o to capture o from the lexical scope.
    Reachable(y) :- Reachable(x), Edge(x, y).
}
```

An even more explicit syntax would require *explicit capture* and *explicit binders*:

```
def reach(o: t): #{Reachable(t), Edge(t, t)} with Order[t] = #{
    Reachable($o).
    forall(x: t, y: t). Reachable(y) :- Reachable(x), Edge(x, y).
}
```

We think part of the elegance of Datalog is in its beautiful syntax, hence we adopted the light-weight, if potentially error-prone syntax.

5 The Flix Datalog Solver

We briefly discuss how Datalog is implemented in the Flix compiler and runtime.

5.1 Compiler

Like any compiler, the Flix compiler has two responsibilities: (a) to ensure the input program is well-formed and (b) to translate the input program to a lower-level format. With respect to (a), the Flix compiler ensures that (i) every Datalog program expression is well-typed, (ii) every Datalog program value satisfies a collection of *range restrictions*, (iii) every Datalog program value constructed at run-time is stratified, and finally (iv) all required type class instances are present. We refer to Madsen and Lhoták [2020] for more information on the row type system and to Starup et al. [2023] for more details on the range restrictions and stratification. With respect to (b), the Flix compiler lowers all Datalog constructs (e.g. `inject` and `query`) and values (i.e. Datalog literals) to a collection of algebraic data types and function calls. The translation is essentially a desugaring with elements of closure conversion and lambda lifting.

5.2 Runtime

Datalog evaluation begins with a call to `query`, which calls the Flix Datalog solver. The solver is implemented in Flix itself.[11] The solver takes a Datalog program represented as an abstract syntax tree, computes its minimal model, and extracts a specific relation as a collection of tuples.

Compilation to RAM. The Datalog solver is implemented as a just-in-time compiler (JIT) from a high-level Datalog abstract syntax tree to a low-level intermediate representation (IR) called the relational algebra machine (RAM) IR. The RAM IR is optimized and then interpreted. The RAM IR is a low-level imperative language with relational algebra operators. It is inspired by the IR used in Soufflé [Scholz et al. 2016]. The IR is a statement-based language with local variables that hold relations (tuples), constructs for querying and updating such variables, and `if` and `while` control-flow constructs. The translation from Datalog abstract syntax trees to the RAM IR elaborates the semi-naïve evaluation strategy. In particular, every predicate symbol is elaborated into three sets: full, delta, and new. The use of delta sets gives semi-naïve evaluation its speed. The elaboration of Datalog into the RAM IR has many implementation benefits: it enables many optimizations as rewrites, and it makes the interpreter simple.

[11]This has two practical benefits: First, a Flix program with first-class Datalog constraints is treated as a single program by the compiler, enabling optimizations such as monomorphization and inlining without being blocked by an FFI boundary. Second, compilation produces a single self-contained JAR with no external dependencies.

Data Representation. The Flix Datalog solver represents relations and lattices using a concurrent B^+ tree inspired by Soufflé [Jordan et al. 2019c]. The B^+ tree supports range queries and parallel insertions using optimistic locking [Nielsen and Tallouzi 2025]. In particular, the B^+ tree does *not* use Java's `ReentrantLock` but rather Java's `StampedLock` which reduces contention.

Join Reordering and Index Selection. The Flix Datalog solver uses heuristics to perform automatic join reordering [Arch et al. 2022]. Specifically, the solver executes the Datalog program on a subset of the EDB facts to estimate the size of each relation, and then applies Sellinger's algorithm to reorder atoms within each rule [Selinger et al. 1979]. After join reordering, the solver performs mininum index selection using the same algorithm as Soufflé [Subotić et al. 2018]. In particular, the algorithm ensures that every search can be resolved using a range query on an index– i.e. full table scans are avoided unless explicitly required by a Datalog rule.

Parallelism. The compilation from the high-level Datalog abstract syntax into the RAM IR introduces parallel statements. In particular, all rules within the same stratum can be evaluated in parallel. While parallel insertions into relations and lattices (i.e., the concurrent B^+ trees) are allowed, the key property is that no rule will ever read from a relation (or lattice) that is being modified concurrently.

