Object-Oriented Fixpoint Programming with Datalog

DAVID KLOPP, JGU Mainz, Germany
SEBASTIAN ERDWEG, JGU Mainz, Germany
ANDRÉ PACAK, JGU Mainz, Germany

Modern usages of Datalog exceed its original design purpose in scale and complexity. In particular, Datalog lacks abstractions for code organization and reuse, making programs hard to maintain. Is it possible to exploit abstractions and design patterns from object-oriented programming (OOP) while retaining a Datalog-like fixpoint semantics? To answer this question, we design a new OOP language called OODL with common OOP features: dynamic object allocation, object identity, dynamic dispatch, and mutation. However, OODL has a Datalog-like fixpoint semantics, such that recursive computations iterate until their result becomes stable. We develop two semantics for OODL: a fixpoint interpreter and a compiler that translates OODL to Datalog. Although the side effects found in OOP (object allocation and mutation) conflict with Datalog's fixpoint semantics, we can mostly resolve these incompatibilities through extensions of OODL. Within fixpoint computations, we employ immutable algebraic data structures (e.g. case classes in Scala), rather than relying on object allocation, and we introduce monotonically mutable data types (*mono types*) to enable a relaxed form of mutation. Our performance evaluation shows that the interpreter fails to solve fixpoint problems efficiently, whereas the compiled code exploits Datalog's semi-naïve evaluation.

CCS Concepts: • **Software and its engineering → Object oriented languages**; **Constraint and logic languages**; • **Theory of computation** → *Program analysis*.

Additional Key Words and Phrases: object-oriented, Datalog, fixpoint

ACM Reference Format:
David Klopp, Sebastian Erdweg, and André Pacak. 2024. Object-Oriented Fixpoint Programming with Datalog. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 273 (October 2024), 27 pages. https://doi.org/10.1145/3689713

1 Introduction

Datalog has come a long way from its origin as a database query language [Maier et al. 2018]. What makes Datalog attractive is its fixpoint semantics, which has led to its use in a wide range of applications. Modern usages of Datalog [Huang et al. 2011] include program analysis [Bravenboer and Smaragdakis 2009; Szabó et al. 2021], network monitoring [Abiteboul et al. 2005], and distributed computing [Alvaro et al. 2010, 2011a,b], all of which involve large and complex Datalog programs. The original design of Datalog is unfit for the development and maintenance of these programs.

In recent years, quite a few attempts have been made to improve the programmability of Datalog. Souffle [Jordan et al. 2016] extends Datalog with a module system in which components can encapsulate Datalog elements and inherit from each other to enable code reuse. Formulog [Bembenek et al. 2020] extends Datalog with functional programming to enable inspection and manipulation of complex terms. DataFun [Arntzenius and Krishnaswami 2016] and functional IncA [Pacak and Erdweg 2022] extend a functional programming language with relational-programming constructs to provide an alternative Datalog frontend. Flix [Madsen and Lhoták 2020] extends Datalog with a

Authors' Contact Information: David Klopp, JGU Mainz, Mainz, Germany, davklopp@uni-mainz.de; Sebastian Erdweg, JGU Mainz, Mainz, Germany, erdweg@uni-mainz.de; André Pacak, JGU Mainz, Mainz, Germany, pacak@uni-mainz.de.

ACM 2475-1421/2024/10-ART273
https://doi.org/10.1145/3689713

functional metalanguage to construct and solve Datalog programs at run-time. But only QL [Avgustinov et al. 2016] explores how object-oriented programming and Datalog can be integrated by implementing a light-weight object-oriented yet pure and relational surface language for Datalog.

In this paper, we study how to integrate Datalog's fixpoint semantics and object-oriented programming (OOP). Specifically, we aim to support dynamic object allocation with unique identities, inheritance with dynamic dispatch, and mutation. Our goal is to unleash the well-known advantages of these OOP features for building and maintaining software and frameworks at scale, accommodating the growing complexity in current usage scenarios of Datalog. In particular, it should become possible to employ OOP design patterns to organize programs, such as the visitor pattern for separating data and computation. At the same time, we must retain Datalog's characteristic least fixpoint semantics, which assists developers in processing cyclic data by efficiently iterating a computation until its result is stable. Unfortunately, the least fixpoint semantics and OOP are in conflict. Consider the following Scala-like OOP example:

```
1  class DataAnalysis(var nodeCount: Int = 0):
2    def processNode(n: Node): Data =
3      nodeCount += 1
4      return new Data(n.info)
5
6  val analysis = new DataAnalysis
7  val graph = ... // a possibly cyclic graph to be analyzed
8  val data = ...  // relational code that iterates processNode(..) on graph until stable
```

The last line of this example illustrates an interaction between relational and OOP code: We want to process `graph` using the least fixpoint semantics. The problem is that the iterated computation has side effects: mutation and object allocation in lines 3 and 4. This computation does not have a least fixpoint because the allocator and the mutable variable keep changing in each iteration. Therefore, there exists no integration of OOP and Datalog without compromises.

We present OODL, an OOP language with least fixpoint semantics, which we define in two different styles. First, we define a fixpoint interpreter by adopting recent fixpoint algorithms from big-step abstract interpretation [Keidel et al. 2023]. The interpreter is correct and finds least fixpoints, but has insufficient performance. Therefore, second, we also develop a compiler that translates OODL to Datalog, to reuse existing efficient Datalog solvers. Our translation shows how OOP concepts can be faithfully encoded in Datalog. We eliminate mutable local variables with an SSA transformation but, interestingly, no ϕ-nodes are necessary in the generated code. However, there are limitations: No traditional allocation or field mutation may take place, during a fixpoint subcomputation. Instead of traditional object allocation, developers must use immutable algebraic data (e.g., case classes in Scala), which do not carry an object ID. To support mutable fields, we describe a novel abstraction for monotonically mutable data types (*mono types*) that is compatible with fixpoint semantics. Mono types are a OOP-inspired writable mutable collection `m += a`, permitting only monotonically increasing observations `m.result` for a user-defined preorder. Mono types generalize lattice-based aggregation [Madsen et al. 2016; Szabó et al. 2018].

We implemented the interpreter, compiler, and three case studies for OODL, including a performance evaluation. The interpreter fails to solve fixpoint problems efficiently, whereas the compiled code exploits Datalog's semi-naïve evaluation. In summary, we make the following contributions:

- We expose the tension between least fixpoint, object allocation, and mutation (Section 2).
- We develop a fixpoint interpreter for an OOP language, which is correct but slow (Section 3).
- We show how to compile OOP constructs to Datalog including object allocation (Section 4).
- We present an encoding of mutation in Datalog using SSA and timestamps (Section 5).

- We propose alternatives to allocation and mutation for fixpoint computations (Section 6).
- We discuss case studies, the language design and performance comparison (Section 7).

2 Object-Oriented Programming in Datalog: Motivating Example

Our goal is to understand how OOP and Datalog can be deeply integrated to exploit Datalog's semi-naïve fixpoint evaluation for OOP programs. In this section, we introduce a larger motivating example to explain why such integration is desirable, both for OOP and for Datalog.

Our motivating example stems from the domain of program analysis, which is a common use case for Datalog. Consider we want to analyze this simple language of inter-dependent definitions:

$$\begin{array}{lll} \text{(programs)} & p & ::= \overline{d} \\ \text{(definitions)} & d & ::= \texttt{def}\ x = e \\ \text{(expressions)} & e & ::= n \mid x \mid e + e \end{array}$$

We want to perform a dependency analysis of this language in three steps:

(1) Given an abstract syntax tree (AST), compute the corresponding abstract syntax graph (ASG). In an ASG, each variable reference carries a pointer to the referenced definition, as illustrated using numeric subscripts in the following example program:
```
def a1 = c3; def b2 = 2; def c3 = a1; def d4 = b2; def main5 = c3
```
(2) Given the ASG of a main definition, compute the dependency graph of all definitions reachable from it. A definition depends on all definitions referenced in its body. The dependency graph for $\texttt{main}_5$ is $G = \{(5,3),(3,1),(1,3)\}$. Definitions $\texttt{d}_4$ and $\texttt{b}_2$ are unreachable from $\texttt{main}_5$.
(3) Given a dependency graph and a definition, find all definitions it transitively depends on. That is, $\texttt{a}_1$, $\texttt{c}_3$, and $\texttt{main}_5$ transitively depend on $\{1,3\}$, while $\texttt{b}_2$ depends on $\emptyset$ and $\texttt{d}_4$ on $\texttt{b}_2$.

Below, we discuss how this dependency analysis can be implemented in OOP and Datalog.

2.1 Dependency Analysis in OOP

Figure 1 shows a possible implementation of the dependency analysis in Scala3. We encode the abstract syntax of our language as a corresponding class hierarchy. We can encode our example program from above as follows:

```
val d1 = new Def("a", new Var("c")); val d2 = new Def("b", new Num(2));
val d3 = new Def("c", new Var("a")); val d4 = new Def("d", new Var("b"))
val main = new Def("main", new Var("c")); val prog = new Prog(List(d1, d2, d3, d4, main))
```

To process programs, we employ the visitor pattern: Each class `accept`s a visitor object and invokes the appropriate `visit` method. The visitor pattern separates data definitions from their processing, allowing us to add operations without changing the classes that define the abstract syntax.

We define two visitors to implement the dependency analysis. The first visitor `NameAnalysis` resolves references to their definition by looking up the name in the list of definitions. For bound variables, we store a pointer to the binding definition in field `Var.target` of the reference, converting the AST into an ASG. That is, after invoking `prog.accept(new NameAnalysis(prog))`, the `target` field of all references is set. Since `NameAnalysis` traverses the AST, it always terminates. Datalog's least fixpoint semantics is only required when recursively processing cyclic data, as in the next steps.

The second visitor `DependencyAnalysis` processes graph-shaped data with cycles. Figure 1 shows an idealized version of Scala3, where we assume graph traversals magically terminate once their result becomes stable. Later, when we compile OOP to Datalog, such idealized graph traversals will indeed be supported and translate to terminating Datalog code. We implement Step (2) of the dependency analysis through a visitor that adds a dependency `Edge` from current definition `d` to the referenced definition `v.target` whenever it is defined. In addition, the visitor continues the

```
abstract class Visitor: ...
class Prog(val defs: List[Def]): ...

class Def(val name: String, val exp: Exp): ...
val undef = new Def("undef", new Num(-1))

abstract class Exp: ...
class Num(val value: Int) extends Exp: ...
class Var(val name: String, var target: Def = undef) extends Exp:
    def accept(v: Visitor, d: Def): Unit = v.visitVar(this, d)

// Step (1). Constructs ASG by setting Var.target field
class NameAnalysis(val prog: Prog) extends Visitor:
    override def visitVar(v: Var, d: Def): Unit = prog.defs.find(_.name == v.name) match
        case Some(d) => v.target = d
        case None => // unbound variable

class Edge(val from: Def, val to: Def)
class DependencyAnalysis extends Visitor:
    // Step (2). Constructs dependency graph by traversing ASG, may DIVERGE
    var edges: Set[Edge] = Set.empty
    override def visitVar(v: Var, d: Def): Unit =
        if (v.target != undef) {
            edges += new Edge(d, v.target)
            v.target.accept(this)
        }
    // Step (3). Computes transitive closure by traversing edges, may DIVERGE
    def dependencies(d1: Def): Set[Def] =
        directDeps(d1) ++ (for (d2 <- directDeps(d1); d3 <- dependencies(d2)) yield d3)
    def directDeps(d: Def): Set[Def] = edges.filter(_.from == d).map(_.to)
```

Fig. 1. Implementing the dependency analysis in OOP using the visitor pattern and idealized graph traversals.

traversal at the referenced definition, so that we effectively traverse the ASG. For example, consider a run of the dependency analysis on the main definition `main.accept(new DependencyAnalysis)` after `NameAnalysis`. The dependency analysis will visit $\texttt{main}_5$, $\texttt{c}_3$, $\texttt{a}_1$, $\texttt{c}_3$, $\texttt{a}_1$, and so on. That is, the analysis is in a loop because the ASG is cyclic. Moreover, the set of edges is *not* stable since we continuously create new `Edge` objects that we compare by object identity. Had we used structural equality for `Edge` objects, a least fixpoint would exist, but the OOP semantics does not find it.

