From Datalog to FLIX: A Declarative Language for Fixed Points on Lattices

Magnus Madsen  
University of Waterloo, Canada  
mmadsen@uwaterloo.ca

Ming-Ho Yee  
University of Waterloo, Canada  
ming-ho.yee@uwaterloo.ca

Ondřej Lhoták  
University of Waterloo, Canada  
olhotak@uwaterloo.ca

Abstract

We present FLIX, a declarative programming language for specifying and solving least fixed point problems, particularly static program analyses. FLIX is inspired by Datalog and extends it with lattices and monotone functions. Using FLIX, implementors of static analyses can express a broader range of analyses than is currently possible in pure Datalog, while retaining its familiar rule-based syntax.

We define a model-theoretic semantics of FLIX as a natural extension of the Datalog semantics. This semantics captures the declarative meaning of FLIX programs without imposing any specific evaluation strategy. An efficient strategy is semi-naïve evaluation which we adapt for FLIX. We have implemented a compiler and runtime for FLIX, and used it to express several well-known static analyses, including the IFDS and IDE algorithms. The declarative nature of FLIX clearly exposes the similarity between these two algorithms.

Categories and Subject Descriptors F3.2 [Semantics of Programming Languages]: Program Analysis

General Terms Logic Programming, Static Analysis

Keywords logic programming, static analysis, Datalog

1. Introduction

Least fixed point problems are ubiquitous in mathematics and computer science, significantly in programming languages, and particularly in program analysis. Given a monotone function \( F \) on a lattice, the goal is to find the least \( x \) for which \( F(x) = x \). At the lowest and most general level, a program is a function \( F \) that instructs a machine how to change its overall state at each computation step. A static analysis computes an abstract state \( \hat{x} \) that over-approximates all possible concrete states that a program can reach. Every sound approximation must satisfy \( F(\hat{x}) \subseteq \hat{x} \), where \( F \) is an abstraction of the concrete transformation function \( F \), since if a state in \( \hat{x} \) can be reached by a computation, then so can a state in \( F(\hat{x}) \). The least \( \hat{x} \) satisfying this property can be computed by starting from the least element \( \bot \) and iteratively applying \( F \) until the fixed point is reached [15, 35].

Static analyzers, which involve fixed-point computations, are complex pieces of software often implemented in general-purpose languages such as C++ or Java. The many mutual dependencies, imposed by the fixed-point problem, are typically expressed using a complex arrangement of worklists. The decision of how to structure the worklists is global, so these large analyzers become difficult to restructure and modify. It also becomes difficult to understand precisely the analysis problem that the implementation is actually solving, and to assure oneself that the implementation is correct. Moreover, the complexity of the dependencies between inter-related sub-analyses often leads analysis implementors to sacrifice precision. For example, some interprocedural analysis frameworks use a call graph precomputed with conservative assumptions to compute dataflow information that would enable a more precise call graph.

To overcome these difficulties, some analysis designers have turned to Datalog [7, 9, 63]. A Datalog program is a set of rules and its solution is the minimal model that satisfies those rules. An important benefit of a Datalog program is its modularity: because the rules are declarative, individual sub-analyses can be easily composed by taking the union of their rules, and the Datalog solver takes care of the mutual dependencies automatically. Thus, it is easy to understand an analysis by understanding its components individually. Correctness of each component implies correctness of the overall analysis. Furthermore, Datalog solvers implement many important optimizations, such as index selection, query planning, and parallel execution [3, 27, 28, 58]. In a handcrafted static analyzer, each of these optimizations must be implemented manually.
However, Datalog has important limitations that restrict its applicability to program analysis:

**Lattices.** Datalog is inherently limited to rules on relations, i.e. powersets of tuples, but common static analyses operate over a wide variety of other lattices. Some simple lattices can be embedded in powersets, but at a very high computational cost, and more interesting lattices cannot be encoded at all. For example, we can embed the constant propagation lattice over a finite domain in the following way: $\bot$ is represented by the empty set, each constant is represented by a singleton set, and $\top$ is represented by any set that contains a specially designated $\top$ element. We then add a rule that adds the $\top$ element to every set of two or more elements. However, this $\top$ rule cannot prevent the Datalog program from processing the original non-singleton, non-$\top$ sets. We get the worst of both worlds: the precision is the same as with the constant propagation lattice, but the computational cost is the same as with the much more expensive arbitrary-sets-of-constants lattice. Moreover, when the domain of the constants is infinite, such as the integers, the lattice cannot be encoded at all.

**Functions.** A related issue is that functions can only be expressed as tabulated relations in Datalog. This can be cumbersome and slow, and functions with an infinite domain or codomain cannot be expressed at all. Consequently, most operations on lattices cannot be expressed. Returning to the constant propagation lattice, even if the lattice itself could somehow be expressed in Datalog, there would be no way to express abstract addition or multiplication on its elements. Moreover, the lack of functions, as well as compound datatypes, means that even a simple context-sensitive analysis such as $k$-CFA cannot be expressed.

**Interoperability.** A practical limitation of most Datalog solvers is that they do not interact well with existing infrastructure. For instance, static analyzers are often forced to serialize their facts to a file, externally execute the solver, and then read the solution back from a file. Besides additional overhead, this requires a tedious mapping between the analysis and the Datalog solver format.

To overcome these limitations, we propose FLIX, a new declarative language for fixed-point problems. FLIX is a rule-based language inspired by Datalog and extended with lattices and monotone functions.

In summary, this paper makes the following contributions:

- We present FLIX, a new programming language inspired by Datalog and extended with lattices and functions.
- We define a model-theoretic semantics of FLIX as a natural generalization of the Datalog semantics.
- We show how several well-known static analyses, which are inexpressible in Datalog, can be expressed in FLIX.
- We present declarative formulations of the IFDS and IDE algorithms in FLIX, which clearly and concisely demonstrate that IDE is a generalization of IFDS.
- We discuss our implementation of a preliminary FLIX solver and its relation to semi-naive evaluation. We experimentally compare its performance to hand-crafted static analyzers and the DLV Datalog solver.

## 2. Motivation

### 2.1 Points-to Analysis with Datalog

Undoubtedly, the “killer-app” for Datalog has been points-to analysis of object-oriented programs [4, 7, 30, 37, 54, 55, 63]. This involves the specification and computation of sophisticated whole-program subset-based points-to analysis [1, 31].

Consider the following Java fragment:

```java
1 ClassA o1 = new ClassA() // object A
2 ClassB o2 = new ClassB() // object B
3 ClassB o3 = o2;
4 o2.f = o1;
5 Object r = o3.f; // Q: What is r?
```

A points-to analysis for this program can help answer questions such as: “to what object can the local variable $r$ point?”

In this case, $r$ can point to object $A$ since: (i) variable $o1$ points to object $A$, (ii) variable $o2$ points to object $B$, (iii) variable $o3$ points to the value of variable $o2$ which is object $B$, (iv) object $A$ is written to field $f$ of object $B$ due to i and ii, (v) the value of $r$ is object $A$ since variable $o3$ points to object $B$ due to iii, and the value of field $f$ is $A$ due to iv.

Unfortunately, such complicated reasoning is necessary due to the multiple recursive dependencies in the dataflow of an object-oriented program: specifically, that local variable information depends on heap information which itself depends on local variable information. Datalog provides an elegant formalism to express such recursive rules. Figure 1 shows a points-to analysis for a minimal object-oriented language. This formulation has just four rules, but is sufficient to capture the entirety of the above reasoning.