In-and-Out of Datalog. An important question is: *what is the performance cost of moving data in and out of Datalog?* That is, what is the performance overhead of `inject` and `query`? We conducted a simple experiment where given a `List[(a, b)]` of 1M edges (x, y), we construct a `Set[(b, a)]` of the inverted edges (y, x) using (a) `List.foldLeft` and `Set.insert`, and (b) using a Datalog program with the single rule `Result(y, x) :- Edge(x, y)`, and calling `inject` and `query`. The Datalog implementation is approximately 2x–3x slower since it involves (i) copying the list of edges into a B^+ tree, (ii) copying the inverted edges into a B^+ tree, and (iii) converting the B^+ tree to a `Set[(b, a)]`.

We argue that this example is *not* an argument against using Datalog. Rather, the example shows that we should not use Datalog for mundane list or set computations where Datalog evaluation strategies, i.e., semi-naïve and parallel evaluation, cannot offer any benefit. Moreover, the example is degenerate because we would expect most Datalog computations to involve (i) a small input (e.g., the edges of a graph), (ii) a large computation (e.g., the transitive closure of the graph), and (iii) a small output (e.g., the existence of specific paths in the transitive closure). In such Datalog programs, where actual fixpoint computation is required, the overhead of moving data "in-and-out" of Datalog would be minimal.

Performance. Flix has always put ergonomics first and performance second, whereas many other Datalog solvers have made the opposite choice. The performance of the Flix Datalog solver is useable but not yet at the level of state-of-the-art Datalog solvers such as LogicBlox [Aref et al. 2015], Soufflé [Scholz et al. 2016] and GDlog [Sun et al. 2023]. In two simple reachability benchmarks, one using 1M pages from the Danish Wikipedia and the other using 1M edges from the Californian road network, Flix is approximately 8x to 13x slower than the Soufflé interpreter [Hu et al. 2021].

6 Applications

We now present several applications written in Flix. The point is *not* to show that one can write large Datalog programs or large functional programs — that is already known. The point *is* to show that the combination of Datalog and functional programming is elegant and effective and that the features outlined earlier are useful and practical. Table 1 shows an overview of the applications and the features they use. All applications are available in the artifact; we only discuss a subset here.

	First-Class Datalog	Inject & Query	Polymorphism	Rho Abstraction	Head / Guard Exprs	Fun. Predicates	Stratified Negation	Lattice Semantics
Class Hierarchy	✓	✓		✓				
Delivery Date	✓	✓			✓			✓
Ford-Fulkerson	✓	✓	✓		✓			✓
Graph Library †	✓	✓	✓		✓		✓	✓
IFDS Framework	✓	✓	✓		✓	✓		
IDE Framework	✓	✓	✓		✓	✓		✓
Ullman's Algorithm	✓	✓			✓			✓

Table 1. Applications and the features they use. † from [Starup et al. 2023].

6.1 Ullman's Algorithm

We begin with a Datalog-related application: How can we compute whether a Datalog program is stratified? And if so, its strata? Ullman's algorithm is a simple and elegant solution that follows directly from the definition of stratification. We implement it using lattice semantics:

```
def ullman(p: Datalog): Option[Map[PredicateSymbol, Int32]] =
    let numberOfPredicates = /* compute the number of predicates */;
    let (pos, neg) =         /* compute positive and negative edges */;
    let db = inject pos, neg into PositiveEdge/2, NegativeEdge/2;
    let pr = #{
        Stratum(pd; 0) :- PositiveEdge(pd, _).
        Stratum(pd; 0) :- PositiveEdge(_, pd).
        Stratum(pd; 0) :- NegativeEdge(pd, _).
        Stratum(pd; 0) :- NegativeEdge(_, pd).
        Stratum(ph; max(pbs, phs)) :-
            PositiveEdge(ph, pb), Stratum(pb; pbs), Stratum(ph; phs).
        Stratum(ph; max(pbs + 1, phs)) :-
            NegativeEdge(ph, pb),
            Stratum(pb; pbs),
            Stratum(ph; phs),
            if pbs < numberOfPredicates.
    };
    let m = query db, pr select (pd, s) from Stratum(pd; s) |> Vector.toMap;
    let stratified = Map.forAll((_, s) -> s < numberOfPredicates, m);
    if (stratified) Some(m) else None
```

We rely on the same property as Ullman to ensure termination: If we ever encounter a stratum higher than the number of predicates, there must be a negative cycle. If so, we return `None`. Otherwise, we return `Some(m)`, where `m` maps each predicate symbol to its stratum. We have implemented the last stratification check using `Map.forAll`, but we could also have implemented it inside Datalog.

