Flix: A Meta Programming Language for Datalog

Magnus Madsen[1], Jonathan Lindegaard Starup[1] and Ondřej Lhoták[2]

[1]Department of Computer Science, Aarhus University, Denmark
[2]David R. Cheriton School of Computer Science, University of Waterloo, Canada

Abstract

We illustrate how Flix can be used as a powerful and expressive meta programming language for Datalog.

1. Introduction

Flix is a functional-first, imperative, and logic programming language [1, 2, 3]. Flix has algebraic data types, pattern matching, higher-order functions, channel and process-based concurrency, parametric polymorphism, type classes, and higher-kinded types. The Flix type and effect system is based on Hindley-Milner and supports complete type inference.

A unique feature of Flix is its support for Datalog programs as first-class values. Datalog values can be stored in local variables, passed as arguments to functions, returned from functions, stored into data structures, composed with other Datalog values, and have their minimal model queried. Thus Flix can be viewed as a powerful meta programming language for Datalog.

Datalog expressions and their values are structurally typed. The type system ensures that predicate symbols, and their terms, have consistent types within the same Datalog program. Polymorphism enables programmers to write modular and reusable families of Datalog programs. The type system also ensures that every Datalog value constructed at runtime is stratified.

Flix extends Datalog from *constraints on relations* to *constraints on lattices*. Lattice semantics makes it possible to declaratively express many applications, including program analyses and two-players games, which cannot readily be expressed in pure Datalog. Flix also supports stratified negation and a modularity mechanism which we call local predicates.

During our work, we have discovered a new programming pattern, which we dub the *inject-program-query* pattern, where a function is implemented by turning its arguments into Datalog facts, solving a Datalog program, and returning a subset of the minimal model as the result.

Flix embraces the essence of Datalog: Datalog program values are declarative, they look like ordinary Datalog clauses, they have a minimal model, and they are solvable by standard techniques such as semi-naïve evaluation.

We believe that embedding Datalog inside a general-purpose programming language, such as Flix, enables programmers to use Datalog for the specific programming tasks where it really shines: to declaratively express and solve fixed-point computations on relations and lattices with the advantages of a static type system, a large standard library, and excellent IDE support.

Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry, September 05, 2022, Genova - Nervi, Italy

magnusm@cs.au.dk (M. Madsen); jls@cs.au.dk (J. L. Starup); olhotak@uwaterloo.ca (O. Lhoták)

CEUR Workshop Proceedings (CEUR-WS.org)

2. Programming with Datalog in Flix

We illustrate, with a series of small examples, how Flix can be used as a typed meta programming language for Datalog. All examples are runnable Flix programs.

Inject–Program–Query. Flix is primarily a functional programming language. Flix programs are structured around immutable data structures and functions that operate on them. But in Flix we can write a function whose internal implementation uses Datalog to compute its output from its inputs. For example, we can write a function that when given a graph computes its transitive closure:

```
def clo(g: List[(Int32, Int32)]): List[(Int32, Int32)] =
    let db = inject g into Edge;
    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)
```

The clo function takes a graph, represented as a list of edges, computes its transitive closure, and returns the result as a list of edges. The function is implemented as follows: First, we convert the list of edges into a set of Datalog Edge facts and we store the result in the local variable db. In Datalog terminology, we are constructing the EDB from the input list. This is needed because we have to associate the predicate symbol Edge with the list of tuples in g and to build an index on them. Second, we define a Datalog value which computes the transitive closure of the Edge relation and we store the program in the local variable pr. In Flix, the special syntax `#{...}` delimits a Datalog value. Third, and finally, we use the query construct to compose db and pr into one Datalog program value, to compute its minimal model, and to extract all pairs (x, y) from the Path relation and return them as a list. Note that the minimal model is not computed until we explicitly use query. We name this programming-style the *inject-program-query* pattern.

We could have written the clo function in a functional– or imperative–style, but neither would have been as short, elegant, clear, obviously correct, and efficient as the Datalog style. That's the power of declarative logic programming!