The program defines six relations: the four input relations `New`, `Assign`, `Load`, and `Store`, and the two derived relations `VarPointsTo` and `HeapPointsTo` which hold the solution. To analyze the code fragment above, we represent it as a set of facts that are fed to a Datalog solver:

```
New("o1", "A").
New("o2", "B").
Assign("o3", "o2").
Store("o2", "f", "o1").
Load("r", "o3", "f").
```

Running the solver infers a solution containing the fact `VarPointsTo("r", "A")`, as expected. As an example of how to understand a Datalog program, consider the second rule in Figure 1. This rule states that if there is an assignment from $v2$ to $v1$ and $v2$ points to some object $h2$, then $v1$ points to $h2$. Furthermore, notice the mutual recursion in the rules for `VarPointsTo` and `HeapPointsTo` in Figure 1.
The analysis presented in the previous section is sufficient defining what is essentially an instance of a type class \[62\.\]

A lattice definition is a built-in mechanism for partial order, \(\bot\) is the bottom element, \(\top\) is the top element, \(\sqsubseteq\) is the partial order of parity lattice elements. \(F\) define the components of a lattice and to express monotone type, and its expression body. \(F\) define a named function, the type of its arguments, its return tag must be prefixed by the enum name, thus \(\text{Parity.Odd}\) of the elements of the parity lattice. In \(F\) inspired by Scala and Datalog.

The syntax of \(F\) is a set of declarations to specify the names, arities, and types their associated operations, and monotone functions over functional programming language for specifying lattices, combined with a logic language for expressing errors. This analysis requires both points-to and dataflow analysis to determine if the denominator in a division is possibly zero. We can use a constant propagation analysis or interval analysis to discover this information. However, as argued in the introduction, such analyses are not expressible in Datalog, but they are expressible in FLIX.

In this section, we show how to develop such an analysis, but for the sake of exposition, we use the parity lattice instead of the constant propagation or interval lattices. As a reminder, the parity lattice tracks whether numbers are odd or even. Figure 2 shows a FLIX program that extends the points-to analysis with a dataflow analysis.

The FLIX language is composed of two parts: a pure functional programming language for specifying lattices, their associated operations, and monotone functions over their elements, combined with a logic language for expressing rules over relations and lattices. Bringing everything together is a set of declarations to specify the names, arities, and types of functions, relations, and lattices. The syntax of FLIX is inspired by Scala and Datalog.

We briefly walk through the program in Figure 2.

The \textit{enum definition} on lines 5–9 defines a tagged union of the elements of the parity lattice. In FLIX, references to a tag must be prefixed by the enum name, thus \texttt{Parity.Odd} refers to the odd element of the parity enum.

The \textit{function definitions} on lines 13–20 and 23–24 each define a named function, the type of its arguments, its return type, and its expression body. FLIX functions are used to define the components of a lattice and to express monotone filter and transfer functions. Here, the 1\texttt{eq} function defines the partial order of parity lattice elements. FLIX functions are expressed using a small, pure functional language.

The \textit{lattice definition} on lines 28–29 associates a 5-tuple \((\bot, \top, \sqsubseteq, \sqcup, \sqcap)\) of lattice components with a type, where \(\bot\) is the bottom element, \(\top\) is the top element, \(\sqsubseteq\) is the partial order, \(\sqcup\) is the least upper bound, and \(\sqcap\) is the greatest lower bound. A lattice definition is a built-in mechanism for defining what is essentially an instance of a type class [62].

\begin{verbatim}
VarPointsTo(v1, h1) :- New(v1, h1).
VarPointsTo(v1, h2) :- Assign(v1, v2),
  VarPointsTo(v2, h2).
VarPointsTo(v1, h2) :- Load(v1, v2, f),
  VarPointsTo(v2, h1),
  HeapPointsTo(h1, f, h2).
HeapPointsTo(h1, f, h2) :- Store(v1, f, v2),
  VarPointsTo(v1, h1),
  VarPointsTo(v2, h2).
\end{verbatim}

\begin{verbatim}
// an almost complete FLIX program.

// an enum definition that defines
// the elements of the parity lattice.
enum Parity {
  case Top,
  case Even, case Odd,
  case Bot
}

// a function definition that defines
// the partial order of the parity lattice.
def leq(e1: Parity, e2: Parity): Bool =
  match (e1, e2) with {
    case (Parity.Bot, _) => true
    case (Parity.Even, Parity.Odd) => true
    case (Parity.Odd, Parity.Odd) => true
    case (_, Parity.Top) => true
    case _ => false
  }

// additional lattice definitions ...
def lub(e1: Parity, e2: Parity): Parity = ...
def glb(e1: Parity, e2: Parity): Parity = ...

// association of the lattice operations
// with the parity type.
let Parity< = (Parity.Bot, Parity.Top,
  leq, lub, glb);

// monotone filter and transfer functions ...
def isMaybeZero(e: Parity): Bool = ...
def sum(e1: Parity, e2: Parity): Parity = ...

// declaration of relations ...
rel Load(var: Str, base: Str, field: Str);
rel VarPointsTo(var: Str, obj: Str);

// additional declarations ...

// declaration of lattices ...
lat IntVar(var: Str, Parity<>);
lat IntField(var: Str, field: Str, Parity<>);

// additional declarations ...

// VarPointsTo and HeapPointsTo rules ...

lat IntVar(v, i) :- Int(v, i).
lat IntVar(v, i) :- Assign(v, v2), IntVar(v2, i).
lat IntVar(v, i) :- Load(v, v2, f),
  VarPointsTo(v2, h),
  IntField(h, f, i).
lat IntField(h, f, i) :- Store(v1, f, v2),
  VarPointsTo(v1, h),
  IntVar(v2, i).

// rule for addition of parity elements.
lat IntVar(r, sum(i1, i2)) :- AddExp(r, v1, v2),
  IntVar(v1, i1),
  IntVar(v2, i2),

// rule for potential division-by-zero errors.
ArithmeticError(r) :- DivExp(r, v1, v2),
  IntVar(v2, i2),
  isMaybeZero(i2).
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{A field-sensitive subset-based points-to analysis for an object-oriented programming language, e.g. Java.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{A subset-based, field-sensitive points-to analysis combined with a dataflow analysis expressed in FLIX.}
\end{figure}
Like Haskell, FLIX allows only one type class instance per type. The definition assumes that the supplied functions satisfy the properties of a complete lattice; otherwise, the semantics of the FLIX program is undefined. The notation Parity<> is used to distinguish the parity type from the parity type class instance.

The relation declaration on line 36 (and 37) defines a relation by specifying its name and its attributes (columns), together with their names and types. The lattice declaration on line 41 (and 42) defines a lattice by specifying its name and its attributes (columns), together with their names and types. The last attribute of a lattice declaration must have a type equipped with a lattice. Intuitively, the IntVar lattice is the map lattice from strings to elements of the parity lattice.

A rule definition in FLIX is similar to a Datalog rule, but is more expressive. First, a FLIX rule may contain function applications in the last term of the head predicate of a rule. For example, consider line 59:

\[ \text{IntVar}(v, \text{sum}(v1, v2)) :- \ldots \]

Here, the sum function is used to compute the abstract sum of two parity lattice elements. The function must be monotone in its arguments. Second, a FLIX rule may use a filter function which is monotone over the boolean. For example, consider lines 64–66:

\[ \text{ArithmeticError}(r) :- \text{DivExp}(r, v1, v2), \]
\[ \text{IntVar}(v2, 12), \]
\[ \text{isMaybeZero}(12). \]

Here, the filter function isMaybeZero selects only those parity elements 12 that may be zero, i.e. the Parity.Even and Parity.Top elements.

This example demonstrates three key features of FLIX: the ability to express lattices, monotone filter functions that select a subset of lattice elements, and monotone transfer functions that express mappings between lattice elements. These features go beyond what is possible in Datalog and give FLIX its expressive power.

2.3 Design Choice: Language or Framework?

A natural question to ask is why implement FLIX as a programming language instead of as a framework? First, we want the logic language and the functional language to receive equal treatment. If we had embedded FLIX in a functional language, expressing rules would have suffered, and vice versa. Second, we want full control over the language to ensure that the choices of evaluation strategy and data structures are up to the implementation.

At the same time, we want to ensure interoperability with the JVM, providing access to the existing ecosystem of static analysis frontends and tools. FLIX allows programmers to access JVM types and methods, and use them in their definitions of lattices and functions. Furthermore, FLIX provides an API for accessing computed solutions.

3. Semantics

In this section, we present the model-theoretic semantics of Datalog, and then extend that semantics to FLIX. To do so, we abstract away some details of the full FLIX language.