6.2 The IFDS and IDE Frameworks

The Interprocedural Finite Distributive Subset (IFDS) algorithm [Reps et al. 1995] efficiently solves a class of context-sensitive interprocedural dataflow analysis problems by casting such problems in terms of finding reachable nodes in a graph. A declarative implementation of the algorithm, such as in the form of a Datalog program, elegantly shows the essence of the algorithm and the underlying graph, which would be obscured in an imperative implementation by tricky worklists and associated invariants.

6.2.1 IFDS Background. Finite distributive subset analyses are those whose abstract domain is the powerset of some finite set D and whose transfer functions distribute over set union. For example, D could be the set of variables in the program under analysis.

Equivalently, finite distributive subset analyses are those whose transfer functions can be represented by bipartite graphs like the example in Figure 1(a). The graph contains an edge from a node a in the upper part to a node b in the lower part whenever $b \in f(\{a\})$. The special node 0 represents the empty set; there is an edge from 0 to b whenever $b \in f(\{\})$. By distributivity, the function f can be read off from the graph. For example, $f(\{a, b, c\}) = f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\})$, where the values of f on the empty set and on singleton sets are given directly by the edges in the graph.

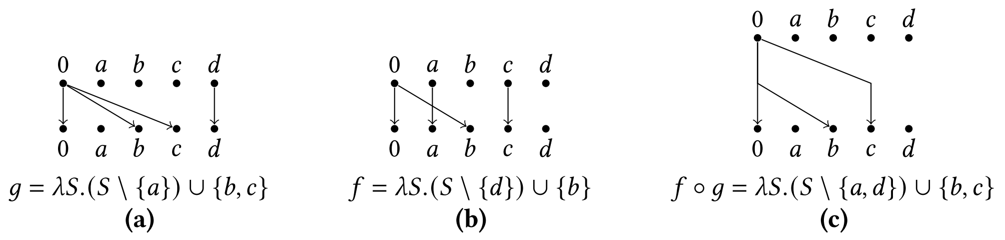
0 a b c d
0 a b c d
$g = \lambda S.(S \setminus \{a\}) \cup \{b, c\}$
(a)
0 a b c d
0 a b c d
$f = \lambda S.(S \setminus \{d\}) \cup \{b\}$
(b)
0 a b c d
0 a b c d
$f \circ g = \lambda S.(S \setminus \{a, d\}) \cup \{b, c\}$
(c)

Fig. 1. Example graph representations of IFDS transfer functions. This example is from Naeem et al. [2010].

This graph representation of transfer functions enables efficient implementations of function composition and pointwise union of functions (i.e. $f \cup g = \lambda S.f(S) \cup g(S)$), the two key operations in a context-sensitive dataflow analysis. Function composition is implemented as the relational product of the edges of two graphs, as shown in Figure 1. Union of functions is implemented by taking the union of the edges in the graphs representing the functions.

The input to the IFDS algorithm is an *exploded supergraph*, which combines the structure of the interprocedural control flow graph of the program under analysis and the graphs representing the transfer functions. The goal of the analysis is to compute the set of supergraph nodes that are reachable from the program start node using paths with a specific property called *realizability*.

6.2.2 Application of Flix Functional Predicates. While the key ideas of IFDS can be simply and elegantly expressed in plain Datalog for illustrative purposes, such an implementation is not practical if it requires the entire exploded supergraph to be provided as an explicit input relation. In practice, that supergraph can be much larger than the set of its realizably reachable nodes, so it can be more costly to explicitly construct that input supergraph than to then run the IFDS algorithm itself. A practical implementation needs to compute parts of the exploded supergraph implicitly on demand, as the IFDS computation demands sets of edges from specific nodes. Instead of representing each transfer function as an explicit bipartite graph, we need to implement it implicitly as a function that, given a node of that graph, computes the set of other adjacent nodes.

A Flix program can incorporate such an implicitly defined relation into an otherwise declarative expression of the fixed-point problem using the functional predicates that were explained in Section 3.9. For example, the rule that handles dataflow within a procedure from program point n to a successor program point m is written as follows:

```
    PathEdge(d1, m, d3) :- CFG(n, m), PathEdge(d1, n, d2), let d3 = eshIntra1(n, d2).