Lastly, we compute the transitive dependencies of a given definition. We compute the transitive closure in Datalog-style, but using for-comprehensions: The set of dependencies is the set of direct dependencies plus the dependencies of all direct dependencies. This definition is recursive, but has a least fixpoint for any finite graph. In our example, we traverse the dependency graph stored in `edges`. Even if `edges` were finite, OOP cannot find the least fixpoint of `dependencies` for cyclic graphs.

To summarize, OOP allows us to use the visitor pattern, which is effective for implementing the dependency analysis. However, OOP lacks a crucial linguistic feature for processing cyclic graphs: a least fixpoint semantics. Without it, we need to manage the graph traversal ourselves, using a worklist, a set of seen nodes, and a cache of prior results to detect when the computation is stable. How can we integrate Datalog and OOP to retain the advantages of both?

2.2 Dependency Analysis in Datalog

Before discussing our approach and the involved challenges, we should question: Why should we use OOP in the first place? Why don't we implement the dependency analysis in Datalog directly? The problem is that Datalog is a simple logic programming language that lacks many features programmers know from mainstream languages: Datalog does not support structured programming (control flow), no mutable data, there is no data abstraction and no polymorphism of any kind. We discussed various approaches to improve the programmability of Datalog in the introduction. However, none of them support mutable data or dynamic object allocations.

To implement the dependency analysis in Datalog directly, we have to rewrite the program completely to fit the logic programming paradigm. For Step (1), rather than traversing the AST and updating a field, we must "search" the AST to find and associate variables with targeted definitions:

```
resolved(prog, e, trgDef) :-
    parentProg(e, prog), // enumerate all expressions in prog
    Var(x, e),           // ensure e is a variable and retrieve its name x
    Prog(prog, trgDef),  // enumerate all trgDef in prog
    Def(trgDef, x, _).   // filter trgDef with name x

target(prog, e, trgDef) :- resolved(prog, e, trgDef).   // either e resolves to trgDef, or ...
target(prog, e, trgDef) :-
    parentProg(e, prog), Var(x, e), Prog(prog, trgDef), // find candidates as above
    ¬ resolved(prog, e, trgDef),                        // if e is unresolved
    Def(trgDef, "undef", _).                            // then use "undef" fallback

parentProg(e, prog) :- parentDef(e, def), Prog(prog, def).

parentDef(e, def) :- Def(def, _, e).
parentDef(e, def) :- Add(a, e, _), parentDef(a, def).
parentDef(e, def) :- Add(a, _, e), parentDef(a, def).
```

While `resolved` associates variables to their targets, we had to include another relation `target` to simulate the mutable field `Var.target`. Here we exploit domain knowledge: Either a variable is resolved or its target is the default `"undef"` definition. It is not obvious how to generalize this code to allow multiple updates of `Var.target`.

To compute the dependency graph in Step (2), again we must reformulate our computation significantly. Rather than traversing the ASG, we must "search" for edges using relational queries:

```
depGraph(prog, def, trgDef) :-
    reachable(prog, def),       // enumerate all definitions reachable in def
    parentDef(e, def),          // enumerate all expressions in each def
    Var(_, e),                  // ensure e is a variable
    target(prog, e, trgDef),    // retrieve the variable's target
    ¬ Def(trgDef, "undef", _). // ensure trgDef is not "undef"

reachable(prog, def) :-
    Prog(prog, def),          // enumerate all definitions in prog
    Def(def, "main", _).      // the main def is reachable
reachable(prog, def) :-
    target(prog, e, def),     // enumerate all vars e that resolve to def
    ¬ Def(def, "undef", _),   // ensure def is not "undef"
    parentDef(e, otherDef),   // find the containing definitions of those variables
    reachable(otherDef).      // def is reachable if any of the other definitions is reachable
```

We use relation `reachable` to restrict the dependency graph to nodes reachable from the main definition. Note how we again exploited domain knowledge: The mutable field `edges` is a set that grows monotonically, which is why we're able to encode it as a Datalog relation directly. It is not obvious how to generalize this to support other mutable collections and non-monotonic updates.

To summarize, it is possible to implement the dependency analysis in Datalog, but we have to change our approach drastically. Most importantly, we had to change our way of thinking about the problem: Instead of control-flow driven traversals of structured data, we had to re-invent the analysis in the style of a database query, loosing the visitor pattern along the way. Moreover, we had to exploit domain knowledge twice to make the encoding tractable. In this paper, we propose an approach that compiles OOP programs to Datalog automatically, such that developers can model domain concerns in OOP while benefiting from Datalog's least fixpoint semantics.

3 Least-Fixpoint Semantics for OOP

This paper studies how to integrate a least fixpoint semantics into object-oriented programming (OOP). We can tackle this problem from two alternative directions: We can start with an OOP language and modify its semantics to compute least fixpoints. Or, we can start with a language that has least fixpoint semantics (e.g. Datalog) and embed an OOP language into it. This paper studies and compares both approaches, starting with the first alternative in the present section.

Before we start our exploration, we clarify what it means to compute least fixpoints for OOP.

Definition 1 (OOP fixpoint). Given a big-step OOP semantics $e, \rho, \sigma \Rightarrow v, \sigma'$ that evaluates an expression e under environment ρ (binding local variables) and heap σ (mapping object IDs to object records) to a value v and an updated heap σ'. The semantics computes a fixpoint if $e, \rho, \sigma \Rightarrow v, \sigma'$ implies $e, \rho, \sigma' \Rightarrow v, \sigma'$. That is, if we run the program a second time starting with the updated heap σ', we obtain stable results v and σ'. The semantics computes the *least* fixpoint if v and σ' are the smallest results satisfying this definition, relative to some partial order on values discussed below.

In general, not all OOP programs have a least fixpoint. For example, consider the program `def loop1(x) = loop1(new Object())`, which tries to allocate an infinite number of objects on the heap. For this program, no heap σ' satisfies our fixpoint property. However, we may not confuse finding a least fixpoint with program termination. For example, the program `def loop2(x) = loop2(x + 1)` does have a least fixpoint: the empty value $\bot$ (indicating abnormal execution) and the empty heap. In `loop1`, object allocation triggers a side-effect that extends the heap in each recursive call, precluding a stable result. In contrast, `loop2` computes with primitive values that are not allocated on the heap, so that the empty heap satisfies the fixpoint property.

Our fixpoint property is relative to a partial order on values. As illustrated in the previous section, fixpoint computations typically collect results in a recursively defined set. For example, the dependency analysis shown in Figure 1 recursively traverses the ASG to collect dependency edges in a set, and it recursively traverses the dependency graph to collect transitive dependencies in a set. Our fixpoint property is satisfied by any sufficiently large set of edges and dependencies, for which no new edges or dependencies can be found. But we are interested in the least fixpoint: the smallest set of edges and dependencies that is complete. Accordingly, we use a partial order on values that compares set values using the subset relation, while comparing other values pointwise.

To keep the exposition short and simple, the fixpoint property from above does not actually enforce termination. We only require that the semantics yields a fixpoint if it terminates at all. According to this definition, the semantics could diverge even for programs that have a least fixpoint. However, our intention is that the semantics should compute a least fixpoint whenever it exists. We next present an approach for computing least fixpoints for OOP languages effectively.

$$\frac{}{\mathsf{fix}\ (e,\rho,\sigma,\langle c_1 \cdots (e,\rho,\sigma)_b \cdots c_n\rangle)\ \mathsf{was}\ (v_r,\sigma_r) \Rightarrow v_r,\sigma_r,\langle c_1 \cdots (e,\rho,\sigma)_{\mathsf{true}} \cdots c_n\rangle}\ \text{Fix-Recurrent}$$

$$\frac{(e,\rho,\sigma) \notin cs \qquad e,\rho,\sigma,cs \cdot (e,\rho,\sigma)_{\mathsf{false}} \Rightarrow v',\sigma',cs' \cdot (e,\rho,\sigma)_{\mathsf{false}}}{\mathsf{fix}\ (e,\rho,\sigma,cs)\ \mathsf{was}\ (v_r,\sigma_r) \Rightarrow v',\sigma',cs'}\ \text{Fix-Nonrecurrent}$$

$$\frac{(e,\rho,\sigma) \notin cs \qquad e,\rho,\sigma,cs \cdot (e,\rho,\sigma)_{\mathsf{false}} \Rightarrow v',\sigma',cs' \cdot (e,\rho,\sigma)_{\mathsf{true}} \qquad (v',\sigma') = (v_r,\sigma_r)}{\mathsf{fix}\ (e,\rho,\sigma,cs)\ \mathsf{was}\ (v_r,\sigma_r) \Rightarrow v_r,\sigma_r,cs'}\ \text{Fix-Stable}$$

$$\frac{\begin{array}{ll}(e,\rho,\sigma) \notin cs & e,\rho,\sigma,cs \cdot (e,\rho,\sigma)_{\mathsf{false}} \Rightarrow v',\sigma',cs' \cdot (e,\rho,\sigma)_{\mathsf{true}} \\ (v',\sigma') \neq (v_r,\sigma_r) & \mathsf{fix}\ (e,\rho,\sigma',cs')\ \mathsf{was}\ (v',\sigma') \Rightarrow v'',\sigma'',cs''\end{array}}{\mathsf{fix}\ (e,\rho,\sigma,cs)\ \mathsf{was}\ (v_r,\sigma_r) \Rightarrow v'',\sigma'',cs''}\ \text{Fix-Iterate}$$

Fig. 2. A big-step fixpoint algorithm for OOP, iterating each recurrent configurations (e,ρ,σ) until stable.

3.1 A Fixpoint Interpreter for OOP Languages

To compute least fixpoints for OOP programs, we start with a standard big-step OOP semantics $e,\rho,\sigma \Rightarrow v,\sigma'$, implemented as a recursive interpreter. Our key idea is to draw inspiration from abstract interpretation: Abstract interpreters compute the fixpoint of a transfer function over an abstract domain. In particular, we build on recent fixpoint algorithms [Keidel et al. 2023] developed for big-step abstract interpreters, whose implementation resembles a recursive interpreter [Darais et al. 2017]. Essentially, a big-step abstract interpreter repeats each recursive sub-computation until a fixpoint is reached, so that the final analysis result is a sound over-approximation of the program behavior. We adopt this idea to compute fixpoints in the concrete semantics of an OOP interpreter.

We formalize a fixpoint algorithm for OOP in Figure 2 as an auxiliary reduction relation fix:

$$\mathsf{fix}\ (e,\rho,\sigma,cs)\ \mathsf{was}\ (v_r,\sigma_r) \Rightarrow v',\sigma',cs'$$

The fix reduction relation takes an interpreter configuration $c = (e,\rho,\sigma)$ and a configuration stack $cs = \langle c_1 \cdots c_n\rangle$ as input, together with a previous intermediate fix result v_r and σ_r. The configuration stack traces recursive calls of the OOP program to detect and prevent infinite loops. When an input configuration c reoccurs on the stack, the computation would go into a loop. Rule Fix-Recurrent aborts such computations, marks the configuration as recurrent c_{true}, and yields the previous fix result. If c does not occur on the stack, we push c_{false} on the stack and distinguish three cases.