3.1 Model-Theoretic Semantics of Datalog

Syntax. A Datalog program \( P \) is a set of rules of the form \( A_0 \leftarrow A_1, \ldots, A_n \), where \( A_0 \) is the head of the rule, \( A_1, \ldots, A_n \) is the body of the rule, and each \( A_i \) is an atom. A fact is a rule with an empty body. An atom has the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol and \( t_1, \ldots, t_n \) are terms. A term is either a variable \( x \) or a constant value \( v \). Values are typically primitive integers and strings. Datalog has no operations on primitive values nor any compound datatypes. Figure 3 shows the syntax of Datalog programs.

Herbrand Universe and Base. We now briefly describe the Herbrand interpretation of a Datalog program. This is a simple and elegant approach, commonly used to define the meaning of Datalog programs [9]. The Herbrand universe \( \mathcal{U} \) of a Datalog program \( P \) is the set of all possible ground terms. A ground term is a non-variable term, i.e. a constant value appearing somewhere in the program \( P \). The Herbrand universe is finite since the program \( P \) contains a finite number of constants. The Herbrand base \( \mathcal{B} \) of a Datalog program \( P \) is the set of all ground atoms. A ground atom is a predicate symbol that occurs in \( P \) with its arguments drawn from the Herbrand universe. Note that the Herbrand base respects the arity of the predicates. The Herbrand base is also finite. As an example, given the Datalog program

\[ A(1). \quad B(2, 3). \quad A(x) :- B(x, _). \]

the Herbrand universe \( \mathcal{U} \) is

\[ \mathcal{U} = \{1, 2, 3\} \]

and the Herbrand base \( \mathcal{B} \) is

\[ \mathcal{B} = \begin{pmatrix} A(1), & A(2), & A(3), \\ B(1, 1), & B(1, 2), & B(1, 3), \\ B(2, 1), & B(2, 2), & B(2, 3), \\ B(3, 1), & B(3, 2), & B(3, 3) \end{pmatrix} \]

Intuitively, the Herbrand base can be thought of as the set of “all possible facts.”

Interpretations and Models. An interpretation \( I \) of a Datalog program \( P \) is a subset of the Herbrand base \( \mathcal{B} \). A ground atom \( A \) is true w.r.t. an interpretation \( I \) if \( A \in I \). A conjunction of atoms \( A_1, \ldots, A_n \) is true w.r.t. an interpretation if each atom is true in the interpretation. A ground rule is true if either the body conjunction is false or the head is true. A ground rule is a rule where all atoms are ground.

A model \( M \) of a Datalog program \( P \) is an interpretation, i.e. a subset of the Herbrand base \( \mathcal{B} \), that makes each ground
We now extend the Datalog semantics to F

We extend the values of Datalog with enums (tagged unions),

The minimal model of 

This is desirable because it separates the definition of

their own lattices and operations on them. In order to do that,

set of facts that can be derived from

can think of

I

T

iteration of

p

Ground rule instance that is satisfied by

I

T

interpretation of a Datalog program

into how a solver can be implemented. Let

agreeing on the solution.

log solvers can use different evaluation strategies while still

how

the solution is from

P

Computing the Minimal Model. The model-theoretic se-

mantics does not tell us how to compute the minimal model.

This is desirable because it separates the definition of what

the solution is from how to compute it. Thus, different Data-

log solvers can use different evaluation strategies while still

agreeing on the solution.

The model-theoretic semantics does provide some insight

into how a solver can be implemented. Let I be an inter-

pretation of a Datalog program P. We define the immediate

consequence operator T_p of I as the head atoms of each

ground rule instance that is satisfied by I. In other words, we

can think of I as the current set of facts, and T_p(I) as the set

of facts that can be derived from I and P in “one-step.”

The minimal model of P can then be computed by repeated

iteration of T_p(I) starting from the empty set of facts, i.e. it is

the least fixed point of T_p^∞(Ø). This strategy is called “naïve”

evaluation since it re-evaluates every rule whenever a new

fact is inferred. A better strategy is discussed in Section 3.7.

3.2 From Datalog to FLIX

We now extend the Datalog semantics to FLIX.

A complete lattice ℓ is a 6-tuple ℓ = (E, ⊥, ⊤, ∈, ⊑, ∩),

where E is a set of elements, ⊥ ∈ E is the least element,

⊤ ∈ E is the greatest element, ∈ is the partial order on E, ⊑

is the least upper bound, and ∩ is the greatest lower bound.

We want to allow static analysis implementors to express

their own lattices and operations on them. In order to do that,

we extend the values of Datalog with enums (tagged unions),
tuples, and sets. Furthermore, we add a pure functional
programming language. For the purpose of this paper, this
language can be thought of as a minimal lambda calculus,
but the full FLIX language is richer. It includes algebraic
data types, pattern matching, collections (e.g. lists, sets, and
maps), and a static type system.

We make six changes to the Datalog semantics to extend

it with lattices. In brief, the steps are:

1. Associate every predicate symbol with a lattice.
2. Extend the Herbrand base into cells and introduce a

complete lattice on all ground atoms in each cell.
3. Partition the Herbrand universe with lattice elements.
4. Introduce compactness of an interpretation.
5. Extend the definition of a model to incorporate the partial

order on ground atoms.
6. Introduce a partial order on models and update the defini-

tion of minimality.

We now describe each step in greater detail.

First, we change the grammar of Datalog programs to

associate every predicate symbol with a lattice ℓ. Thus, for

example, a fact and a rule now look like

\[ A_ℓ(0dd). \quad B_ℓ(x) : – A_ℓ(x). \]

where ℓ refers to a lattice, in this case, the parity lattice.

Second, we change the definition of the Herbrand uni-

verse U to include all possible ground terms T and lattice

elements E of the FLIX program P. The Herbrand base B

remains the same, but uses the new definition of the Herbrand

universe U. Notice that the Herbrand universe (and conse-

quently the Herbrand base) may be infinite if a lattice has

infinitely many elements.

Third, we partition the Herbrand base such that two ground

atoms A = p_ℓ(v_1, ..., v_n) and B = p_ℓ'(v_1', ..., v_m') are in

the same cell S if they have the same predicate symbol (i.e.

p_ℓ = p_ℓ'), the same number of terms (i.e. n = m), and the

first n terms are equal (i.e. v_1 = v_1' ∧ ⋯ ∧ v_(n-1) = v_(n-1)'). Notice that all predicate symbols in the same cell have

the same associated lattice ℓ. For each cell S, we introduce a

complete lattice L_S = (S, ⊥S, ⊤S, ∈S, ⊑S, ∩S). Given two

ground atoms A = p_ℓ(v_1, ..., v_n) and B = p_ℓ'(v_1', ..., v_m'),

we define their partial order as follows: if n = 1 then A ∈ S B

when v_1 ⊆ v_1'. Otherwise, if n > 1 then A ⊆ S B when

v_1 = v_1' ∧ ⋯ ∧ v_(n-1) = v_(n-1)' and v_n ⊆ v_n'. In words,

in order for A ⊆ S B, the first n − 1 components of A and

B must match exactly, and the last component of A must

be less than or equal to the last component of B according
to the partial order defined for the lattice ℓ. The definition

of the remaining lattice components is straightforward.

The intuition is that unary predicates correspond to a single lattice

element whereas multi-ary predicates correspond to a single
tuple from a map lattice. To ensure a finite number of cells,

we restrict the terms that can appear in a ground atom. Recall

\[
P \in \text{Prog} ::= R_1, \ldots, R_n
\]

\[
R \in \text{Rules} ::= A_0 \leftarrow A_1, \ldots, A_n
\]

\[
A \in \text{Atoms} = p(t_1, \ldots, t_n)
\]

\[
t \in \text{Terms} = x | v
\]

\[
p \in \text{Predicates} = \text{is a finite set of predicate symbols.}
\]

\[
x \in \text{Variables} = \text{is a finite set of variables.}
\]

\[
v \in \text{Values} = \text{is a finite set of values.}
\]

Figure 3. Grammar of Datalog programs.
that the Herbrand universe \( \mathcal{U} = \mathcal{T} \cup \mathcal{E} \) consists of ground terms \( \mathcal{T} \) and lattice elements \( \mathcal{E} \). We require that in every ground atom \( p_t(v_1, \ldots, v_n) \), the values \( v_1, \ldots, v_{n-1} \) must be drawn from \( \mathcal{T} \), and \( v_n \) must be drawn from \( \mathcal{E} \). Since \( \mathcal{T} \) consists of terms that syntactically appear in the program \( P \), it is finite and consequently the number of cells is finite.