```

For each supergraph node (n, d_2) to which there already exists a realizable path from (s_p, d_1), where s_p is the start node of the procedure that contains program point n, the rule calls the function `eshIntra1` to compute the set of supergraph nodes (m, d_3) to which there are edges in the exploded supergraph from (n, d_2). The head of the rule records that each such supergraph node (m, d_3) is also realizably reachable from (s_p, d_1).

6.2.3 Synergy Between Datalog Relations and Functional Computation. In the case of interprocedural edges from a call site to a called procedure, the IFDS algorithm needs to follow them in both directions, sometimes from caller to callee, and at other times from callee to caller; thus it needs not only the transfer function but also its inverse. An imperative implementation would implement the inverse using an auxiliary table tracking the nodes that the transfer function was called with and the nodes that it returned, along with a worklist to ensure that additions to the table are propagated to all parts of the algorithm that depend on the inverse transfer function. In Flix, this is handled with just a relation that tabulates the calls to the transfer function. Parts of the algorithm that depend on the forward or inverse transfer function are written as rules that depend on this relation, and the Datalog fixed-point semantics ensures that they are all correctly re-evaluated for each new dataflow fact that is tabulated. The Flix rule that tabulates calls to the call-site-to-procedure-start-node transfer function is written as follows:

```
EshCallStart(call, d2, target, d3) :-
    PathEdge(_d1, call, d2),
    CallGraph(call, target),
    let d3 = eshCallStart1(call, d2, target).
```

The `PathEdge` and `CallGraph` atoms identify the nodes for which the transfer function needs to be evaluated. The rule records those evaluations in the `EshCallStart` relation. Other rules that depend on the transfer function, either in the forward or inverse direction, are written to just depend on the `EshCallStart` relation.

6.2.4 Polymorphism for Modularity and Reuse. Using the polymorphism that was explained in Section 3.4, the Flix implementation of IFDS is implemented independently of the types of procedures `p`, program locations `n`, and elements of the dataflow analysis domain `d`. Using the functional nature of Flix, the transfer functions of a specific dataflow analysis can be instantiated to a specific program under analysis and passed as first-class function arguments to the Flix IFDS implementation. Thus, the Flix IFDS implementation is fully modular and generic in not only the dataflow analysis problem being solved and the program under analysis, but even the data types used to represent the analysis domain and the intermediate representation in which that program is written.

6.2.5 IDE Background. The Interprocedural Distributive Environment (IDE) algorithm [Sagiv et al. 1996] extends the ideas of IFDS to dataflow analyses that operate on environments, mappings from D to L, where L is some lattice of values, rather than on just subsets of a set D. For example, the algorithm can perform constant propagation if D is defined as the set of variables and L as the constant propagation lattice. The algorithm works by labeling each edge of the exploded supergraph with a representation of a *microfunction*, a function from L to L. Any distributive transfer function over environments, $(D \to L) \to (D \to L)$, can be represented as a bipartite graph over two copies of D whose edges are labelled with such microfunctions $L \to L$. Unlike the IFDS algorithm, which computes only a set of realizably reachable exploded supergraph nodes, the IDE algorithm additionally computes, for each such reachable node, the merge over paths of the composition of the microfunctions found on each realizable path to the supergraph node.

6.2.6 Application of Lattices. We have implemented the IDE algorithm in Flix as an extension of the IFDS implementation, making use of the lattice semantics that were explained in Section 3.11. The IDE algorithm is defined not only over the lattice L, but over a second lattice F of representations of microfunctions. To be efficient, the algorithm requires an IDE instantiation to define a data structure to compactly represent such a microfunction and efficient implementations of function application and function composition on that representation.