Polymorphism. In the previous example, the signature of the clo function was:

```
def clo(g: List[(Int32, Int32)]): List[(Int32, Int32)]
```

but nothing in the implementation of the function requires the graph to have integer vertices. In Flix, Datalog program values may be polymorphic in their term types, and hence we could have used the more general signature:

```
def clo(g: List[(t, t)]): List[(t, t)] with Eq[t], Order[t]
```

This signature works for any graph provided that the polymorphic type t conforms to the Eq and Order type classes which enable the Datalog solver to compare terms for equality and to build indexes on them. With the polymorphic signature, we can reuse the clo function for graphs with different types of vertices. For the remainder of the paper, for brevity, we will omit the Eq and Order constraints, but they are of course required when polymorphic types are used. The Flix standard library include instances of these type classes for all base types and for data structures such as lists, maps, and sets.

Integration with Lexical Scope. In Flix, Datalog values are not islands; they can reference the lexical scope of the functional language. In other words, a Datalog value can capture variables from the lexical scope. For example, we can write a function that when given a graph, a source vertex, and a destination vertex, computes if there is path from the source to the destination:

```
def reach(g: List[(t, t)], s: t, d: t): Bool =
    let db = inject g into Edge;
    let pr = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
        Yes() :- Path(s, d).
    };
    query db, pr select () from Yes() |> nonEmpty
```

The reach function takes a graph g, a source vertex s, and a destination vertex d as input, and returns returns true if there is a path from s to d in the graph g. The last rule in the Datalog program states that the nullary predicate Yes holds if there is a Path(s, d) *fact* where s and d are *not* Datalog variables, but rather Flix program variables bound by the lexical scope, i.e. the function arguments. Thus, at runtime, the Datalog program has concrete values for s and d.

Integration with Expressions. Datalog values may also refer to expressions:

```
def clo(g: List[(t, l, t)], p: l -> Bool): List[(t, t)] =
    let db = inject g into Edge;
    let pr = #{
        Path(x, y) :- Edge(x, w, y), if p(w).
        Path(x, z) :- Path(x, y), Edge(y, w, z), if p(w).
    };
    query db, pr select (x, y) from Path(x, y)
```

This clo function takes a labelled graph as input, represented as a list of triples, and returns its transitive closure as a list of edges. The function also takes a predicate function p : $l \to$ Bool which determines when an edge is active. We use the predicate in the body of the two rules, writing if p(w), illustrating that Datalog rules may refer to Boolean expressions in the functional language of Flix. In particular, p is *not* a logical predicate, but a reference to a Flix expression.

Datalog Programs as First-Class Values. We can refactor the first example into multiple functions for better modularity and reuse:

```
def paths(): #{ Edge(t, t), Path(t, t) | r } = #{
    Path(x, y) :- Edge(x, y).
    Path(x, z) :- Path(x, y), Edge(y, z).
}
```

Here the paths function returns a Datalog *value*. The (row) polymorphic type captures that the value contains Edge and Path predicate symbols that are of arity two and whose term types are of type t. The row variable r allows us to combine this value with other Datalog values that may contain additional predicate symbols.
We can rewrite the clo function to use the new paths function:

```
def clo(g: List[(t, t)]): List[(t, t)] =
    let db = inject g into Edge;
    query db, paths() select (x, y) from Path(x, y)
```

Local Predicates. We often want to write Datalog program values where some predicate symbols are local to the computation. Flix supports such *local predicates*, which are given unique, fresh names at runtime, allowing us to hide them from the type of a Datalog value. We can think of such local predicates as similar to local variables. For example, we can write:

```
def unconnected(): #{ Edge(t, t), Unconnected(t, t) | r } =
    #(Edge, Unconnected) -> #{
        Node(x) :- Edge(x, _). Node(y) :- Edge(_, y).
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
        Unconnected(x, y) :-
            Node(x), Node(y), not Path(x, y).
    }
```

The unconnected function returns a Datalog value which computes the set of pairs of vertices that are not connected by any path of edges. The Edge facts are the *input* and the Unconnected facts are the *output* of the Datalog value. The predicates Edge and Path are local.

The key construct is `#(Edge, Unconnected) -> e` which is reminiscent of a lambda abstraction in that it evaluates e to a Datalog value v and then gives fresh names to every predicate symbol other than Edge and Unconnected in v. This renaming, which is similar to alpha renaming, hides the Node and Path predicates symbols ensuring that they are completely local to the Datalog value, i.e. they are given unique names that do not overlap with any other names.