Fourth, we introduce the notion of compactness. An interpretation \( I \) is compact iff every cell \( S \) in the partition of \( I \) has one unique element.

Fifth, an interpretation is still a subset of the Herbrand base, but we change the definition of when an interpretation is a model. Specifically, a ground atom \( A \) is true w.r.t. an interpretation \( I \) if there exists \( A' \in I \) such that \( A \equiv_s A' \), where \( A \) and \( A' \) are in the same cell \( S \). As before, a conjunction of atoms \( A_1, \ldots, A_n \) is true w.r.t. an interpretation if each atom is true in the interpretation. Finally, a ground rule is true if either the body conjunction is false or the head is true. A model \( M \) of \( P \) is then an interpretation that makes each ground instance of each rule in \( P \) true.

Sixth, we define a partial order \( \equiv_M \) on models. Given two models \( M_1 \) and \( M_2 \), we say that \( M_1 \) is less than or equal to \( M_2 \) if for every ground atom \( A_1 \in M_1 \), belonging to cell \( S \), there is a ground atom \( A_2 \in M_2 \), also belonging to \( S \), such that \( A_1 \equiv_s A_2 \). A model \( M \) is minimal if it is compact and there is no other model \( M' \) such that \( M' \equiv_M M \).

Example. The FLIX program defined over the parity lattice \( \ell = \{\bot, \top, \text{Even}, \text{Odd}\} \) with the facts

\[
A_\ell(\text{Even}), A_\ell(\text{Odd}), B_\ell(\text{Odd}).
\]

has the Herbrand universe

\[
\mathcal{U} = \{\bot, \top, \text{Even}, \text{Odd}\}
\]

and the Herbrand base

\[
B = \{A_\ell(\bot), A_\ell(\text{Even}), A_\ell(\text{Odd}), A_\ell(\top), B_\ell(\bot), B_\ell(\text{Even}), B_\ell(\text{Odd}), B_\ell(\top)\}
\]

An interpretation of the program is a subset of \( B \), e.g.: 

\[
I_1 = \{A_\ell(\top)\} \\
I_2 = \{A_\ell(\top), B_\ell(\bot)\} \\
I_3 = \{A_\ell(\top), B_\ell(\text{Odd}), B_\ell(\top)\} \\
I_4 = \{A_\ell(\text{Even}), A_\ell(\text{Odd}), B_\ell(\text{Odd})\} \\
I_5 = \{A_\ell(\top), B_\ell(\top)\} \\
I_6 = \{A_\ell(\top), B_\ell(\text{Odd})\}
\]

The interpretations \( I_1 \) and \( I_2 \) are not models of the program since neither makes \( B_\ell(\text{Odd}) \) true. \( I_3 \) and \( I_4 \) are models, but they are not compact. \( I_5 \) is a compact model, but it is not minimal as evidenced by the true minimal model \( I_6 \).

Example. The FLIX program defined over the sign lattice \( \ell = \{\bot, \top, \text{Neg}, \text{Zer}, \text{Pos}\} \) with the facts

\[
A_\ell(\text{Pos}), A_\ell(2, \text{Pos}), A_\ell(2, \text{Neg}).
\]

has the Herbrand universe

\[
\mathcal{U} = \{1, 2, \bot, \top, \text{Neg}, \text{Zer}, \text{Pos}\}
\]

and the Herbrand base

\[
B = \{A_\ell(1, \bot), A_\ell(1, \text{Neg}), A_\ell(1, \text{Zer}), A_\ell(1, \top)\}
\]

An interpretation of the program is a subset of \( B \), e.g.: 

\[
I_1 = \{A_\ell(1, \top)\} \\
I_2 = \{A_\ell(1, \bot), A_\ell(1, \text{Neg}), A_\ell(2, \top)\} \\
I_3 = \{A_\ell(1, \top), A_\ell(2, \top)\} \\
I_4 = \{A_\ell(1, \bot), A_\ell(2, \top)\}
\]

Here, \( I_2, I_3 \) and \( I_4 \) are models, \( I_3 \) and \( I_4 \) are compact, but only \( I_4 \) is minimal.

Least Upper and Greatest Lower Bounds. To clarify how lattices are used in FLIX, suppose that we have the two facts:

\[
A(\text{Odd}). \\
B(\text{Even}).
\]

Then the FLIX program with the two rules

\[
R(x) :~ A(x). \\
R(x) :~ B(x).
\]

has \( R(\top) \) in the minimal model since \( \top \) is the only element greater than or equal to both \( \text{Odd} \) and \( \text{Even} \). On the other hand, the FLIX program with the one rule

\[
R(x) :~ A(x), B(x).
\]

has \( R(\bot) \) in the minimal model since \( \bot \) is the only element less than or equal to both \( \text{Odd} \) and \( \text{Even} \).

Computing the Minimal Model. We have described how to compute the minimal model of a Datalog program as the least fixed point of the immediate consequence operator \( T_p \). We can adopt a similar strategy for FLIX programs with one change: the domain of the operator is restricted to compact interpretations, and since the set of one-step derivable facts may not be compact, we must compute the least upper bound in every cell of the interpretation before we apply \( T_p \) again.

Ullman [59, Chapter 3] presents a naive evaluation algorithm for Datalog with a detailed proof that it terminates and computes the minimal model. The proof relies on two key properties: monotonicity of the immediate consequence operator and finiteness of ascending chains of interpretations. By being careful to maintain monotonicity in defining the FLIX version of the immediate consequence operator, and by insisting that the FLIX lattices be of finite height, we can apply the same proof to a naive evaluation algorithm for FLIX.
3.3 The Full FLIX Language

The FLIX language is richer than the core formalism presented in the previous subsection. In this subsection, we describe two important extensions: monotone filter functions and monotone transfer functions. Without these features, programming in FLIX would be severely limited. Monotonicity of these functions is necessary to maintain the monotonicity of the immediate consequence operator, which in turn is necessary for the correctness of the naïve evaluation algorithm.

Monotone Filter Functions. A monotone filter function is a function from one or more lattice elements to true or false, and is monotone when the booleans are ordered false < true. For example, assume that isMaybeZero is a function over the parity lattice elements. Then the rule

\[
\text{ArithmeticError}(r) :- \quad \text{DivExp}(r, v1, v2), \\
\text{IntVar}(v2, i2), \quad \text{isMaybeZero}(i2).
\]

captures a possible division-by-zero error for result variable \( r \) when there is an expression \( r = v1 \div v2 \), the value of variable \( v2 \) is the parity lattice element \( i2 \), and \( i2 \) is possibly zero according to the filter function. The important point here is that the DivExp relation and IntVar lattice are both explicitly represented and tabulated, whereas isMaybeZero is a reference to a function expression, i.e. a piece of code. Thus, filter functions may operate over domains that are too difficult or even impossible to tabulate.

We add filter functions to the model-theoretic semantics with the following changes:

1. Extend the definition of a rule to allow filter function applications \( f_i(\ldots) \) in addition to atoms.
2. Extend the definition of when a ground rule instance to allow filter function applications in the last term of a head predicate.
3. Extend the definition of when a ground atom is true.

A ground atom \( A \) with function applications is true iff after all function applications are evaluated, the resulting ground atom is true.

Monotone Functions. A monotone function is a function from one or more lattice elements to a lattice element, which is order-preserving. These functions are used to implement “transfer” functions in FLIX. For example, in the rule

\[
\text{IntVar}(r, \text{sum}(i1, i2)) :- \quad \text{AddExp}(r, v1, v2), \\
\text{IntVar}(v1, 1), \quad \text{IntVar}(v2, i2).
\]

the monotone function \( \text{sum} \) computes the abstract addition of two parity lattice elements \( 1 \) and \( i2 \). We highlight two important design choices here.