(1) Fix-Nonrecurrent. If the computation does not revisit c, yield the computation's result.
(2) Fix-Stable. If the computation revisits c but its result coincides with the previous fix result (v_r,σ_r), we have found a fixpoint and yield its result.
(3) Fix-Iterate. The computation does revisit c but its result differs from the previous fix result. We have not found a fixpoint yet and must iterate the computation further, updating the intermediate fix result to the current result.

This fixpoint algorithm is essentially a specialized version of the family of fixpoint algorithms described by Keidel et al. [2023], and we inherit the algorithm's correctness from them. However, our algorithm does not guarantee termination since we do not join the previous fix result (v_r,σ_r) and new fix result (v',σ') in rule Fix-Iterate. Consequently, the fixpoint algorithm may diverge, for example, when an entry in the heap oscillates and no fixpoint exists. On the other hand, if a computation's result stabilizes eventually, our algorithm will find and yield its fixpoint correctly.

Our fixpoint algorithm must be plugged into the OOP semantics for each rule that handles a possibly non-terminating construct. For example, the reduction rule for method invocations must evaluate the method's body using the fix relation, as to detect and prevent recurrent method calls. The OOP semantics must also thread the configuration stack *cs* in store-passing style. We have implemented an interpreter for a standard OOP language with mutable variables and fields that uses our fixpoint algorithm this way. While yielding correct results, the performance is unsatisfactory.

3.2 Fixpoint Interpreters Are Naïve

Our fixpoint algorithm is computationally expensive since it resembles what is known as *naïve evaluation* in Datalog literature [Green et al. 2013]. A fixpoint algorithm is naïve if it repeats the computation using the combined result of all prior iterations. When the fixed computation is monotone, the fixpoint algorithm is bound to rederive the results of iteration k in all later iterations $i > k$. State of the art Datalog systems use *semi-naïve evaluation*, which only considers results first found in iteration k when deriving results in iteration $k + 1$, eliminating redundant derivations. Unfortunately, it is not clear how we could adopt semi-naïve evaluation. First, there are no guarantees that the user-defined OOP computations are monotone. Second, semi-naïve evaluation has so far exclusively been applied to pure programming languages, such as Datalog or Datafun [Arntzenius and Krishnaswami 2020]. In contrast, OOP languages involve object allocation on the heap and subsequent modifications of those objects. It is not clear how to make such computations incremental, as required for semi-naïve evaluation.

We conducted performance measurements to show that the performance penalty for naïve evaluation is significant. Specifically, we benchmarked our interpreter on the standard `path` program, which computes the transitive closure over small strongly connected graphs that we generate. The graph on the right compares the interpreter's performance to a Datalog solver that uses semi-naïve evaluation. While semi-naïve evaluation computes the transitive closure instantaneously, the naïve evaluation of the fixpoint interpreter fails to scale. Even worse, when representing the graph's nodes as objects rather than primitive integers, the performance of the fixpoint interpreter deteriorates even further. This is because the fixpoint algorithm checks the stability of the heap in each fixpoint iteration, which takes time linear in the size of the heap. In the remainder of this paper, we develop an alternative fixpoint semantics for OOP by translation to Datalog.

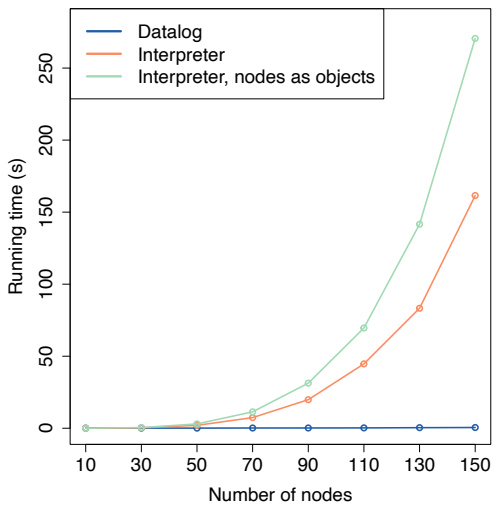


Fig. 3. Running times of the naïve interpreter vs semi-naïve Datalog for the standard `path` program

4 Compiling Immutable OOP to Datalog

As shown in the previous section a Datalog solver using semi-naïve evaluation outperforms a fixpoint-based interpreter significantly. To utilize the performance of semi-naïve evaluation for fixpoint computation, we aim to deeply integrate OOP and Datalog, by compiling OODL to Datalog. We start with an immutable OODL variant with unmodifiable local variables and fields.

$$
\begin{array}{lll}
\text{(programs)} & p & ::= \overline{c} \\
\text{(classes)} & c & ::= \texttt{class}\ C(\overline{x})\ \texttt{extends}\ D\ \{\overline{f}, \overline{m}\} \\
\text{(methods)} & m & ::= \texttt{def}\ m(\overline{x}) = \{\overline{s}; e\} \\
\text{(statements)} & s & ::= \texttt{val}\ x = e \mid \texttt{if}(e == e)\ \{\overline{s}\}\ \texttt{else}\ \{\overline{s}\} \\
\text{(expressions)} & e & ::= v \mid x \mid e.f \mid \texttt{new}\ C(\overline{e}) \mid e.m(\overline{e}) \mid \texttt{basefun}(e) \\
\text{(values)} & v & ::= \texttt{base}
\end{array}
$$

Fig. 4. Syntax of immutable OODL with base values and base functions.

Figure 4 shows the Scala-like syntax of immutable OODL. A program consists of classes. Each class has a single super class, a list of constructor parameters $\overline{x}$, fields $\overline{f}$, and methods $\overline{m}$. As usual, we omit the superclass for classes that extend the root class (e.g., `Object` in Java). The number of constructor parameters must match the number of transitively inherited fields, which are initialized to constructor arguments by the implicit default constructor. A method contains a sequence of statements followed by a single expression, which yields the returned value. For now, we only consider a single statement that binds an immutable local variable for the remainder of the method, and a test for object equality. We extend OODL with more statements in Section 5 when introducing mutation. Besides standard OOP expressions, we also include base functions for convenience. We will use Scala3-like syntax when presenting OODL programs in the remainder of this paper.

For a simple example OODL program, consider the following encoding of Peano numbers:

```
class Nat:
    def add(that: Nat): Nat = that
class Zero extends Nat
class Succ(p: Nat) extends Nat:
    val pred: Nat = p   // field initialization is implicit in OODL, but explicit in examples
    def add(that: Nat): Nat = { val newPred = this.pred.add(that); new Succ(newPred) }
```

Although we omitted types in the grammar, OODL is statically typed and we annotate types in our examples. We do not formalize the type system of OODL because it is completely standard. Our translation to Datalog relies on typing information for resolving field and method lookups.

4.1 Compilation Strategy for Immutable OODL

We propose to integrate OOP and Datalog by compiling OOP features to Datalog. To this end, we must find encodings for objects, fields, constructors, and methods in Datalog. The most important question is how to encode dynamically allocated objects at run-time. Prior work supported statically declared objects [Abiteboul et al. 1993] or enumerates all objects of a class centrally [Avgustinov et al. 2016]. In contrast, OODL supports full OOP with dynamic, conditional, and decentralized object creation. We represent objects with unique identities created at run-time:

Principle 1: Objects as Unique Identities. An object identity $\mathrm{OID}(C, \omega)$ consists of the run-time class C of the object and a numeric value ω. As usual for OOP languages, we require the run-time class C of an object for dynamic dispatch. The allocation count ω uniquely identifies objects and can be used for testing object equality. When compiling OODL programs to Datalog, we thread and increment ω to guarantee OIDs are truly unique, and methods take an allocation count as input and produce an updated allocation count output. Consequently, our targeted Datalog language must support arithmetics or algebraic data types, as is standard in many modern Datalog systems including Souffle [Jordan et al. 2016] and IncA [Szabó et al. 2018]. Note that our interpreter does not use an allocation count, but instead relies on the meta-interpreter to generate unique objects.

Principle 2: Fields as Binary Relations. Objects are records that associate values to fields. Rather than storing field values in the object representation, we store field values in dedicated relations. For each field f of class C, we define a relation $C\#f \subseteq \mathsf{OID} \times v$, where a Datalog value v is either an OID or a base value. Since an object can only carry one value per field, relation $C\#f$ has a functional dependency $\mathsf{OID} \to v$, meaning OID uniquely determines v. Note that we will have to revise our encoding of fields once we consider mutability in Section 5.

Principle 3: Constructors Allocate Objects. Constructors create new objects, assign them an OID, and initialize their fields. For example, `new Succ(new Succ(new Zero()))` evaluates to an object `OID(Succ,2)`, where the `Succ#pred` relation contains two entries: `(OID(Succ,2), OID(Succ,1))` and `(OID(Succ,1), OID(Zero,0))`. In our design, we generate the required fresh OIDs at the call site of the constructor and provide the OID to the constructor as an input alongside all constructor arguments. That is, for each class C we generate a relation $C \subseteq \mathsf{OID} \times v \times \cdots \times v$, where the OID uniquely determines the other columns. These constructor relations are responsible for associating the constructor arguments with the appropriate field relations for the given OID. Moreover, we can use the constructor relation to enumerate all objects of a given class that have been created so far.

Principle 4: Methods with Dispatch Tables. While it is well-known that we can encode functions $f : (T_1, \ldots, T_n) \to T$ as relations $f \subseteq T_1 \times \cdots \times T_n \times T$, it is not clear how to encode dynamic dispatch over dynamically allocated objects in a logic programming language. Datalog itself does not support any form of higher-order control flow, so that we have to eliminate dynamic dispatch during compilation. We might be tempted to generate a relation $C\#m$ for each method m defined in C as done in prior work, but this strategy will fail. Abiteboul et al. [1993] follow this approach and translate (overriding) methods to mutually exclusive relations: relation `Succ#add` is applicable if `this` has run-time type `Succ` and `Nat#add` is applicable if `this` does not have run-time type `Succ`. QL [Avgustinov et al. 2016] uses similar guards to exclude receiver objects that have more specific implementations. Unfortunately, the required negative predicate calls conflict with dynamically allocated objects, since we obtain unstratified dependency graphs. In our example, `Succ.add` invokes `Nat.add`, which we would need to guard by `not Succ(this,_)`. However, the constructor relation `Succ` depends cyclically on its invocation in `Succ.add` with a negative call, which is invalid in Datalog.

We propose a different strategy for supporting dynamic dispatch based on a simple observation: Since the run-time class of a receiver object is only known during execution, we must treat it as a run-time value in Datalog and use it to dynamically resolve a call's target. This allows us to select the single most-specific invocation target rather than querying all candidates and filtering out inapplicable ones. Specifically, for each method m in the program, we generate a dispatch table $dispatch_m \subseteq C \times D$ with functional dependency $C \to D$. Given the run-time class C of a receiver object, we can query $dispatch_m$ to obtain the class D with the most-specific implementation of m for C. We can populate the dispatch tables at compile time, similar to virtual method tables in standard OOP compilers. For $dispatch_{add}$ we obtain three entries: (`Nat`, `Nat`), (`Zero`, `Nat`), and (`Succ`, `Succ`).