6.2.7 Typeclasses for Instantiating Lattice Operations. Like the IFDS implementation, the Flix IDE implementation is fully modular and generic in the analysis domain, transfer functions, intermediate representation, and program under analysis. It takes an input of the following type:

```
type alias IDE[p, n, d, f, l] = {
    zero            = d,
    main            = p,
    cfg             = List[(n, n)],
    startNodes      = List[(p, n)],
    endNodes        = List[(p, n)],
    callGraph       = List[(n, p)],
    eshIntra        = (n, d) -> Vector[(d, f)],
    eshCallStart    = (n, d, p) -> Vector[(d, f)],
    eshEndReturn    = (p, d, n) -> Vector[(d, f)],
    id              = f,
    apply           = (f, l) -> l,
    compose         = (f, f) -> f
}
```

It also requires type class instantiations to ensure that the types `f` and `l` are lattices. To instantiate the generic implementation, a user provides a type `p` of procedures, a type `n` of program points, a type `d` of elements of the set D, a type `f` of representations of microfunctions, and a type `l` of elements of the dataflow lattice L. The user provides the interprocedural control flow graph using the parameters `main`, `cfg`, `startNodes`, `endNodes`, and `callGraph`. The user provides transfer functions implicitly using the Flix functions `eshIntra` (intraprocedural), `eshCallStart` (for call edges), and `eshEndReturn` (for return edges). Notice that each of these functions returns not only a set of adjacent supergraph nodes `d`, but also a representation `f` of the microfunction that labels each exploded supergraph edge. Finally, the user provides an identity microfunction `id` and implementations of application (`apply`) and composition (`compose`) on the data structure that represents microfunctions.

7 Teaching Logic Programming and Program Analysis with Flix

We have been using Flix in two courses at Aarhus University. In total, we have taught about a hundred and fifty students over a five year period. We briefly share some of our experiences.

We teach a 1st year extracurricular undergraduate course on logic programming. The course is an elective and intended for talented students. The learning objectives are to (a) introduce the declarative logic programming paradigm, (b) illustrate how it can be used to solve small programming problems, and (c) introduce the idea of fixed points. Another course objective is to show the students that there is more to programming than what they are taught in the Introduction to Programming course, which is object-oriented. In the course, the students are given several programming assignments. For example, the students are asked to compute aunts and uncles given a parent-of relation. To our delight, we find that students are quick to learn Datalog.

We also teach a 4th year graduate course on program analysis. The learning objectives of this course are vast. The ones that pertain to this paper are (a) to understand partial orders, lattices, monotone functions, and fixed point theory, (b) to express points-to analyses as Datalog programs, and (c) to express dataflow analyses as Datalog programs enriched with lattice semantics. In the course, the students are asked to implement a context-insensitive points-to analysis for a Java-like language and to implement an intra-procedural sign analysis.

In comparison to DLV [Leone et al. 2006], which was used in previous iterations of the courses, we find that students benefit from (a) the quick and easy installation of Flix, (b) the Flix type system, which prevents simple mistakes, and (c) the IDE support offered by the Flix Visual Studio Code extension. For the 1st year students, we find that the most common mistakes are related to arity mismatches and confusion about term types, both errors caught by Flix. For the 4th year students, we find that two common mistakes are (i) use of non-monotone functions and (ii) incorrect use of negation, both errors which are not caught by Flix. Otherwise, the students comfortably use most features presented in this paper, including first-class Datalog values, the inject-program-query pattern, and lattice semantics.

8 Related Work

Datalog has experienced a resurgence of research interest in the last decade. We present related work along six axes: foundational research, Datalog applications, Datalog and functional programming, Datalog extensions, debugging and provenance, and finally performance and incremental evaluation.

Foundational Research on Datalog. Datalog emerged out of research on databases, knowledge systems, and logic programming in the 1970s and 1980s [Minker 1988]. A major development was stratified negation [Minker 1988]. Another was the discovery of the magic set transformation [Bancilhon et al. 1985]. Research on Datalog later evolved into Answer-Set Programming (ASP) [Brewka et al. 2011]. The last decade has seen a resurgence of interest in Datalog from researchers and industry [De Moor et al. 2012; Huang et al. 2011]. An accessible, if somewhat dated, introduction to Datalog is given by Ceri et al. [1989]. A textbook treatment is given by Greco and Molinaro [2016].

Datalog Applications. Datalog has been used in a variety of applications [Huang et al. 2011], but perhaps one of the most successful applications has been program analysis. Datalog has been used for large scale points-to analysis of Java [Bravenboer and Smaragdakis 2009; Smaragdakis and Bravenboer 2011; Whaley and Lam 2004], especially for context-sensitive analysis [Smaragdakis et al. 2011]. Datalog has also been used for static analysis of smart contracts [Dietrich et al. 2015; Grech et al. 2019, 2020]. Other applications include bioinformatics [Seo 2018], big-data analytics [Halperin et al. 2014; Seo et al. 2013; Shkapsky et al. 2016], and networking and distributed systems [Backes et al. 2019; Conway et al. 2012; Loo et al. 2009].