First, we require such transfer functions to be strict and monotone. Strictness ensures that when a function is applied to \( \bot \) it returns \( \bot \). Monotonicity requires that the function is order-preserving and is necessary to ensure that the immediate consequence operator is also monotone.

Second, we only allow non-filter functions to appear in the last term of the head predicate of a rule. This ensures that the implementation can evaluate the rule body first, in any order, and then evaluate the head. If non-filter functions could appear in the body, then one could write

\[
R(z) :- A(x), B(y, f(x)), C(z, g(y)). \\
R(x) :- A(y, f(x)), B(z, g(y)), C(x, h(z)).
\]

where the first rule implicitly enforces an evaluation order\(^1\). Moreover, it is not clear how to evaluate the second rule. Thus, we disallow non-filter functions in rule bodies.

We add monotone “transfer” functions to the model-theoretic semantics with the following changes:

1. Extend the definition of a rule to allow the last term of a head predicate to contain function applications \( f_i(\ldots) \).
2. Extend the definition of a ground rule instance to allow function applications in the last term of a head predicate.
3. Extend the definition of when a ground atom is true.

A ground atom \( A \) with function applications is true iff after all function applications are evaluated, the resulting ground atom is true.

3.4 Compositionality of FLIX Programs

An important property of Datalog programs is compositionality: given two disjoint Datalog programs \( P_1 \) and \( P_2 \), i.e. two programs that share no predicate symbols, the union of their rules is a Datalog program \( P \). The model of \( P \) is the union of the models of \( P_1 \) and \( P_2 \). FLIX retains this property, allowing composition of analyses. For example, given a constant propagation analysis and a reachability analysis, we can combine the two to obtain a conditional constant propagation analysis.

We share information between the two analyses by introducing the shared predicates \( \text{isReachable}(s), \text{isTrue}(s), \text{isFalse}(s) \). The constant propagation analysis infers facts for the \( \text{isTrue}(s) \) and \( \text{isFalse}(s) \) predicates and uses the \( \text{isReachable}(s) \) predicate, while the reachability analysis does the opposite.

In abstract interpretation terminology, the compositionality discussed above is known as the direct product [14]. Given two analyses, e.g. sign and parity, the direct product corresponds to running each analysis independently. No information is shared and neither lattice is used to refine the other. For example, the element \((2\text{er}, 0\text{dd})\) in the Cartesian product \( \text{Sign} \times \text{Parity} \) does not correspond to any concrete value and could be replaced by \((\bot, \bot)\). The reduced product uses this idea to refine and share information between lattices [11, 14]. The logical product is more precise than the reduced product, and under certain conditions can be constructed automatically [29]. FLIX provides the direct product automatically, but the reduced and logical products must be implemented manually.

---

\(^1\) Since the functions \( f \) and \( g \) are opaque and may not have an inverse.
3.5 Negation, Stable Models, and Stratification

Pure Datalog does not allow negation; every atom in the body of a rule must appear unnegated. Many proposals have been made to extend logic and Datalog programs with negation [20, 21, 23, 26, 36]. A fundamental challenge is how to define the semantics of programs such as:

\[ A(x) \leftarrow \neg B(x), \quad B(x) \leftarrow \neg A(x). \]

To overcome these issues, several solutions including stable models [25], well-founded semantics [61], and many-valued logic [23] have been developed. A common solution is to restrict the use of negation to so-called stratifiable programs [2]. A program is stratifiable if it satisfies a simple syntactic property which ensures that no negative cycles occur in the program. FLIX currently does not support any form of negation, but it is something we plan to add.

3.6 The Theoretical Expressive Power of Datalog

We have previously stated that certain static analyses are “inexpressible” in pure Datalog. Strictly speaking, this is not true: a well-known result from logic and complexity theory shows that the data complexity of Datalog is PTIME-complete [17]. The data complexity of a program is the complexity when the input is restricted to facts (i.e. the rules are kept fixed) and PTIME is the class of all polynomial time algorithms. Intuitively, any algorithm which runs in polynomial time can be encoded as a Datalog program. Naturally, this includes polynomial time static analyses.

However, this encoding simulates a Random Access Machine that tabulates all steps taken by the algorithm, and furthermore, requires operations such as integer arithmetic to be represented as finite relations. While this is sufficient for an existence proof, such tabulation is completely impractical for an implementation.

3.7 The Semi-naïve Evaluation Strategy

The model-theoretic semantics of FLIX describes the structure of the minimal model of a program, but not how to compute it. This is a good thing, since it gives the fixed-point solver maximum freedom in the choice of data structures and evaluation strategy. As explained before, the model-theoretic semantics inspires a naïve evaluation strategy based on the immediate consequence operator. The idea is to repeatedly re-evaluate every rule while maintaining a monotonically growing set of facts. While this strategy is simple and correct, it is hopelessly inefficient.

A better strategy, known as semi-naïve evaluation, tracks the dependencies between predicates and predicate symbols appearing in rule bodies. Under this evaluation strategy, whenever a new fact for a predicate \( p \) is inferred, only the rules containing \( p \) in their body are re-evaluated, and only for the new fact. For example, in the FLIX program

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

if the fact \( \text{Path}(1, 2) \) is inferred, then only the third rule is re-evaluated under the environment \( \{ x \mapsto 1, y \mapsto 2 \} \). The first and second rules are not re-evaluated. With naïve evaluation, all three rules would be re-evaluated.

We use a variant of semi-naïve evaluation, adapted for FLIX by taking the compactness requirement into account. For example, in the FLIX program

\[ A(\text{odd}), \]
\[ B(\text{even}). \]
\[ A(x) \leftarrow B(x), \]
\[ B(x) \leftarrow \neg A(x). \]

we initially infer the two facts \( A(\text{odd}) \) and \( B(\text{even}) \), which cause evaluation of the two rules. This infers the new fact \( A(\text{even}) \). Next, we must re-evaluate the third rule, but we must not use the environment \( \{ x \mapsto \text{even} \} \) since this breaks the compactness requirement. Instead, we must compute the least upper bound of \( A(\text{odd}) \cup A(\text{even}) = A(\top) \), and re-evaluate the third rule under the environment \( \{ x \mapsto \top \} \). This gives the correct minimal model \( M \) where \( B(\top) \in M \).

In addition to naïve evaluation, Ullman [59, Chapter 3] also defines and proves correctness of a semi-naïve evaluation algorithm for Datalog. The semi-naïve algorithm maintains a so-called incremental relation for each predicate. Whenever the algorithm evaluates a rule with head predicate \( p_i \) to yield a new relation \( P_i' \), the incremental relation is computed as the set difference \( \Delta P_i = P_i' \setminus P_i \), where \( P_i \) is the old relation for the predicate \( p_i \). The incremental relation contains only the facts that were newly computed in \( P_i' \), that were not already contained in the old relation \( P_i \).

Then, the process of evaluating a rule is modified to the following procedure. Letting \( n \) be the number of atoms in the body of the rule, the rule is evaluated \( n \) times. Each time, one of the \( n \) atoms is instantiated using the incremental relation \( \Delta P_i \) corresponding to the predicate specified by the atom. All other atoms of the body of the rule are instantiated using their corresponding old relations \( P_i \). Informally, this ensures that each newly-inferred fact in \( \Delta P_i \) is considered with all of the existing facts in the old relations for the other atoms. Ullman proves that each such incremental evaluation step yields the same resulting relation for the head predicate as the corresponding full evaluation step in the naïve algorithm. Furthermore, he proves inductively that every iteration of the semi-naïve algorithm infers the same facts as the corresponding iteration of the naïve algorithm (but using less work), and therefore the final outputs of the two algorithms are the same.