Principle 5: Demanded Control Flow. The execution of Datalog programs is solely driven by data flow, but does not have a notion of control flow. Therefore, we must encode OODL's control flow using data flow in Datalog. Like prior work that compiled functional programs to Datalog [Pacak and Erdweg 2022], we employ the *demand transformation* [Tekle and Liu 2010] to achieve this. The demand transformation is a magic set transformation [Beeri and Ramakrishnan 1991] that computes the demanded inputs of a relation, that is, how a relation is being queried in other parts of the program. The demand transformation then adds guards to the relations to ensure it only computes results when demanded. Effectively, this enforces a derivation order on Datalog that corresponds to the control flow of OOP.

$$
\begin{array}{lll}
\text{(Datalog programs)} & D & ::= \overline{r} \\
\text{(rules)} & r & ::= \mathsf{R}(\overline{t}) \ \texttt{:-}\ \overline{a}. \\
\text{(atoms)} & a & ::= t = t \mid t \neq t \mid \mathsf{R}(\overline{t}) \\
\text{(terms)} & t & ::= v \mid x \mid \mathsf{basefun}(\overline{t}) \\
\text{(values)} & v & ::= \mathsf{OID}(C, n) \mid \mathsf{base} \\
\text{(derived atom forms)} & a & ::= \ldots \mid \blacklozenge \mid \overline{a} \vee \overline{a}
\end{array}
$$

Fig. 5. An intermediate representation for Datalog with base values and demand markers ◆.

4.2 Formal Translation of Immutable OODL to Datalog

We are now ready to translate immutable OODL to Datalog. Technically, we target a variant of Datalog as defined in Figure 5. Our targeted Datalog is negation-free but supports base functions and values as well as the construction of OID terms. To simplify the translation, we extend Datalog with two derived forms that can be eliminated in a subsequent step. First, we introduce a demand marker ◆, which signifies that the containing rule may have unbound variables. These variables will be bound as inputs by the demand transformation in a later step. Second, we introduce a conditional construct $\overline{a_1} \vee \overline{a_2}$, which combines the result from $\overline{a_1}$ (if any) with those from $\overline{a_2}$ (if any). We subsequently eliminate such disjunctions by converting rules to disjunctive normal forms.

We define the translation of OODL to Datalog in Figure 6, providing different translation functions for the various syntactic constructs of OODL. For expressions and statements, we thread allocation counts ω through the translation to ensure unique OIDs for constructor calls. We translate expressions using function $\mathsf{E}[\![e]\!]_\omega$, which yields $t \dashv_{\omega'} \overline{a}$, that is: A term t representing the value e evaluates to, a sequence of atoms $\overline{a}$ required for computing t, and an update allocation count ω'.

OODL values and variables translate to Datalog values and variables directly. For field lookups $e.f$, we first compile e to obtain its Datalog representation t. Based on the static type C of e, we find the class D that defines field f. We then query $D\#f$ to obtain the field's value. To assist the reader, we use a blue font to mark terms that signify inputs in the sense of the demand transformation. For fields, the receiver t is an input whereas the field value y is an output.

We handle constructor calls as described in Principle 3 above. After translating all constructor arguments $e_1, \ldots, e_n$, we use the final allocation count ω_n to create a fresh $\mathsf{OID}(C, \omega_n)$. We then query the constructor relation C to initialize the fields of the object. Note that the rule for constructor calls is the only place where ω is incremented, yielding $\omega_n + 1$.

For method invocations, we query the OID rather than constructing a new one. That is, we obtain the run-time class C of t_0 and use it to query the dispatch table of m to obtain the dispatch target D. We then query relation m to trigger the implementation of D, given the receiver object t_0 and the arguments $t_1, \ldots, t_n$. We also pass the latest allocation count ω_n as input to m, so that objects created within m get fresh OIDs. The query of m provides a return value y and an updated allocation count ω'. Calls to base functions are handled straight forwardly.

We compile statements using $\mathsf{S}[\![s]\!]_\omega$, which only yields a sequence of atoms and an updated allocation count, but no term. For immutable local variables `val x = e`, we introduce a corresponding binding $x = t$ in Datalog. For conditionals, we generate a Datalog conditional over the appropriate atoms, carefully making sure to thread the allocation count through all recursive translations.

The remaining translation functions handle fields, methods, and classes, for which we generate Datalog rules. The translation of fields is maybe most mysterious: For an immutable field f in class C, function $\mathsf{F}[\![f]\!]_C$ yields a rule $C\#f(\mathit{this}, v)$ whose body only contains a demand marker. Indeed, the demand transformation will make sure to bind these parameters in accordance with the queries that occur in the constructor relation C. Likewise, we generate a demand marker when translating

$$\mathsf{E}[\![.]\!]_\omega : e \to t \dashv_\omega \overline{a}$$

$$\mathsf{E}[\![v]\!]_\omega = v \dashv_\omega \varepsilon$$

$$\mathsf{E}[\![x]\!]_\omega = x \dashv_\omega \varepsilon$$

$$\mathsf{E}[\![e.f]\!]_\omega = y \dashv_{\omega'} \overline{a},\ D\#f(t, y)$$
where $\mathsf{E}[\![e]\!]_\omega = t \dashv_{\omega'} \overline{a}$
$C = typeof(e), D = findFieldDef(C, f)$
y is fresh

$$\mathsf{E}[\![\mathsf{new}\ C(e_1, \ldots, e_n)]\!]_\omega = y \dashv_{(\omega_n+1)} \overline{a_1}, \ldots, \overline{a_n},\ y = \mathsf{OID}(C, \omega_n),\ C(y, t_1, \ldots, t_n)$$
where $\mathsf{E}[\![e_1]\!]_\omega = t_1 \dashv_{\omega_1} \overline{a_1} \quad \cdots \quad \mathsf{E}[\![e_n]\!]_{\omega_{n-1}} = t_n \dashv_{\omega_n} \overline{a_n}$
y is fresh

$$\mathsf{E}[\![e_0.m(e_1, \ldots, e_n)]\!]_\omega = y \dashv_{\omega'} \overline{a_0}, \ldots, \overline{a_n},\ t_0 = \mathsf{OID}(C, \_),\ dispatch_m(C, D),\ m(D, t_0, t_1, \ldots, t_n, y, \omega_n, \omega')$$
where $\mathsf{E}[\![e_0]\!]_\omega = t_0 \dashv_{\omega_0} \overline{a_0} \quad \cdots \quad \mathsf{E}[\![e_n]\!]_{\omega_{n-1}} = t_n \dashv_{\omega_n} \overline{a_n}$
y and ω' are fresh

$$\mathsf{E}[\![\mathsf{basefun}(e_1, \ldots, e_n)]\!]_\omega = \mathsf{basefun}(t_1, \ldots, t_n) \dashv_{\omega_n} \overline{a_1}, \ldots, \overline{a_n}$$
where $\mathsf{E}[\![e_1]\!]_\omega = t_1 \dashv_{\omega_1} \overline{a_1} \quad \cdots \quad \mathsf{E}[\![e_n]\!]_{\omega_{n-1}} = t_n \dashv_{\omega_n} \overline{a_n}$

$$\mathsf{S}[\![.]\!]_\omega : s \to \overline{a} \dashv_\omega$$

$$\mathsf{S}[\![\mathtt{val}\ x = e]\!]_\omega = \overline{a},\ x = t \dashv_{\omega'}$$
where $\mathsf{E}[\![e]\!]_\omega = t \dashv_{\omega'} \overline{a}$

$$\mathsf{S}[\![\mathtt{if}(e_1 == e_2)\ \{s_3; \ldots; s_k\}\ \mathtt{else}\ \{s_{k+1}; \ldots; s_n\}]\!]_\omega = \overline{a_1}, \overline{a_2},\ (t_1 = t_2, a_3, \ldots, a_k) \vee (t_1 \neq t_2, a_{k+1}, \ldots, a_n) \dashv_{\omega_n}$$
where $\mathsf{E}[\![e_1]\!]_\omega = t_1 \dashv_{\omega_1} \overline{a_1}, \quad \mathsf{E}[\![e_2]\!]_{\omega_1} = t_2 \dashv_{\omega_2} \overline{a_2}$
$\mathsf{S}[\![s_3]\!]_{\omega_2} = \overline{a_3} \dashv_{\omega_3} \quad \cdots \quad \mathsf{S}[\![s_n]\!]_{\omega_{n-1}} = \overline{a_n} \dashv_{\omega_n}$

$$\mathsf{F}[\![.]\!]_C : f \to r$$

$$\mathsf{F}[\![f]\!]_C = C\#f(this, v)\ \text{:-}\ \blacklozenge.$$

$$\mathsf{M}[\![.]\!]_C : m \to r$$

$$\mathsf{M}[\![\mathtt{def}\ m(\overline{x}) = \{s_1; \ldots; s_n; e\}]\!]_C = m(C, this, \overline{x}, t, \omega, \omega')\ \text{:-}\ \blacklozenge,\ \overline{a_1}, \ldots, \overline{a_n},\ \overline{a}.$$
where $\mathsf{S}[\![s_1]\!]_\omega = \overline{a_1} \dashv_{\omega_1} \quad \cdots \quad \mathsf{S}[\![s_n]\!]_{\omega_{n-1}} = \overline{a_n} \dashv_{\omega_n}$
$\mathsf{E}[\![e]\!]_{\omega_n} = t \dashv_{\omega'} \overline{a}$

$$\mathsf{C}[\![.]\!] : c \to \overline{r}$$

$$\mathsf{C}[\![\mathtt{class}\ C(x_1, \ldots, x_l)\ \mathtt{extends}\ D\ \{f_1, \ldots, f_k, m_1, \ldots, m_n\}]\!] = r_{init},\ \mathsf{F}[\![f_1]\!]_C, \ldots, \mathsf{F}[\![f_k]\!]_C,\ \mathsf{M}[\![m_1]\!]_C, \ldots, \mathsf{M}[\![m_n]\!]_C,\ \overline{r}$$
where $r_{init} = C(this, x_1, \ldots, x_l)\ \text{:-}\ \blacklozenge, C\#f_1(this, x_1), \ldots, C\#f_k(this, x_k), D(this, x_{k+1}, \ldots, x_l).$
$\overline{r} = \langle dispatch_m(C, D)\ \text{:-}\ \varepsilon.\ \mid m \in methodsTrans(C), D = findMethodDef(C, m)\rangle$

$$\mathsf{C}[\![\mathtt{class}\ \mathtt{Object}\ \{\}]\!] = \mathtt{Object}(this)\ \text{:-}\ \blacklozenge.$$

Fig. 6. Generating Datalog terms and atoms for OODL expressions and statements, and generating Datalog rules for OODL fields, methods, and classes.

a method, because the implementing class, receiver object, argument values, and allocation count depend on the calls that occur in the program. The demand transformation will make sure to bind the corresponding logic variables appropriately.

Finally, the translation of classes ties everything together. Moreover, we add a rule for the implicit default constructor: We use the first k constructor arguments to initialize the local fields of C, and pass all subsequent constructor arguments to the constructor of the superclass D. We add dispatch rules for receiver objects of run-time type C for each invocable method m, dispatching to the closest definition in D. Finally, we define the root class `Object` with no arguments, fields or methods.