Embedding Datalog. In addition to Flix, we know of several efforts to integrate Datalog into general-purpose programming languages: Ascent [Sahebolamri et al. 2023, 2022], Datafrog [McSherry 2018], Datafun [Arntzenius and Krishnaswami 2019, 2016], and Functional incA [Pacak and Erdweg 2022]. While Flix can be seen as a meta-programming language for Datalog or as a deeply embedded domain specific language (eDSL), both Datafun and Functional incA aim to blur the distinction between Datalog and the functional world. Specifically, Datafun is a functional programming language with a type system that tracks monotonicity [Arntzenius and Krishnaswami 2016]. The type system ensures that evaluation of Datafun programs terminates. Datafun supports an adapted form of semi-naïve evaluation [Arntzenius and Krishnaswami 2019]. Functional incA is also a functional language but with the goal of compilation to plain old Datalog such that existing solvers can be used. Datafrog is a light-weight Datalog engine written in Rust [McSherry 2018]. Ascent is a Datalog system implemented in Rust [Sahebolamri et al. 2023, 2022]. Ascent allows Datalog programs, expressed as Rust macros, to interact with Rust code. Like Flix, Ascent also leverages type classes (traits) to integrate with user-defined lattices. Flix — unlike Datafun and Functional IncA, but like Ascent and Datafrog — embraces Datalog: First-class Datalog program values *look* and *work* like ordinary Datalog programs and they are solvable by standard Datalog techniques. We believe that the declarative nature of logic rules is an *advantage*.

Datalog Extensions. Formulog is a Datalog dialect extended with support for SMT theories [Bembenek et al. 2020]. Formulog allows programmers to construct logic terms, as part of rule evaluation, which are then passed to an SMT-solver during query evaluation. Formulog, like the original Flix, allow Datalog rules to interact with a small ML-like functional programming language. Dedalus is a Datalog dialect extended with time for use in distributed systems [Alvaro et al. 2011]. In Dedalus, every fact is associated with a specific time stamp. A deductive rule infers new facts in the same time instant as its body predicates. An inductive rule infers new facts in the next time instant as its body predicates. Finally, a persistence rule propagates facts from one time instant to the next. We believe that since Flix retains the Datalogness of Datalog, it should be conceptually straightforward to extend Flix in the direction of Formulog, Dedalus, and other similar extensions.

Debugging and Provenance. A large body of work has studied how to compute provenance information for Datalog, i.e. the ability to answer the question: “how was this fact derived?” [Deutch et al. 2014; Green et al. 2007; Köhler et al. 2012; Zhao et al. 2020]. Two common challenges are: (i) how to efficiently store provenance information such that provenance queries can later be answered, and (ii) how to present provenance information to programmers such that it is understandable. While provenance can precisely answer why a fact was derived, it cannot tell us why a fact is absent. In recent work, Pacak and Erdweg [2023] has gone in a new direction and proposed a debugger for Datalog which allows the programmer to insert breakpoints and to “step-through” how a Datalog rule is evaluated. We have shown how to use lattice semantics as a “poor man’s” provenance.

Performance and Incremental Evaluation. The research literature on efficient evaluation of Datalog is extensive. Over the last decades, impressive progress has been made and state-of-the-art Datalog solvers now scale to billions of facts running on machines with dozens of cores and terabytes of memory. An important strand of research has focused on data structure selection, index selection, and join ordering [Abeysinghe et al. 2024; Jordan et al. 2019a,b; Sahebolamri et al. 2023; Subotić et al. 2018; Veldhuizen 2014]. Another strand has focused on incremental evaluation, i.e. how to recompute the minimal model of a Datalog program when its input is slightly changed [Pacak et al. 2022; Ryzhyk and Budiu 2019; Szabó et al. 2018, 2021, 2016]. A more recent strand has focused on compilation of Datalog to GPUs [Martínez-Angeles et al. 2014; Shovon et al. 2022; Sun et al. 2023].

9 Conclusion

We have presented a “Grand Tour” of the Datalog facilities in the Flix programming language. We have illustrated how programmers can write functions implemented using Datalog and we have shown how to build modular and reusable families of Datalog programs using first-class Datalog values, rho abstraction, parametric polymorphism, and type classes. We have described the `inject` and `query` programming constructs, which allows programmers to move data between the functional and Datalog worlds. We have described a collection of features that significantly increase the flexibility and expressive power of Datalog, including head and guard expressions, functional predicates, lattice semantics, and more.