To adapt semi-naïve evaluation to FLIX, we must first adapt the definition of the incremental relation. In Datalog, \( P_i' \supseteq P_i \) by the monotonicity of rule evaluation, so \( \Delta P_i = P_i' \setminus P_i \). This latter property is important because each step of semi-naïve evaluation can be informally thought of as determining \( \Delta P_i \cup P_i \), while the corresponding step of naïve evaluation computes \( P_i' \). However, in FLIX, it is not generally true that \( P_i' \supseteq P_i \) because rule evaluation applies a least upper bound in each cell \( S \) to obtain a compact
We have implemented a compiler and runtime for FLIX. An alternative definition of the incremental relation \( \Delta P_i \) is therefore necessary so that \( P'_i = \Delta P_i \cup P_i \), analogously to the Datalog property \( P'_i = P_i \). We have defined a set of facts to be compact if it contains one ground atom for each cell \( S \); let us denote this ground atom \( ga(P, S) \).

When \( P'_i \) and \( P_i \) are the relations for a predicate before and after evaluating a rule, we define their incremental relation \( \Delta P_i = \{ ga(P'_i, S) | S \in cells \land ga(P'_i, S) \not\subset ga(P_i, S) \} \). In words, the incremental relation \( \Delta P_i \) contains every ground atom from \( P'_i \) which is strictly greater than the ground atom for the same cell in \( P_i \). Since \( P'_i \subseteq P_i \), this definition of \( \Delta P_i \) does imply the desired property \( P'_i = \Delta P_i \cup P_i \).

The process of evaluating a rule can then be incrementalized in the same way as for Datalog, in that the rule is evaluated as many times as there are atoms in its body, and each time, one of the atoms is instantiated with the FLIX version of the incremental relation \( \Delta P_i \). The resulting relation for the head predicate is then made compact by computing a least upper bound for each cell \( S \). We can show that this incremental rule evaluation step yields the same resulting relation for the head predicate as the corresponding full evaluation step, in the same way as the analogous incremental evaluation step does in Datalog. Therefore, the inductive proof of the equivalence of outputs of naïve and semi-naïve evaluation for Datalog then applies analogously to FLIX.

4. Evaluation

We have implemented a compiler and runtime for FLIX. The entire implementation is roughly 30,000 lines of Scala code. The toolchain includes a parser, a type checker, an interpreter, an indexed database, and a semi-naïve fixed-point solver. The source code is freely available on GitHub.²

We evaluate the usefulness of FLIX by implementing three existing static analyses: the Strong Update analysis [39] and the IFDS and IDE algorithms [52, 53].

4.1 The Strong Update Analysis

The Strong Update analysis is a points-to analysis for C programs that propagates singleton sets to sets flow sensitively and larger sets flow insensitively. The paper presents the analysis as a set of constraints (Figure 7 in that paper) and then as an imperative algorithm (Figure 9 in that paper). The latter is implemented in C++ and evaluated [39]. The FLIX implementation shown in Figure 4 follows the constraint specification of the analysis directly: there is a one-to-one correspondence between the FLIX rules and the constraints from Figure 7 of the Strong Update paper. The PtSU function is defined as:

\[
\text{PtSU}(\overline{x})(a) \triangleq \begin{cases} 
\text{su}(\overline{x})(a) & \text{if } \text{su}(\overline{x})(a) \neq \top \\
pt(a) & \text{if } \text{su}(\overline{x})(a) = \top 
\end{cases}
\]

The monotonicity of this definition depends on the unstated fact that \( \text{su}(\overline{x})(a) \subseteq pt(a) \) when \( \text{su}(\overline{x})(a) \) is a singleton set.

2 https://github.com/fllix

In the corresponding FLIX rule, monotonicity is explicit: the rule first selects all \( b \in pt(a) \), and then uses the function \( \text{filter} \) to reject all elements other than \( p \) in the case that \( \text{su}(\overline{x})(a) \) is the singleton \( \{p\} \).

4.2 IFDS

IFDS is a framework that can be instantiated to solve a large class of interprocedural context-sensitive dataflow analyses, the interprocedural finite distributive subset analyses [52]. The paper that defines the framework presents it as a one-page algorithm in pseudocode that contains many worklist updates and implicit quantifications. Anecdotaly, many people find the algorithm difficult to understand, and checking its correctness requires a long proof.

Figure 5 declaratively specifies the desired properties of an IFDS solution. It is also a set of FLIX rules that can be executed to compute the solution. The rules compute a set of path edges, each leading from a data point \( d_1 \) at the start of a procedure to a data point \( d_2 \) at an instruction \( m \) within the procedure, and a set of summary edges that summarize the transfer function of each call to a procedure. To implement a specific dataflow analysis, one must provide the transfer functions of that analysis. These are specified to FLIX in the form of three functions \( \text{eshIntra} \) (intraprocedural),

```java
enum SU lattice {
    case Top,
    case Single(Str),
    case Bottom
}
def filter(t: SU lattice, b: Str): Bool =
    match t with {
        case SU lattice.Bottom => false
        case SU lattice.Single(p) => b == p
        case SU lattice.Top => true
    }
Pt(p, a) :- AddrOf(p, a).
Pt(p, a) :- Copy(p, q), Pt(q, a).
Pt((p, b) :- Load(p, q), Pt(q, a), PtSU(1, a, b).
PtH(a, b) :- Store(1, p, q), Pt(p, a), Pt(q, b).
SUBefore(l2, a, t) :-
    CFG(l1, l2),
    SUAfter(l1, a, t).
SUAfter(l1, a, t) :-
    SUBefore(l1, a, t),
    Preserve(l1, a).
SUAfter(l1, a, SU lattice.Single(b)) :-
    Store(l1, p, q),
    Pt(p, a), Pt(q, b).
PtSU(1, a, b) :-
    PtH(a, b),
    SUBefore(l1, a, t),
    filter(t, b).
```

Figure 4. FLIX version of the Strong Update analysis [39]. SUBBefore and SUAfter is the information before and after label \( l \), written \( \text{ptSU}(l)[a] \) and \( \text{ptSU}(l)[a] \) in the paper. Preserve is the complement of the Kill set.
PathEdge(d1, m, d3) :-
    CFG(n, m),
    PathEdge(d1, n, d2),
    d3 ← eshIntra(n, d2).
PathEdge(d1, m, d3) :-
    CFG(n, m),
    PathEdge(d1, n, d2),
    SummaryEdge(n, d2, d3).
PathEdge(d3, start, d3) :-
    PathEdge(d1, call, d2),
    CallGraph(call, target),
    EshCallStart(call, d2, target, d3),
    StartNode(target, start).
SummaryEdge(call, d4, d5) :-
    CallGraph(call, target),
    EndNode(target, end),
    EshCallStart(call, d4, target, d1),
    PathEdge(d1, end, d2),
    d5 ← eshEndReturn(target, d2, call).
EshCallStart(call, d, target, d2) :-
    PathEdge(_, call, d),
    CallGraph(call, target),
    d2 ← eshIntra(call, d, target).
Result(n, d2) :-
    PathEdge(_, n, d2).

Figure 5. FLIX implementation of the IFDS analysis [52].

eshCallStart (call site to start node), and eshEndReturn (end node to return site). The special arrow syntax in the first rule d3 ← eshIntra(n, d2) binds d3 to each element in the set returned by the call.

It is essential that the transfer functions be specified as functions; they cannot be tabulated and given as input to any solver in the form of relations. The reason is that the overall goal of the IFDS algorithm is to compute a set of reachable pairs (n, d). If we knew the set of arguments (n, d) for which eshIntra needs to be tabulated (the set of all arguments with which the analysis will call eshIntra), then we would already know which pairs are reachable and there would be no need to run the IFDS analysis. Alternatively, tabulating eshIntra for all possible pairs (n, d) would be much more costly than performing the IFDS algorithm itself, which calls eshIntra only on the much smaller subset of pairs (n, d) that are reachable.

Another important detail not discussed in the IFDS paper is that the algorithm applies not only the eshCallStart function (in the third rule) but also its inverse (in the fourth, SummaryEdge rule). Since computing the inverse is usually impractical, any implementation of the algorithm must tabulate the function for the arguments on which it is called in the forward direction. The FLIX program explicitly tabulates the function in the EshCallStart relation. As a result, the relation can be consulted in both directions in the third and fourth rule, and this can be written declaratively as it is in the original formulation of the IFDS algorithm.