This completes our translation of OODL to Datalog. For example, we compile method `Succ.add` from above to the following Datalog code.

```
// OOP code: def add(that: Nat): Nat = { val newPred = this.pred.add(that); new Succ(newPred) }
add("Succ", this, that, out, ω0, ω2) :- ◆,  // demand inputs this, that, and ω0
    Succ#pred(this, v),                     // read v = this.pred
    v = OID(C,_), dispatch_add(C, D),       // dispatch this.pred.add(..)
    add(D, v, that, newPred, ω0, ω1),       // invoke this.pred.add(..)
    out = OID("Succ",ω1), ω2 = ω1 + 1,      // create fresh OID, increment allocation counter
    Succ(out, newPred)                      // invoke Succ constructor
```

Note that our translation enjoys separate compilation: Each class can be translated to Datalog separately given the interface of their superclasses. The superclass interfaces are required by *findFieldDef* and *findMethodDef* to find the appropriate field and method definitions. The demand markers ◆ can be eliminated after linking by the demand transformation.

4.3 Eliminating Demand Markers

Our translation uses demand markers to encode the control-flow of OOP. For example, encoded methods are only executed once there is demand for *this*, $\overline{x}$, and ω, which happens when the execution reaches a call of the method. However, demand markers are not essential to Datalog, they can be eliminated using the demand transformation.

The demand transformation [Tekle and Liu 2010] rewrites a Datalog program in a way that enforces a demand-driven execution even when the underlying Datalog engine uses a bottom-up evaluation strategy, which is data-driven. The key idea is to generate auxiliary relations to communicate information from relation call-sites to their definition. To this end, the demand transformation proceeds in three steps.

(1) Demand analysis identifies demanded columns. For `add`, these are *this*, *that*, and ω_0.
(2) Generate an input relation *input_R*($\overline{C}$) for each relation R with demanded columns $\overline{C}$. For each call-site of R, *input_R* has one rule to record the arguments at the call-site.
(3) Replace demand markers by calls to the newly generated input relations.

For example, the demand transformation inserts a call `input_add(this,that,`ω_0`)` in place of ◆ in `add`. `input_add` has one rule for the recursive call-site within `add` and one rule for each external call-site.

5 Compiling Mutable OOP to Datalog

In Section 2, we used mutation to construct abstract syntax graphs and to accumulate dependency edges in a visitor. Indeed, many OOP programs and design patterns rely on mutation, including the observer, iterator, and lazy initialization patterns. The OODL interpreter supports mutation via meta-interpretation. However, it is not obvious how to support mutation in Datalog, since it has a monotonic semantics: The set of facts continuously grows during execution, but facts are never revoked. Thus, how can we invalidate the old state and continue with an updated state? In this section, we extend OODL with mutation and show how to translate mutable OODL to Datalog.

We support two forms of mutation in OODL, namely mutable local variables and mutable fields:

$$\text{(statements)} \quad s ::= \ldots \mid \texttt{var}\ x = e \mid x = e \mid e.f = e$$

While locals can be declared immutable (`val`) or mutable (`var`), we consider all fields to be mutable by default. Below, we explain how to encode mutable variables and fields in Datalog.

5.1 Mutable Local Variables Using SSA

We handle mutable local variables using the Static Single Assignment (SSA) method [Cytron et al. 1991; Rosen et al. 1988]. A program is in SSA when each variable is assigned exactly once and, thus, no mutation occurs. Effectively, an SSA transformation eliminates mutable variables and uses a family of *immutable* variables instead, one for each assignment. Consider the following OODL program before and after the SSA transformation:

```
var a: Int = 3; a = 4; var b: Int = a
```

```
val a0: Int = 3; val a1: Int = 4; val b0: Int = a1
```

The program declares two integer variables `a` and `b`, reassigns the value of `a`, and assigns the final value to `b`. By transforming the program to SSA form, we distinguish two versions of variable `a` and only assign the final version of `a` to `b`. We can compile the resulting immutable program using the translation from Section 4. But for languages with conditional control flow like ours, an SSA transform also generates ϕ-nodes to select a variable version. In the following example, we count the number of edges where `e1 == e2`. The ϕ-node for `c$3` selects one of the versions of `c`:

```
def count(e1: Edge, e2: Edge): Int =
  var c: Int = 0
  if (e1 == e2) { c = 1 }
  else { c = 2 }
  c
```

```
def count(e1: Edge, e2: Edge): Int =
  val c0: Int = 0
  if (e1 == e2) { val c1: Int = 1 }
  else { val c2: Int = 2 }
  val c3: Int = ϕ(e1 == e2, c1, c2); c3
```

Thus, the SSA transformation eliminates mutable variables but introduces ϕ-nodes. But, interestingly, it is not necessary to support ϕ-nodes in Datalog directly. This is because each branch of a conditional construct translates to its own Datalog rule, allowing us to eliminate ϕ-nodes on-the-fly. For our example, we get two rules, where the last atom selects the appropriate version of `c`:

```
count(e1, e2, c3) :- c0 = 0, e1 = e2, c1 = 1, c2 = 2, c3 = c1.
count(e1, e2, c3) :- c0 = 0, e1 ≠ e2, c1 = 1, c2 = 2, c3 = c2.
```

5.2 Mutable Fields Using Timestamps

We used SSA to distinguish different versions of local variables. SSA is applicable to local variables since we can determine precise use-def chains for each variable statically. In contrast, use-def chains are (largely) intractable for fields, because we need to precisely predict the control and data flow for invocations of setter and getter methods. Therefore, we use a dynamic approach for handling mutable fields: timestamps (μ). Consider the following example with a mutable field `Num.value`:

```
class Num(n: Int):
    var value: Int = n
val n1 = new Num(10)
n1.value = 20
val n2 = new Num(10)
n2.value = 50
n1.value = 30
```

Object	μ	value
`n1 = OID(Num, 0)`	0	10
`n1 = OID(Num, 0)`	1	20
`n2 = OID(Num, 1)`	0	10
`n2 = OID(Num, 1)`	2	50
`n1 = OID(Num, 0)`	3	30

$$\mathsf{F}[\![\,.\,]\!]_C : f \longrightarrow \overline{r}$$
$$\mathsf{F}[\![f]\!]_C = C\#f(\mathit{this}, \mu, v) \text{ :- } \blacklozenge.$$
$$\quad C\#f_\infty(\mathit{this}, \mu, max(\mu')) \text{ :- } \blacklozenge,\ C\#f(\mathit{this}, \mu', \_), \mu' < \mu.$$
$$\mathsf{E}[\![\,.\,]\!]^{\mu}_{\omega} : e \longrightarrow t\ \dashv^{\mu}_{\omega}\ \overline{a}$$
$$\mathsf{E}[\![e.f]\!]^{\mu}_{\omega} = y\ \dashv^{\mu'}_{\omega'}\ \overline{a},\ D\#f_\infty(t, \mu', \mu_\infty),\ D\#f(t, \mu_\infty, y)$$
$$\quad \text{where } \mathsf{E}[\![e]\!]^{\mu}_{\omega} = t\ \dashv^{\mu'}_{\omega'}\ \overline{a}, \quad C = typeof(e), \quad D = \mathit{findFieldDef}(C, f), \quad y \text{ is fresh}$$

$$\mathsf{S}[\![\,.\,]\!]^{\mu}_{\omega} : s \longrightarrow \overline{a}\ \dashv^{\mu}_{\omega}$$
$$\mathsf{S}[\![e_1.f = e_2]\!]^{\mu}_{\omega} = \overline{a_1},\ \overline{a_2},\ D\#f(t_1, \mu_2, t_2)\ \dashv^{\mu_2+1}_{\omega_2}$$
$$\quad \text{where } \mathsf{E}[\![e_1]\!]^{\mu}_{\omega} = t_1\ \dashv^{\mu_1}_{\omega_1}\ \overline{a_1}, \quad \mathsf{E}[\![e_2]\!]^{\mu_1}_{\omega_1} = t_2\ \dashv^{\mu_2}_{\omega_2}\ \overline{a_2}, \quad C = typeof(e), \quad D = \mathit{findFieldDef}(C, f)$$

Fig. 7. Compiling fields, field reads, and field writes to Datalog.

On creation, we initialize mutable fields at timestamp 0. But when updating a field, we use the current timestamp to tag the new value. We collect all timestamped values in a singe table per field. This way, the above program induces the Datalog table `Num#value` on the right. That is, each field store adds a row to the table and increments the timestamp. Over time, we monotonically accumulate all values of a field at different timestamps. Note that we use a single global mutation counter shared by all fields and objects as timestamp. Other, more structured timestamps can be used as long as OID and timestamp uniquely determine the field's value.

We conducted microbenchmarks on the Peano number example, comparing both approaches. This involved two aspects: (1) scaling the number of mutations per object and (2) scaling the number of object allocations. A per-object timestamp is best suited when repeatedly modifying the same object. In such cases, the size of the counter remains unaffected. However, multiple allocations increase the counter's size, thereby affecting performance. Conversely, our global counter performs best when the number of mutations per object is small, since we only need to consider a few entries per object. Object allocations do not impact performance, since we can filter based on the object.

One downside of the global timestamp is that reading a field is more complicated. When reading a field, we need to find the most recently written value for the given object. But our timestamp represents the last time *any* field has been written. For example, when reading `n2.value` after executing all five statements from above, we need to read the field at timestamp 2, even though the current timestamp has moved on. Therefore, we generate an additional relation $C\#f_\infty$ to compute the most recently used timestamp for an object.

Figure 7 shows the translation semantics for mutable fields. Indeed we generate two relations per field: One relation $C\#f(\mathit{this}, \mu, v)$ to collect all values v with their timestamp μ written to field $C.f$ of object *this*. And one relation $C\#f_\infty$ to compute the most recently written timestamp for an object: Given the current timestamp μ, we find all timestamps μ' at which $\mathit{this}.f$ was written before. We then select the maximal μ' using aggregation in Datalog. With these relations in place, we can read a field $e.f$ by fetching the most recently written timestamp μ_∞ from $C\#f_\infty$ and then reading the value v at that timestamp from $C\#f$.

Note that we modified the translation functions for expressions and statements to thread the timestamp μ alongside the allocation count ω. The only place the timestamp changes is during field assignment $e_1.f = e_2$: After inserting a row into $C\#f$ at the current timestamp μ_2, the translation yields an incremented timestamp $\mu_2 + 1$. Besides that, the translation handles μ analogously to ω.

We extend our example from above with an increment method to illustrate the translation:

```
class Num(n: Int):                    Num(this, x) :- ◆, Num#value(this, 0, x), Object(this).
  var value: Int = n                  Num#value(this, μ, v) :- ◆.
  def inc(): Int =                    Num#value∞(this, μ, max(μ')) :- ◆, Num#value(this,μ',_), μ' < μ.
    val old = this.value              inc("Num", this, old, ω, ω, μ, μ') :-
    this.value = old + 1                  Num#value∞(this, μ, μold), Num#value(this, μold, old),
    old                                   Num#value(this, μ, old + 1), μ' = μ + 1.
```

We can see here that the constructor initializes `Num.value` at timestamp 0. Method `inc` takes the current timestamp μ as input, finds the most recently written timestamp μ_{old} for `this.value` and reads it. It then writes a new value `old + 1` and increments the timestamp. Of course, our translation works for arbitrary mutable OODL programs with many mutable fields and where field assignments do not necessarily respect encapsulation. We impose no restrictions on the use of mutable fields.

6 Object-Oriented Fixpoint Computations

We want to deeply integrate Datalog and OOP, to exploit the advantages of OOP and the efficient fixpoint computations of Datalog. To this end, we first need to understand the design space regarding what operations are allowed within fixpoint computations and how these translate to Datalog.