We have presented several applications that use these features, including implementations of Ullman’s algorithm to stratify Datalog programs and the IFDS and IDE algorithms for context-sensitive program analysis, and more. The implementations of IFDS and IDE fulfill a long-term goal: to have fully modular, polymorphic, typed, and declarative formulations of these algorithms that can be instantiated with any abstract domain.

We believe that embedding Datalog inside a general-purpose programming language, such as Flix, is an important step toward bringing the power of logic programming to the masses.

10 Data-Availability Statement

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

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

A fully functional and reusable artifact is available [Madsen and Lhoták 2025].

References

Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. 2024. Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. *Proceedings of the ACM on Programming Languages* 8, POPL (2024). doi:10.1145/3632928

Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. *Dedalus: Datalog in Time and Space.* Springer Berlin Heidelberg. doi:10.1007/978-3-642-24206-9_16

Krzysztof R. Apt and Roland N. Bol. 1994. Logic programming and negation: A survey. *The Journal of Logic Programming* 19–20 (1994). doi:10.1016/0743-1066(94)90024-8

Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotić, and Bernhard Scholz. 2022. Building a join optimizer for soufflé. In *International Symposium on Logic-Based Program Synthesis and Transformation.*

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* doi:10.1145/2723372.2742796

Michael Arntzenius and Neel Krishnaswami. 2019. Seminaïve evaluation for a higher-order functional language. *Proceedings of the ACM on Programming Languages* 4, POPL (2019). doi:10.1145/3371090

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming.* doi:10.1145/2951913.2951948

John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. 2019. *Reachability Analysis for AWS-Based Networks.* Springer International Publishing. doi:10.1007/978-3-030-25543-5_14

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems.* doi:10.1145/6012.15399

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428209

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications.* doi:10.1145/1640089.1640108

Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. 2011. Answer set programming at a glance. *Commun. ACM* 54, 12 (2011). doi:10.1145/2043174.2043195

Vince Bárány, Balder ten Cate, and Martin Otto. 2012. Queries with guarded negation. *Proceedings of the VLDB Endowment* 5, 11 (2012). doi:10.14778/2350229.2350250

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

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing.* doi:10.1145/2391229.2391230

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms.* MIT press.

Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. 2012. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers.* Vol. 6702. Springer.

Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for Datalog Provenance. doi:10.5441/002/ICDT.2014.22

Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-scale exhaustive points-to analysis for Java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* doi:10.1145/2814270.2814307

Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* doi:10.1109/icse.2019.00120

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: analyzing the out-of-gas world of smart contracts. *Commun. ACM* 63, 10 (2020). doi:10.1145/3416262

Sergio Greco and Cristian Molinaro. 2016. *Datalog and logic databases.* Springer Nature.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. doi:10.1145/1265530.1265535

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. 2014. Demonstration of the Myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. doi:10.1145/2588555.2594530

Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. Artifact for Paper: An Efficient Interpreter for Datalog by De-specializing Relations. doi:10.1145/3410297

Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. doi:10.1145/1989323.1989456

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019a. Brie: A Specialized Trie for Concurrent Datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. doi:10.1145/3303084.3309490

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019b. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. doi:10.1145/3293883.3295719

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019c. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. doi:10.1145/3293883.3295719

Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. *Declarative Datalog Debugging for Mere Mortals*. Springer Berlin Heidelberg. doi:10.1007/978-3-642-32925-8_12

Daan Leijen. 2004. First-class labels for extensible rows. (2004).

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7, 3 (2006). doi:10.1145/1149114.1149117

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

Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023). doi:10.1145/3607846

Magnus Madsen and Ondřej Lhoták. 2018. Safe and sound program analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3213846.3213847

Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the masses: programming with first-class Datalog constraints. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428193

Magnus Madsen and Ondřej Lhoták. 2025. *Flix: A Design for Language-Integrated Datalog (artifact)*. doi:10.5281/zenodo.15743443

Magnus Madsen, Jonathan Lindegaard Starup, and Ondřej Lhoták. 2022. Flix: A meta programming language for datalog. In *Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry*.