Both of these issues are discussed by Naeem et al. [48].

JumpFn(d1, m, d3, comp(long, short)) :-
    CFG(n, m),
    JumpFn(d1, n, d2, long),
    (d3, short) ← eshIntra(n, d2).
JumpFn(d1, m, d3, comp(caller, summary)) :-
    CFG(n, m),
    JumpFn(d1, n, d2, caller),
    SummaryFn(n, d2, d3, summary).
JumpFn(d3, start, d3, identity()) :-
    JumpFn(d1, call, d2, _),
    CallGraph(call, target),
    EshCallStart(call, d2, target, d3),
    StartNode(target, start),
    SummaryFn(call, d4, d5, comp(comp(cs, se), er)) :-
    CallGraph(call, target),
    StartNode(target, start),
    EndNode(target, end),
    EshCallStart(call, d4, target, d1),
    JumpFn(d1, end, d2, se),
    (d5, er) ← eshEndReturn(target, d2, call).
EshCallStart(call, d, target, d2, cs) :-
    JumpFn(_, call, d, _),
    CallGraph(call, target),
    (d2, cs) ← eshCallStart(call, d, target).
InProc(p, start) :- StartNode(p, start).
InProc(p, m) :- InProc(p, n), CFG(n, m).

Result(n, d, apply(fn, vp)) :-
    ResultProc(proc, dp, vp),
    InProc(proc, n),
    JumpFn(dp, n, d, fn).
ResultProc(proc, dp, apply(cs, v)) :-
    Result(call, d, v),
    EshCallStart(call, d, proc, dp, cs).

Figure 6. FLIX implementation of the IDE analysis [53].

4.3 IDE

IDE is a more general framework that can be instantiated to solve a larger class of interprocedural context-sensitive dataflow analyses, the interprocedural distributive environment analyses [53]. The original presentation of IDE as an imperative algorithm requires two pages. Conceptually, the IDE framework is a direct extension of the IFDS framework, but that is not obvious at all from the worklist-based algorithmic specifications in the two papers. The IDE framework computes the same edges as IFDS, but each edge is decorated with a representation of a so-called micro-function. This correspondence between IFDS and IDE is immediately clear from the declarative specification of IDE shown in Figure 6. Notice that the rules mirror those of the IFDS implementation (with the names PathEdge and SummaryEdge replaced with JumpFn and SummaryFn to match the terminology used in the IFDS and IDE papers). In the IDE algorithm, each rule has just one additional component corresponding to the micro-function on each edge. The first, second, and fourth rules use a FLIX function comp in the head to compute the composition of micro-functions.
We have not yet looked closely at the performance of the IDE framework with a specific analysis, one must not only implement the transfer functions \texttt{eshIntra}, \texttt{eshCallStart}, and \texttt{eshEndReturn}, but also specify two lattices: the value lattice \( V \) that is the domain and range of each micro-function, and the micro-function lattice \( F \) that efficiently represents certain functions from \( V \to V \). Thus, the formulation of the IDE algorithm requires both functions and lattices.

The IDE paper uses a running example of a linear constant propagation analysis in which \( V \) is the constant propagation lattice. The elements of the micro-function lattice \( F \) are \( \lambda l . (a \times l + b) \sqcup c \), where \( a \) and \( b \) are integers and \( c \) is an element of the constant propagation lattice. Figure 7 shows the FLIX implementation of the micro-function composition operation \texttt{comp} that is called from the rules. The functions that implement \( \sqsubseteq \) and \( \sqcup \) on the lattice of micro-functions have a similar structure.

To instantiate the IDE framework with a specific analysis, one must not only implement the transfer functions \texttt{eshIntra}, \texttt{eshCallStart}, and \texttt{eshEndReturn}, but also specify two lattices: the value lattice \( V \) that is the domain and range of each micro-function, and the micro-function lattice \( F \) that efficiently represents certain functions from \( V \to V \). Thus, the formulation of the IDE algorithm requires both functions and lattices.

The IDE paper uses a running example of a linear constant propagation analysis in which \( V \) is the constant propagation lattice. The elements of the micro-function lattice \( F \) are \( \lambda l . (a \times l + b) \sqcup c \), where \( a \) and \( b \) are integers and \( c \) is an element of the constant propagation lattice. Figure 7 shows the FLIX implementation of the micro-function composition operation \texttt{comp} that is called from the rules. The functions that implement \( \sqsubseteq \) and \( \sqcup \) on the lattice of micro-functions have a similar structure.

4.4 Shortest Paths

We have shown how to express several static analyses in FLIX, but FLIX is applicable to other types of fixed-point problems. For example, to compute all-pairs shortest paths, let \((N, \preceq, 0, \geq, \min, \max)\) be a lattice over the natural numbers. Then we can compute the shortest paths as follows:

\[
\text{Dist}(y, d + c) \leftarrow \text{Dist}(x, d), \text{Edge}(x, y, c).
\]

4.5 Performance of the Current FLIX Solver

We have not yet looked closely at the performance of the FLIX solver. Instead, we have focused on language design and semantics. For that reason, the solver contains many inefficiencies that can be overcome with additional engineering. For example, primitive values (e.g., integers) are represented as (boxed) Java objects, functions (including the partial order and least upper bound) are evaluated using an AST-based interpreter, rules are always evaluated left-to-right instead of using a cost-plan, relations are represented as hash maps but some would be more efficiently represented as dense arrays, and our index selection strategy is not optimal.

Table 1 compares the performance of three implementations of the Strong Update analysis on the benchmark programs evaluated in the Strong Update paper. First, we implemented the Strong Update analysis in Datalog by embedding the Strong Update lattice within the relational powerset lattice as described in the introduction. We used the well-known DLV Datalog solver [38] to run this implementation.

The running times of the two implementations on the six DaCapo benchmarks [6] that were evaluated in the original paper are shown in Table 2. In general, the current FLIX im-

```python
def comp(t1: Transformer, t2: Transformer): Transformer = match (t1, t2) with {
    case (_, BotTransformer) => BotTransformer
    case (BotTransformer, NonBotTransformer(a, b, Value.Bot)) => BotTransformer
    case (BotTransformer, NonBotTransformer(a, b, Value.Cst(k))) => NonBotTransformer(0, k, Value.Cst(k))
    case (BotTransformer, NonBotTransformer(a, b, Value.Top)) => NonBotTransformer(0, 0, Value.Top)
    case (NonBotTransformer(a2, b2, c2), NonBotTransformer(a1, b1, c1)) =>
        NonBotTransformer(a1 + a2, (a1 + b2) + b1, lub(prod(c2, a1), b1), c1)
}

Figure 7. Micro-function join operation for the example IDE analysis [53].
```
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DLV</th>
<th>Flix</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Memory (MB)</td>
<td>Time (s)</td>
</tr>
<tr>
<td>470.lbm</td>
<td>1.2</td>
<td>1,205</td>
<td>1.8</td>
</tr>
<tr>
<td>181.mcf</td>
<td>2.5</td>
<td>3,377</td>
<td>114</td>
</tr>
<tr>
<td>429.mcf</td>
<td>2.7</td>
<td>3,392</td>
<td>115</td>
</tr>
<tr>
<td>256.bz2p2</td>
<td>4.7</td>
<td>5,017</td>
<td>49</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>4.4</td>
<td>6,196</td>
<td>215</td>
</tr>
<tr>
<td>164.gzip</td>
<td>8.6</td>
<td>9,259</td>
<td>463</td>
</tr>
<tr>
<td>401.2zip2</td>
<td>8.3</td>
<td>11,844</td>
<td>1,100</td>
</tr>
<tr>
<td>458.byeng</td>
<td>13.9</td>
<td>20,154</td>
<td>1,077</td>
</tr>
<tr>
<td>433.milc</td>
<td>15.0</td>
<td>22,147</td>
<td>-</td>
</tr>
<tr>
<td>175.vpr</td>
<td>17.8</td>
<td>25,977</td>
<td>-</td>
</tr>
<tr>
<td>186.crafty</td>
<td>21.2</td>
<td>32,189</td>
<td>-</td>
</tr>
<tr>
<td>197.parser</td>
<td>11.4</td>
<td>32,606</td>
<td>-</td>
</tr>
<tr>
<td>482.sphinxx3</td>
<td>25.1</td>
<td>42,736</td>
<td>-</td>
</tr>
<tr>
<td>300.twolf</td>
<td>20.5</td>
<td>44,014</td>
<td>-</td>
</tr>
<tr>
<td>456.hmmcr</td>
<td>36.0</td>
<td>68,384</td>
<td>-</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>51.6</td>
<td>89,898</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. Summary of performance results for the Strong Update analysis. Timeout means more than 15 minutes.