6.1 Fixpoint Computations in OODL

In general, a lot of programs in OOP that freely mutate the heap have no unique least fixpoint. For example, consider a simple OOP program that continuously modifies the field of an object. There exists no unique least fixpoint, unless the field's value becomes stable eventually. Indeed, if such a fixpoint exists, our OODL interpreter will find it, since it iterates until the heap stabilizes.[1] However, the same fixpoint guarantees do not hold for our compiled OODL code. To find a fixpoint, we implicitly rely on the semi-naïve evaluation of the generated Datalog code, not on the stabilzation of the heap. Accordingly, the existence of a fixpoint depends on the Datalog encodings we choose for translating OODL. In particular, our encodings of allocation and mutation rely on counters threaded through the Datalog program. These counters increase whenever an object is created or a field is mutated, even if they have no observable impact on the heap. In our example, where a field eventually stabilizes in OODL, the mutation counter in the Datalog translation still diverges and no fixpoint exists. The same applies to allocation.

It is not obvious how to encode allocation or mutation differently in Datalog to prevent these problems. Instead, we introduce abstractions in OODL that avoid these issues while largely retaining expressiveness. Specifically, we add algebraic data types (case classes) to OODL, whose instances do not have object identity and thus can be created during fixpoint computations. And we define a form of monotonic mutation that does not inhibit finding a fixpoint. Before we describe these abstractions in detail, we extend OODL with recursively defined sets to mirror Datalog's relations.

6.2 Fixpoint Computation with Sets

Figure 8 introduces standard OOP constructs for working with immutable sets: set literals, union, and set comprehensions. As usual in imperative languages, we also permit iteration on sets as a statement; a guard is not necessary here since we already support if-statements. Figure 8 also shows how to translate set expressions and iterators to Datalog. For set literals, we generate a Datalog term y whose binding is non-deterministically set to any t_i. Similarly, for set union, y propagates

[1] Technically, the interpreter does not always find a fixpoint, since we do not perform any garbage collection to remove unreachable objects from the heap. This is a limitation of our implementation only.

$$\begin{array}{ll}\text{(expression)} & e ::= \ldots \mid \mathsf{Set}(\overline{e}) \mid e \mathbin{+\!\!+} e \mid \mathsf{for}\ (x \leftarrow e\ \mathsf{if}\ e == e)\ \mathsf{yield}\ e \\ \text{(statements)} & s ::= \ldots \mid \mathsf{for}\ (x \leftarrow e)\ s\end{array}$$

$$\mathsf{E}[\![\,\mathsf{Set}(e_1,\ldots,e_n)\,]\!]^{\mu}_{\omega} = y \dashv^{\mu_n}_{\omega_n} \overline{a_1},\ldots,\overline{a_n},\ (y = t_1 \vee \cdots \vee y = t_n)$$
$$\text{where } \mathsf{E}[\![\,e_1\,]\!]^{\mu}_{\omega} = t_1 \dashv^{\mu_1}_{\omega_1} \overline{a_1} \quad \cdots \quad \mathsf{E}[\![\,e_n\,]\!]^{\mu_{n-1}}_{\omega_{n-1}} = t_n \dashv^{\mu_n}_{\omega_n} \overline{a_n}$$

$$\mathsf{E}[\![\,e_1 \mathbin{+\!\!+} e_2\,]\!]^{\mu}_{\omega} = y \dashv^{\mu_2}_{\omega_2} \overline{a_1},\ \overline{a_2},\ (y = t_1 \vee y = t_2)$$
$$\text{where } \mathsf{E}[\![\,e_1\,]\!]^{\mu}_{\omega} = t_1 \dashv^{\mu_1}_{\omega_1} \overline{a_1}, \quad \mathsf{E}[\![\,e_2\,]\!]^{\mu_1}_{\omega_1} = t_2 \dashv^{\mu_2}_{\omega_2} \overline{a_2}$$

$$\mathsf{E}[\![\,\mathsf{for}\ (x \leftarrow e_1\ \mathsf{if}\ e_2 == e_3)\ \mathsf{yield}\ e_4\,]\!]^{\mu}_{\omega} = t_4 \dashv^{\mu'}_{\omega'} \overline{a_1},\ x = t_1,\ \overline{a_2},\ \overline{a_3},\ t_2 = t_3,\ \overline{a_4}$$
$$\text{where } \mathsf{E}[\![\,e_1\,]\!]^{\mu}_{\omega} = t_1 \dashv^{\mu'}_{\omega'} \overline{a_1}, \quad \mathsf{E}[\![\,e_2\,]\!]^{\mu'}_{\omega'} = t_2 \dashv^{\mu'}_{\omega'} \overline{a_2}, \quad \mathsf{E}[\![\,e_3\,]\!]^{\mu'}_{\omega'} = t_3 \dashv^{\mu'}_{\omega'} \overline{a_3}, \quad \mathsf{E}[\![\,e_4\,]\!]^{\mu'}_{\omega'} = t_4 \dashv^{\mu'}_{\omega'} \overline{a_4}$$

$$\mathsf{S}[\![\,\mathsf{for}\ (x \leftarrow e)\ s\,]\!]^{\mu}_{\omega} = \overline{a_1},\ \overline{a_2} \dashv^{\mu'}_{\omega'}$$
$$\text{where } \mathsf{E}[\![\,e_1\,]\!]^{\mu}_{\omega} = t_1 \dashv^{\mu'}_{\omega'} \overline{a_1}, \quad \mathsf{S}[\![\,s\,]\!]^{\mu'}_{\omega'} = \overline{a_2} \dashv^{\mu'}_{\omega'}$$

Fig. 8. Set constructs and their translation to Datalog.

all t_1 or t_2 non-deterministically. Unfortunately, the non-deterministic set semantics of Datalog are problematic when processing sets in a language with side-effects.

For example, consider the program `for (x <- Set(1,2,3)) yield new Num(x)`, which creates three `Num` objects. Our translation would need to guarantee that each `Num` object is assigned a unique OID. This requires deterministic threading: Process the elements of the set one by one in some deterministic order and thread the allocation counter between iterations. However, as illustrated in Section 2, this strategy would still fail for recursive fixpoint computations, since allocation and mutation counter are unbounded and the fixpoint computation diverges.

Therefore, we use a simplified translation that prohibits side-effects in non-deterministic computations. When translating set comprehensions or iterators, only e_1 may induce side effects that manipulate the allocation counter ω or mutation counter μ. All other computations non-deterministically depend on the set elements of e_1 and therefore may not induce side effects: Their translation takes and yields ω' and μ' unchanged. In the remainder of this section, we adapt our language translation to introduce alternatives for operation that induce side effects.

6.3 Allocating Algebraic Objects in Fixpoint Computations

A deep integration of OOP and Datalog requires allocating new objects inside fixpoint computations. When interpreting OODL, allocating objects might expand the heap, thereby preventing a fixpoint from being found. In our Datalog translation, allocating a new object causes a different problem: We increase the allocation counter. Our translation function from Figure 8 precludes such code (and fixpoint computations would diverge otherwise). To support object allocation in fixpoint computations, we resort to a side-effect free allocation scheme that does not require OIDs.

To this end, we extend OODL with algebraic data types in the form of Scala-like case classes. Case classes are immutable and their equality is structural: Objects that instantiate the same case class are equal if and only if their fields have equal values. Our Datalog translation can allocate instances of case classes without using the allocation counter, since they do not require an OID. For example, we support code that constructs `Path` objects in a non-deterministic order:

```
case class Path(start: Node, end: Node)
def directPaths(start: Node, edges: Set[Edge]): Set[Path] =
    for (e <- edges if e.from == start) yield new Path(start, e.to)
```

$$\mathsf{E}[\![\,\mathsf{new}\ C(e_1,\ldots,e_n)\,]\!]^{\mu}_{\omega} = y \dashv^{\mu_n}_{\omega_n}\ \overline{a_1},\ldots,\overline{a_n},\ y = \mathsf{SID}(C,(t_1,\ldots,t_n))$$
$$\text{where } \mathsf{E}[\![\,e_1\,]\!]^{\mu}_{\omega} = t_1 \dashv^{\mu_1}_{\omega_1}\ \overline{a_1} \quad \cdots \quad \mathsf{E}[\![\,e_n\,]\!]^{\mu_{n-1}}_{\omega_{n-1}} = t_n \dashv^{\mu_n}_{\omega_n}\ \overline{a_n} \quad C \text{ is a case class}, \quad y \text{ is fresh}$$
$$\mathsf{E}[\![\,e.f\,]\!]^{\mu}_{\omega} = y_i \dashv^{\mu'}_{\omega'}\ \overline{a},\ t = \mathsf{SID}(\_,(\ldots,y_i,\ldots))$$
$$\text{where } \mathsf{E}[\![\,e\,]\!]^{\mu}_{\omega} = t \dashv^{\mu'}_{\omega'}\ \overline{a} \quad C = \mathit{typeof}(e), \quad C \text{ is a case class}, \quad i = \mathit{findIndex}(C,f)$$

Fig. 9. Instances of case classes can be allocated without side-effect using a structural ID (SID).

```
def paths(start: Node, edges: Set[Edge]): Set[Path] = directPaths(start, edges) ++
    for (p1 <- directPaths(start, edges)) yield
        for (p2 <- paths(p1.end, edges)) yield new Path(start, p2.end)
```

This program complies with our translation function from Figure 8, since we only instantiate case classes. The exact semantics of case classes follows below. But also note the unbounded recursion in `paths`, which would not normally terminate in an OOP language without least fixpoint semantics. The interpreted version of this OODL program terminates, since we preclude case classes from the fixpoint computation by not storing them in the heap. The Datalog translation also terminates and computes the transitive closure of `edges` because the allocation and mutation counter are stable.

Figure 9 adds two new translation rules for creating instances and reading fields from case classes. The key ingredient for encoding case classes are structural identities (SID), which we use instead of OIDs here. An SID consists of the name of a case class and a tuple of values, one for each field of the case class. We use SIDs to identify and to compare case-class objects, since structurally equal objects have equivalent SIDs. We also use SIDs to lookup fields from a case-class object as also shown in Figure 9. That is, for case classes, we do not store fields in relations, since that would duplicate the SID. In this subsection, we showed that case classes enable object allocation within fixpoint computations while still supporting inheritance, field reading, and dynamic dispatch. Next, we propose a technique for reintroducing mutation within fixpoint computations.

6.4 Relaxed Mutation Using Mono Types — Monotonically Mutable Data Types

A Datalog fixpoint computation only yields a result when no new tuples are derivable. Since we use and increment a mutation counter when writing to a field, mutation precludes fixpoint computations. To compute fixpoints in Datalog despite mutation, we propose using relaxed mutation in a novel abstraction of *monotonically mutable data types*, or *mono types* for short.

A mono type is a mutable container to which values can be added, but never removed. However, mono types are not collections: They can process the added values and provide derived results. Technically, a mono type implements the following interface:

```
package mono
trait Type[V, R]:
  def +=(v: V): Unit
  def result: R
```

```
∀ m: mono.Type[V,R], v: V.
    m.result ⊑ (m += v; m.result)
```

Operation `+=` accepts values of type `V` and processes them internally. Operation `result` provides the result of the internal processing. However, the operations of a mono type are restricted in one important way: Observations made through `result` must be monotonically increasing according to some partial order $\sqsubseteq$ on type `R`. This means, previous observations are subsumed by later observations after additional elements have been added to the mono type. We deliberately refrain from specifying any other properties about the memory ordering for mono types. For example, we

do not specify that operation `+=` commutes (the order in which elements are added is irrelevant). The monotonicity property suffices to support integration into Datalog.