Magnus Madsen, Jonathan Lindegaard Starup, and Matthew Lutze. 2023a. Restrictable Variants: A Simple and Practical Alternative to Extensible Variants. doi:10.4230/LIPICS.ECOOP.2023.17

Magnus Madsen and Jaco van de Pol. 2020. Polymorphic types and effects with Boolean unification. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428222

Magnus Madsen and Jaco van de Pol. 2021. Relational nullable types with Boolean unification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021). doi:10.1145/3485487

Magnus Madsen, Jaco van de Pol, and Troels Henriksen. 2023b. Fast and Efficient Boolean Unification for Hindley-Milner-Style Type and Effect Systems. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). doi:10.1145/3622816

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/2908080.2908096

Carlos Alberto Martínez-Angeles, Inês Dutra, Vítor Santos Costa, and Jorge Buenabad-Chávez. 2014. *A Datalog Engine for GPUs*. Springer International Publishing. doi:10.1007/978-3-319-08909-6_10

Frank McSherry. 2018. Rust-Lang/datafrog: A Lightweight Datalog engine in Rust. https://github.com/rust-lang/datafrog. [Accessed 23-06-2025].

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. doi:10.1145/1142473.1142552

Jack Minker. 1988. *Foundations of deductive databases and logic programming*. Morgan Kaufmann.

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). doi:10.1145/3290325

Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. *Practical Extensions to the IFDS Algorithm.* Springer Berlin Heidelberg. doi:10.1007/978-3-642-11970-5_8

Casper Dalgaard Nielsen and Adam Yasser Tallouzi. 2025. *Improving the Datalog Engine of Flix.* Master's thesis. Aarhus University. Master's Thesis.

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. doi:10.4230/LIPICS.ECOOP.2022.7

André Pacak and Sebastian Erdweg. 2023. Interactive Debugging of Datalog Programs. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). doi:10.1145/3622824

André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* doi:10.1145/3564719.3568686

Adam Paszke and Ningning Xie. 2023. Infix-Extensible Record Types for Tabular Data. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development.* doi:10.1145/3609027.3609406

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95.* doi:10.1145/199448.199462

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. *Datalog* 2 (2019).

Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1–2 (1996). doi:10.1016/0304-3975(96)00072-2

Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher Micinski. 2023. Bring Your Own Data Structures to Datalog. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). doi:10.1145/3622840

Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction.* doi:10.1145/3497776.3517779

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction.* doi:10.1145/2892208.2892226

P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data.*

Jiwon Seo. 2018. Datalog Extensions for Bioinformatic Data Analysis. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC).* doi:10.1109/embc.2018.8512571

Jiwon Seo, S. Guo, and M. S. Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* doi:10.1109/icde.2013.6544832

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data.* doi:10.1145/2882903.2915229

Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. 2022. Accelerating Datalog applications with cuDF. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3).* doi:10.1109/ia356718.2022.00012

Yannis Smaragdakis and Martin Bravenboer. 2011. *Using Datalog for Fast and Easy Program Analysis.* Springer Berlin Heidelberg. doi:10.1007/978-3-642-24206-9_14

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* doi:10.1145/1926385.1926390

Jonathan Lindegaard Starup, Magnus Madsen, and Ondřej Lhoták. 2023. Breaking the Negative Cycle: Exploring the Design Space of Stratification for First-Class Datalog Constraints. doi:10.4230/LIPICS.ECOOP.2023.31

Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic index selection for large-scale datalog computation. *Proceedings of the VLDB Endowment* 12, 2 (2018). doi:10.14778/3282495.3282500

Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Kristopher Micinski, and Sidharth Kumar. 2023. Optimizing Datalog for the GPU. doi:10.48550/ARXIV.2311.02206

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). doi:10.1145/3276509

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* doi:10.1145/3453483.3454026

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* doi:10.1145/2970276.2970298

Jeffrey D. Ullman. 1988. *Principles of database and knowledge-base systems.* Computer Science Press.

Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.

John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. doi:10.1145/996841.996859

Carlo Zaniolo, Natraj Arni, and Kayliang Ong. 1993. *Negation and aggregates in recursive rules: the LDL++ approach*. Springer Berlin Heidelberg. doi:10.1007/3-540-57530-8_13

David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Transactions on Programming Languages and Systems* 42, 2 (2020). doi:10.1145/3379446

Received 2025-03-19; accepted 2025-08-12