From Relations to Lattices. Datalog clauses are *constraints on relations*, but Flix also supports *constraints on lattices*. Given a lattice $L = (E, \bot, \sqsubseteq, \sqcup, \sqcap)$ implemented as instances of the type classes LowerBound, PartialOrder, JoinLattice, and MeetLattice we can associate one or more predicate symbols with the lattice. For example, we can define a new type `enum N(Int32)` where the bottom element is `N(Int32.maxValue())` and the partial order is $\geq$. In other words, the lattice is turned "upside down". We can use this lattice to write a Datalog program that computes the single-source shortest-distance (SSSD) from an origin vertex in a graph:

```
def sssd(g: List[(t, Int32, t)], o: t): Map[t, N] =
    let db = inject g into Edge;
    let pr = #{
        Dist(o; N(0)).
        Dist(y; add(d1, d2)) :- Dist(x; d1), Edge(x, d2, y).
    };
    query db, pr select (x, d) from Dist(x; d) |> toMap
```

The sssd function returns a map from every vertex in the graph g to its shortest distance from the origin vertex o. The use of the semi-colon in the head atoms indicates that the Dist predicate symbol is given a lattice interpretation where the lattice is determined by its type. The add function (omitted for brevity) simply adds an integer and a N(x) lattice element. Flix functions used in head atoms must be strict and monotone according to the lattice ordering.

Intuitively, the program can be understood as follows: Initially, the distance to every vertex in the graph is $\bot$ (i.e. `Int32.maxValue()`). The fact `Dist(o; N(0))` asserts that the distance to the origin is zero (i.e. $\top$). The rule asserts that if a vertex x is reachable with distance d1 and there is an edge from x to y with distance d2 then the distance to y is *at least the sum of the two distances d1 and d2*. But since the lattice is upside down, "at least" is "at most". Thus the program computes the shortest distance to every vertex from the origin.

Type System. Datalog expressions are structurally typed, i.e. predicate symbols, and the types of their terms, never have to be declared; they are simply used and their types are inferred. Every Datalog expression is typed with a row type that keeps track of the term types of every predicate symbol. Thus, in every Datalog program value, every predicate symbol is used with consistent arity and term types.

Minimal Models. While Flix is Turing-complete, and hence evaluation of an expression may diverge, if a Datalog expression reduces to a Datalog value then that value has a minimal model. Moreover, a Datalog program enriched with lattice semantics has a minimal model if the lattice components indeed form a lattice and if all operations on the lattice are strict and monotone.

Stratified Negation. Flix supports stratified negation. The Flix type and effect system ensures — at compile-time — that any Datalog value constructed at runtime is stratified. If not, the Flix compiler rejects the program with a compilation error and reports the negative cycle in the dependency graph.

Datalog Engine. Flix programs compile to Java bytecode and run on the JVM. The Flix Datalog engine is a Just-In-Time (JIT) compiler written in Flix itself. At runtime, a Datalog program value is compiled to the Relational Algebra Machine (RAM), an intermediate representation (IR) comparable to C with relational operators, which is then optimized and executed.

3. Ecosystem & Tooling

Flix has a website that describes the language (flix.dev), online documentation (doc.flix.dev), API documentation à la Javadoc (api.flix.dev), an online playground (play.flix.dev), and a fully-featured Visual Studio Code extension.

Flix comes with an extensive standard library that includes common functional data structures such as Option, Result, List, Set, and Map. The library spans more than 30,000 lines of code and offers more than 2,600 functions. Flix also has a Java FFI making it possible to reuse large parts of the Java Class Library. Flix is ready for use, open source, and freely available at: https://flix.dev/

4. Conclusion

Flix is a functional, imperative, and logic programming language with support for Datalog programs as first-class values. We have illustrated how Flix can be used as a powerful, expressive, and typed meta programming language for Datalog. We hope that Flix will help bring Datalog programming to a broader audience.

References

[1] M. Madsen, O. Lhoták, Fixpoints for the Masses: Programming with first-class Datalog Constraints, Proc. of the PACMPL (2020).

[2] M. Madsen, J. van de Pol, Polymorphic Types and Effects with Boolean Unification, Proc. of the PACMPL (2020).

[3] M. Madsen, M. Yee, O. Lhoták, From Datalog to Flix: A Declarative Language for Fixed Points on Lattices, in: Proc. of Programming Language Design and Implementation (PLDI), 2016.