We integrate mono types into Datalog without using a mutation counter. Instead, the insertion of elements to a mono type is asynchronous. However, the monotonicity of mono types ensures that the final result is consistent eventually. Since Datalog computes a fixpoint, it is this final result that other parts of the computation observe in the end, making them eventually consistent themselves.

To ensure termination of fixpoint computations, OODL developers must use mono types in a way that prevents infinite ascending chains of observations $r_1 \sqsubset r_2 \sqsubset \cdots$ derived by `m.result`. Developers can prevent such chains by either (i) only adding finitely many elements to a mono type or (ii) by using a mono type that satisfies the ascending chain condition. In particular, we allow computations to recursively feed mono-type results back into the mono type as input, but the mono-type result has to become stable after finitely many iterations. That is, mono types generalize lattice-based recursive aggregation as explored by prior work [Madsen et al. 2016; Szabó et al. 2018].

6.5 Examples of Mono Types

We specified mono types algebraically through the signatures of `+=` and `result` and their monotonicity property. While the mono type interface is predefined in OODL, concrete mono type definitions are user-defined. Here, we provide various examples to illustrate the flexibility of our design.

Primitive Mono Types. We start with a few simple mono types that usually can be found as built-in aggregations in Datalog systems. Indeed, we do not compile these mono types to Datalog, but use them as aggregators in the underlying Datalog system (here: in Scala).

```
class Count extends mono.Type[Int, Int]:
    var count: Int = 0 // Scala mutation
    def +=(v: Int): Unit = this.count += 1
    def result: Int = this.count
```

```
class Sum extends mono.Type[Int, Int]:
    var sum: Int = 0 // Scala mutation
    def +=(v: Int): Unit = this.sum += Math.abs(v)
    def result: Int = this.sum
```

The count mono type increments an internal counter for each element added, and the sum mono type adds the element to compute the sum of all elements. Both mono types internally rely on a mutable Scala field. This mutation is justified since we do not rely on a strong ordering for reads or writes, but only require monotonicity, which is given here. We can also combine mono types to form more complex mono types. For example, we could introduce an `Average` mono that tracks both the sum and count of elements. Note however, that computing the mean by dividing the sum result by the count does not grow monotonically and as such, must not be part of the mono type.

Lattice Mono Types. Due to the popularity of using Datalog for program analysis, lattice-based aggregation is of particular interest [Madsen et al. 2016; Szabó et al. 2018]. Each bounded semilattice $(L, \bot, \sqcup)$ forms a mono type as the generic construction on the right illustrates. That is, we compute the join over all added elements, which is monotonic by definition.

```
class L extends mono.Type[L, L]:
    var state: L = ⊥
    def +=(v: L): Unit = this.state
     = this.state ⊔ v
    def result: L = this.state
```

Relational Mono Types. So far, all mono types we showed were defined by implementing the `+=` and `result` operations and translated to (recursive) aggregation in Datalog. However, there is a special class of mono types that does not require aggregation, but can be encoded through regular Datalog relations. We call these mono types *relational*.

The simplest relational mono type is `mono.Set`, which simply collects all added elements. That is, `mono.Set[V]` is a `mono.Type[V, Set[V]]` where we can add values of type `V` and observe the collected values of type `Set[V]`. While it is possible to define `mono.Set` through `+=` and `result`, it is more efficient and convenient to reuse Datalog's relations. In particular, if field $C.f$ has type `mono.Set[V]`, then the

corresponding field relation has type $C\#f \subset \text{OID} \times V$, which can accommodate any number of $v \in V$ per object. Reading `result` for a `mono.Set` then yields all $v \in V$ associated with *this*. All `mono.Set` types are monotonic by construction, because Datalog relations are monotonic under set inclusion.

Another relational mono type is `mono.Map[K, V]`, which associates a value of type V for each key of type K in the map. That is, `mono.Map[K,V]` is a `mono.Type[(K,V), Map[K,V]]`, which accepts (K, V) pairs and yields the entire map. We can lower a `mono.Map` to Datalog relation with a functional dependency. Given field $C.f$ has type `mono.Map[K,V]`, then the corresponding field relation has type $C\#f \subset \text{OID} \times K \times V$ with functional dependency $(\text{OID}, K) \mapsto V$. That is, if $(o, k, v_1) \in C\#f$ and $(o, k, v_2) \in C\#f$, then $v_1 = v_2$, exactly as expected for a map.

However, is each `mono.Map` monotonic, and under which ordering? Consider the following program:

```
val m = new mono.Map[String, Int]();  m += ("foo", 1);  m += ("foo", 2);  m.result
```

What happens when we add a second value for key `"foo"` in `m`? To resolve this situation, we require that `V` itself must be a mono type. That is, given `M <: mono.Type[V,R]`, we can provide `mono.Map[K, M] <: mono.Type[(K,V), Map[K,R]]`. Here, `M` is a mono type that accepts values of type `V` and provides observations of type `R`. Then, a `mono.Map[K, M]` accepts key-value pairs of type `(K,V)` as input and provides observations of type `Map[K,R]`: One `M` observation for each key. For example, `mono.Map[String, mono.Sum]` is a `mono.Type[(String,Int), Map[String,Int]]` that sums up all values associated with same key. For this mono type, the example program associates `"foo"` with value `3`.

There are many more interesting relational mono types, such as `mono.Map[K, mono.Set[V]]` to encode multimaps. The lowering of relational mono types to Datalog is not always easy, preserving monotonicity and functional dependencies correctly. In our implementation, we currently only support `mono.Set` and `mono.Map[K, M]` for non-relational (i.e., primitive and lattice) mono types `M`. We leave a detailed study of the theory and implementation of mono types as future work.

7 Implementation and Evaluation

We have implemented an interpreter for OODL that relies on naïve fixpoint iteration. We also implemented a compiler that translates OODL programs to Datalog code, which we can run using the IncA framework [Pacak et al. 2022]. Our compiler generates Datalog code as described in this paper but supports some additional features. For example, we encode the inheritance hierarchy of classes as facts in the extensional database to support instance-of tests and casting. The code of the implementation and case studies are available at https://gitlab.rlp.net/plmz/inca-scala/-/tree/objectoriented-interp.

In the remainder of this section, we present four case studies that demonstrate the usability of OODL. We use two of the case studies for a performance evaluation, where we measure the running time of (i) the compiled OODL code, (ii) the interpreted OODL code, and (iii) a handwritten Datalog solution (for one of the case studies). Even though our compiler only applies the most basic optimizations, the generated Datalog code is significantly faster than both: handwritten Datalog and interpreted OODL. Additionally, we discuss the effort of porting the case studies to traditional OOP by implementing them in Scala.

7.1 Case Studies

We implement four program analyses in OODL using the visitor pattern. The visitor pattern not only provides reusable traversal rules, but also allows us to modularly add analyses.

Dependency Analysis. We implement the dependency analysis from Figure 1 in OODL. Step (1) and (3) are unchanged: constructing an abstract syntax graph (ASG) and computing the transitive closure over a set of edges. But in Step (2), we modified the code to keep the allocation and mutation

counters stable in the fixpoint computation and to guarantee termination. Specifically, we change class `Edge` into a case class to keep the allocation counter stable, and we collect edges in a `mono.Set` during the fixpoint computation.

```
case class Edge(val from: Def, val to: Def)
class DependencyAnalysis extends Visitor:
    val edges: mono.Set[Edge] = new mono.Set[Edge]()
    override def visitVar(v: Var, d: Def): Unit =
        if (v.target != noDef) { edges += new Edge(d, v.target); v.target.accept(this) }
```

Under normal OOP-semantics, such fixpoint computation would not terminate. For OODL programmers, fixpoint computations are declarative.

Control-flow Analysis. Next, we implement a simple control-flow analysis for a `While` language with arithmetic expressions, assignments, conditionals, and while-loops. We write a visitor to collect control-flow edges in a `mono.Set`:

```
class CfgVisitor extends Visitor:
    val cfg: mono.Set[(Stm, Stm)] = new mono.Set()
    override def visitSequence(s: Seq): Unit = for (l <- s.s1.last) this.cfg += (l, s.s2.first)
    override def visitIf(s: If): Unit = this.cfg ++= Set((s, s.thn.first), (s, s.els.first))
    override def visitWhile(s: While): Unit =
        this.cfg += (s, s.body.first); for (last <- s.body.last) this.cfg += (last, s)
```

We use inheritance and rely on the `Visitor` pattern to visit each node in the AST, but only overwrite the behavior for the three control statements. We use a `mono.Set` to store pairs that represent a flow from one statement to another. Each time we visit a control statement, we add edges to our set.

Compared to traditional Datalog, the OODL implementation has two advantages. First, we reuse the code for the AST traversal through the visitor superclass, allowing us to focus on control statements. Second, we encapsulate the `cfg` field, which is intended to be only used by the methods of `CfgVisitor`. In particular, all visit methods write to the same field without requiring state passing. It is not obvious how to obtain similar advantages in plain Datalog unless relying on our encodings.

Constant Propagation Analysis. Finally, we implemented two flow-sensitive data-flow analysis for the `While` language: a constant analysis and a sign analysis. For the constant analysis, we define a lattice `ConstantLat` as an abstract class with three concrete subclasses: `Bottom`, `SomeConstant(i)` for any integer `i`, and `Top`. The ordering and join operations are defined as usual. As described in Section 6.5, we wrap this lattice in a mono type called `mono.Constant`:

```
class mono.Constant extends mono.Type[ConstantLat, ConstantLat]:
    var state: ConstantLat = new Bottom()
    def +=(c: ConstantLat): Unit = this.state.join(c)
    def result: ConstantLat = this.state
```

Next, we provide an abstract interpreter `aevalConst` for expressions of the `While` language. Numeric literals evaluate to constants, and arithmetic operators are lifted to the constant lattice as usual. The abstract interpreter takes the current statement and `ValueManager` as input to support variable lookup. The `ValueManager` maintains the abstract value of variables. Finally, each `Stm` implements a transfer method that updates variable values in the `ValueManager`.

```
class Num(num: Int) extends Exp:
    override def aevalConst(p: ValueManager, s: Stm): ConstantLat = new SomeConstant(this.num)
class Var(name: String) extends Exp:
    override def aevalConst(p: ValueManager, s: Stm): ConstantLat = p.getValue(s, this.name)
class ValDef(name: String, val rhs: Exp) extends Stm(...):
```

```
override def transfer(pred: Stm, p: ValueManager): Unit =
  for (v <- allVars if v != this.name) p.put(this, v, p.getValue(pred))
  p.put(this, this.name, this.rhs.aevalConst(p, s))
```

Using this setup, we can describe data-flow analyses modularly by implementation of the `ValueManager` abstract class. For the flow-sensitive constant analysis, we track a variable's abstract value at each statement using a nested `mono.Map`. To initialize the analysis, we insert bottom values for each variable at the program's initial statement (not shown). We can then run the analysis by visiting each relevant transfer function once. The fixpoint semantics takes care of the rest: iterating the transfer functions until the `mono.Map` becomes stable.

```
class FSConstantPropagation extends ValueManager:
    var vars: mono.Map[Stm, mono.Map[String, mono.Constant]] = new mono.Map()
    override def getValue(s: Stm, v: String): ConstantLat = this.vars.get(s).get(v).result
    override def put(s: Stm, v: String, m: ConstantLat): Unit = this.vars(s) += (v, m)
    def run(): Unit = for ((from, to) <- this.cfg) to.transfer(from, this)
```

Sign Analysis. We also implemented a flow-sensitive sign analysis based on a `mono.Sign` type. To this end, we only had to implement a different `ValueManager`, but could reuse all other code.[2] These case studies show how the design of OODL enables OOP-style program organization and maintainability while leveraging Datalog's fixpoint semantics.