<table>
<thead>
<tr>
<th>Program</th>
<th>Scala Time (s)</th>
<th>Flix Time (s)</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>luindex</td>
<td>133.6</td>
<td>366.7</td>
<td>2.7x</td>
</tr>
<tr>
<td>antlr</td>
<td>176.7</td>
<td>437.3</td>
<td>2.5x</td>
</tr>
<tr>
<td>hsqldb</td>
<td>187.4</td>
<td>469.2</td>
<td>2.5x</td>
</tr>
<tr>
<td>bloat</td>
<td>203.5</td>
<td>584.1</td>
<td>2.9x</td>
</tr>
<tr>
<td>pmd</td>
<td>247.7</td>
<td>680.1</td>
<td>2.7x</td>
</tr>
<tr>
<td>jython</td>
<td>4,614.7</td>
<td>14,344.8</td>
<td>3.1x</td>
</tr>
</tbody>
</table>

Table 2. Summary of performance results for IFDS.

Implementation of the IFDS algorithm is about 3x slower than the imperative Scala implementation. Importantly, the performance of FLIX scales with the imperative implementation.

5. Related Work

Static Analysis Frameworks. Many static analysis frameworks have been proposed over the years. The Program Analysis Generator (PAG) generates C code for static analyzers from specifications of lattices and descriptions of transfer functions [44]. The TJ Watson WALA library is a static analysis library written in Java [22]. WALA includes implementations of many common static analyses such as points-to analysis, class hierarchy analysis, and the IFDS algorithm [52]. Soot is a Java bytecode analysis framework that has seen wide use in compilation and as a frontend for other static analyses [60]. Hoopil is a generic dataflow analysis and transformation framework written in Haskell [51]. Frama-C is a source code analysis platform for C programs written in OCaml [16]. A weakness of all these frameworks is the implicit assumptions that they make about the analysis, for instance, the existence of a control-flow graph or other intermediate representation, and the choice of flow-sensitivity, context-sensitivity, and memory abstraction. FLIX, like Datalog, makes no such assumptions and is applicable to least fixed point problems in general.

Datalog. Datalog has roots in the database community as a general-purpose query language, but has found uses in many areas of computer science. A comprehensive introduction to Datalog is given by Ceri et al. [9]. A distilled version is given by Huang et al. who argue that interest in Datalog is re-emerging [32]. Further evidence of this is provided by the Datalog 2.0 Workshop [19].

The use of negation in logic programs, such as Datalog and Prolog, has long been studied and there is extensive literature on the subject [2, 20, 21, 23, 25, 26, 36]. The theoretical complexity of logic programs, including Datalog, has also been studied extensively [17].

Datalog has been used in several static analysis tools. codeQuest is a tool for querying various aspects of the source code of a program [30]. Binary decision diagrams (BDDs) have been used to implement efficient points-to analyses specified as relations, both using a custom relational language and Datalog [40, 63]. The Doop framework is a precise and scalable context-sensitive points-to analysis for Java specified in Datalog [7, 54, 55].

FLIX has some similarities to Bloom, a programming language designed to ensure consistency of distributed programs [13]. Bloom, like FLIX, takes inspiration from Datalog and adds support for lattices and monotone functions, but for different reasons. In Bloom, the purpose is to ensure confluence, i.e. that regardless of the order in which messages are received over the network, the same result is computed. Unlike Datalog and FLIX, where the user is interested in some minimal model, Bloom programs are intended to run continuously as network services.
Logic Programming. Prolog [10, 56] is a Turing-complete logic programming language related to Datalog. Every Datalog program is also a Prolog program; however, Prolog allows constructors, negation, and the cut operator. Datalog programs can be efficiently solved by Prolog engines that use tabulation, e.g., XSB [57]. Such Prolog engines have been used to implement different types of program analyses [18]. FLIX is less expressive than Prolog, but ensures that every program terminates and has a unique minimal model. In future work we plan to compare FLIX to Prolog engines.

Constraint Logic Programming (CLP) schemes extend logic programming with a decidable background theory, such as lists, trees, and linear arithmetic [12, 33, 34, 41]. Intuitively, a CLP program is a set of Horn clauses, each equipped with a formula over the background theory. During evaluation, term unification is augmented with the decision procedure of the theory. CLP has been used in program verification and abstract interpretation [5, 24].

Alternation-free Least Fixed Point (ALFP) logic is an extension of Horn clauses with nested universal and existential quantification, stratified negation, and disjunction [49, 50]. It is more powerful than Datalog, but is still guaranteed to have a minimal solution, provided that the program is stratifiable.

Answer Set Programming (ASP) is a logic tailored to solving NP-hard problems that may involve non-monotonic reasoning [8, 42, 43]. A key development was the introduction of stable models (later known as answer sets) that allow models to be defined even in the presence of negative cycles in the implication graph [25]. FLIX does not yet support negation, so the challenges of monotonicity are restricted to ensuring that user-defined functions are monotone.

Prolog, CLP, ALFP, ASP, and other logic languages all offer various trade-offs in terms of performance, expressive power and safety (i.e. the existence of least models). FLIX is yet another point in this large design space and is more general than Datalog, but less general than Prolog.

Static Analyzers based on Horn Clauses. Logic programs, such as Prolog and CLP programs, have also been the subject of static analysis. In this line of research, static analyzers such as PLAI [46] compute the least fixed point of a set of Horn clauses over one or more lattices. These lattices and their associated operations are specified as “plug-ins” to the analysis. A key difference between these techniques and FLIX is the choice of evaluation strategy: FLIX uses bottom-up semi-naïve evaluation rather than top-down evaluation with tabulation, as in XSB. These developments have been used to translate object-oriented programs into constraint Horn clauses on which the static analysis is then performed [45].

6. Conclusion

We have presented FLIX, a declarative programming language for expressing and solving least fixed point problems, particularly static program analyses. FLIX is inspired by Datalog and extends it with lattices and monotone functions. We have defined a model-theoretic semantics for FLIX that is the foundation for any FLIX fixed-point solver, independent of its specific evaluation strategy. We have demonstrated the expressiveness of FLIX by implementing several well-known static analyses, including the IFDS and IDE algorithms. The declarative formulation of these analyses clearly reveals their close relationship. Experimental results show that the current FLIX interpreter is slower than hand-crafted analyzers, but we plan to address this in future work.

7. Future Work

We briefly outline three directions for future work:

Negation. FLIX does not support negation. We believe it is straightforward to extend the semantics and implementation to support stratified FLIX programs. However, we want to explore whether stratification is the right choice for specifying static analyses. Furthermore, we want to look at potentially interesting connections between negation and lattices.

Safety. Every Datalog program is guaranteed to terminate with the unique minimal model. As discussed previously, a FLIX program is also guaranteed to terminate with the minimal model, provided that every lattice is actually a complete lattice, of finite height, and every function is strict and monotone. Unfortunately, a FLIX programmer may inadvertently violate one or more of the required properties when specifying a lattice or function. We plan to investigate the use of automatic program verification techniques to guarantee that FLIX programs are meaningful.

Performance. As discussed in the evaluation, the current implementation of FLIX has many opportunities for improved performance, such as eliminating boxing, replacing the interpreter with compiled JVM bytecode, and using query planning and better index selection.

Acknowledgment

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

References

A. Cortesi, G. Costantini, and P. Ferrara. A Survey on Product


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


S. Fink and J. Dolby. WALA – The TJ Watson Libraries for Analysis, 2012.