7.2 Comparison to Traditional OOP

We ported all case studies to traditional OOP by implementing them in Scala. However, OOP has no built-in support for fixpoint computations. That is, we need to introduce explicit data structures and logic to handle fixpoints. In the remainder of this section, we describe these changes in detail.

Dependency Analysis. The dependency analysis requires explicit fixpoint handling at multiple places. In step (3), the `dependencies` method automatically terminates upon reaching a fixpoint. To achieve the same in OOP, we need to introduce a worklist and track the visited dependencies. However, this alone doesn't guarantee termination of the entire analysis. In step (2), we collect all dependency edges in a `mono.Set` each time we visit a variable. This computation also involves a fixpoint, albeit less apparent. When visiting a variable, the accept method is called on its target definition. The target definition in turn invokes `accept` on its expression, potentially leading to an infinite loop if the expression is a variable. To circumvent this loop, we also track all visited definitions in `visitVar` and only process target definitions that have not been visited before.

Control-flow Analysis. Porting the control-flow analysis to Scala is almost verbatim. This is because computing the control-flow doesn't require a fixpoint computation. We only replace the `mono.Set` with an immutable Scala set. This shows how closely OODL captures OOP.

Constant and Sign Analysis. For our data-flow analyses, OODL iterates the `transfer` function until a fixpoint is reached. This iteration has to be made explicit when translating the case study to Scala: Whenever the abstract value of a variable changes, we process the control-flow graph once more. Moreover, we must also replace all mono types. In particular, instead of representing `vars` with a `mono.Map`, we use an immutable Map and instead of `mono.Constant` we use a normal class. However, this also means we now need to manually join the values before adding them to `vars`, whereas in OODL this is implicitly defined by `mono.Constant`.

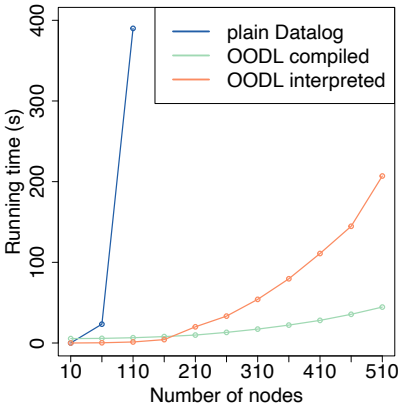
[2]Technically, we changed their type signatures since OODL lacks generics currently, but the implementation did not change.

Summary. In conclusion, porting our case studies to traditional OOP necessitates explicit algorithms and data structures for handling fixpoint computations. This process entails a careful analysis of each program to correctly identify fixpoints, followed by implementing changes throughout the whole programs to guarantee termination. While the amount of changed code is small (~50 LOC each), the code for fixpoint computations is complex and requires non-local reasoning. In OODL, all of this reasoning is implicit, since fixpoints are computed declaratively.

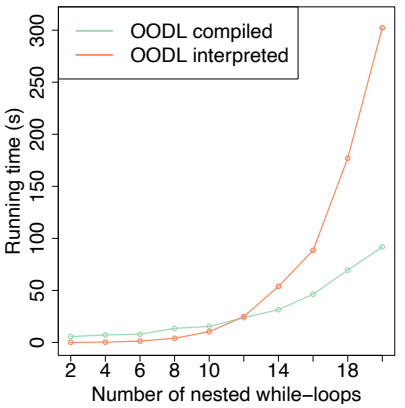
7.3 Performance Evaluation

We developed a compiler for OODL to outperform our fixpoint interpreter from Section 3. By targeting Datalog, the compiled code should benefit from Datalog's semi-naïve evaluation. For the dependency analysis we also measured the execution time of a handwritten Datalog version that we implemented as idiomatically as possible in pure Datalog without relying on our encodings. For sufficiently complex input programs, the compiled code clearly outperforms the interpreter and the handwritten Datalog program. While outperforming the interpreter is expected, outperforming the handwritten Datalog code seems surprising. This performance difference stems from the fact, that the handwritten analysis computes 5.5x more tuples than the OODL implementation. OODL uses a mono type to collect all edges in a single place, while the handwritten Datalog program propagates all edges throughout the program. This shows, that better abstractions can improve performance.

We also measured the performance for the constant and sign analysis, but did not develop a handwritten solution in plain Datalog. We show the results for the constant analysis on the right; the sign analysis yielded similar results. It is clear that the compiled code outperforms the interpreted version of OODL. We also compared these running times to the handwritten Scala code with explicit fixpoint handling: The Scala code only required one second to execute the analysis. However, we know from the literature that optimized static analyses written in Datalog have competitive performance [Bravenboer and Smaragdakis 2009; Szabó et al. 2021]. Our Datalog compiler currently only applies rudimentary optimizations. We expect significantly performance improvements of the compiled code through inter-relational optimizations, which we will explore in future work.



(a) Dependency analysis



(b) Constant analysis

Fig. 10. Running times of dependency and constant analysis

8 Related Work

We study the deep integration of Datalog and OOP to enable fixpoint computations in OOP and OOP abstractions in Datalog. To this end, we propose the design of OODL. Our design improves over prior work in unique ways: (i) dynamic allocation of objects at runtime, (ii) dynamic dispatch of virtual method calls based on the object's run-time type, (iii) mutation of local variables and

fields, (iv) Datalog-style fixpoint computations over case classes, and (v) mono types to support relaxed mutation in fixpoint computations. We discuss related work in the remainder of this section.

Abiteboul et al. [1993] extend Datalog with OOP features such as classes, inheritance, objects, and methods. They translate their extension to Datalog with stratified negation, but do not provide an implementation. In contrast, OODL is a useable, Turing-complete object-oriented language that is compiled to Datalog, without negation but with constructors and aggregation. Abiteboul et al. have a notion of object identity, but all objects must be known at compile time. Their approach also supports *dynamic dispatching*, but uses class guards to find the most specific implementation at run-time. This requires negation, which is problematic when simulating control flow, because the demand transformation introduces additional cycles, which makes the program unstratified [Tekle and Liu 2019]. We follow a different approach, namely creating a dispatch table at compile time.

QL [Avgustinov et al. 2016] is an object-oriented frontend for Datalog with classes, inheritance, and methods, but without mutation. Classes provide a characteristic predicate to enumerate all their "objects", which are immutable values. In fact, any value that satisfies the characteristic predicate is an instance of said class. QL does not have dynamic allocation with unique object identities. Technically, QL operates on primitive values, algebraic data and objects already stored in the extensional database. Objects in OODL are always dynamically created, without relying on preexisting data from the extensional database. QL supports member predicates defined inside a class, which behave similar to methods in OOP. At runtime, a member predicate call dispatches to all most-applicable targets, potentially more than one for a single object. This is in contrast to most OOP semantics as implemented in OODL, where each method call executes exactly one concrete implementation, that depends on the state of an object. Similar to case classes, QL also supports algebraic data types (ADTs) represented by a structural identity and created at run-time [Schäfer et al. 2017]. However, ADTs do not directly exhibit class-like behavior, as they lack support for member predicates. Yet, a class can extend branches of an algebraic data type to add member predicates to it. Thus, users bear the responsibility of introducing class-like behavior to ADTs. In OODL, case classes behave like normal classes; they support field reads, dynamic dispatch, and inheritance out of the box. In summary, the distinct flavor of "objects" in QL and the missing mutation differ from most OOP languages. It is therefore not clear which OOP design patterns carry over to QL.

Dedalus [Alvaro et al. 2011b] extends Datalog to reason about distributed systems by introducing a notion of time to Datalog. A tuple is only valid at a specific point in time, marked by a timestamp. To persist a tuple over time, it must be re-derived for each following timestamp. To invalidate a tuple, they introduce rules that prevent re-deriving the tuple for all following timestamp. OODL's implementation of mutability has some similarities to Dedalus. In contrast to our concept of time, Dedalus' persistent strategy is more complicated and requires more memory. Dedalus re-derives a tuple in every iteration, even if it has not changed. We only store a new value for a field when the field is written to, which makes writing more efficient. As a downside, our strategy complicates and slows down field reads, since we need an aggregation to find the latest timestamp.

JastAdd [Magnusson et al. 2007] is a meta-compilation system that enables extensible compiler implementations through a combination of object-orientation and declarative computations. In particular, fixpoint computations are declaratively defined using circular attributes. A circular attribute is given by an initial value and an equation, which may reference the attribute itself. JastAdd iteratively applies this equation until the result stabilizes. Stability occurs when the equation is monotonic and all possible values of the attribute form a lattice of finite height, where every value has a well-defined notion of equality. In OODL, fixpoint computations are implicitly defined and computed by evaluating the corresponding Datalog program. Our mono types serve a similar purpose to circular attributes but operate differently. Mono types are first-class values that can be

passed around and manipulated across various locations. Their use does not mandate a fixpoint computation unless observations of mono types influence subsequent mono type inputs. Mono types do not need to implement a lattice and the partial order does not need to guarantee finite height, as long as the sequence of observed values satisfies the ascending chain condition.

Functional IncA [Pacak and Erdweg 2022] is a functional frontend for Datalog with first-class functions, algebraic data types, and sets. While we focus on OOP features, dynamic allocation, dynamic dispatch, and mutation, we import some ideas from this work. In particular, we borrow their encoding of control flow using the demand transformation and their encoding of sets. They translate sets to first-order Datalog relations, but use a defunctionalization transformation to also support first-class sets. OODL's case classes are also somewhat similar to algebraic data in functional IncA, but encoded differently. Other than that, the two languages have few similarities.

Flix [Madsen and Lhoták 2020; Madsen et al. 2016] is a functional metaprogramming language for Datalog with support for user-defined lattices to solve fixpoint problems. In Flix, Datalog programs are first-class values, meaning sets of rules can be instantiated, combined, and solved at run-time. This allows for interesting reuse patterns for Datalog programs. In contrast, we rely on object-oriented features to enable code reuse. In particular, OODL enables expressing fixpoint computations by using recursive methods and sets, using (relaxed) mutation to realize OOP-style programming. Flix also supports user-defined lattices, which are a special case of our mono types.

Kuper and Newton [2013] propose the concept of LVars (lattice vars), which are an abstraction for mutable variables intended for deterministic concurrent programming. LVars have join-on-write semantics and served as inspiration for the encoding of lattices as mono types. However, to guarantee deterministic read behaviour, LVars implement a so called threshold read. A threshold read blocks and thereby synchronizes all threads until the LVar crosses a certain threshold value. Since OODL is not designed with parallelism in mind, mono types serve a different purpose in our design, namely to enable mutation inside fixpoint computations. Therefore, we do not need to guarantee deterministic reads, but can rely on the eventual consistency of the fixpoint computation.

Datalog$^\circ$ is a datalog-like language enabling recursive computations over general semirings [Abo Khamis et al. 2022]. It operates on a partially ordered, pre-semiring (POPS) where both ring operations are monotone. To compute the least fixpoint of a Datalog$^\circ$ program, they adapt the semi-naïve evaluation algorithm for a general POPS. This means all relations operate on the same pre-semiring. Decoupling the partial order from the semiring structure allows support for semirings that are not naturally ordered. In contrast, mono types are first-class containers, each implementing distinct behavior. Mono types can be passed around in a Datalog program, exposing different behaviors to different relations. Each bounded semi-lattice gives rise to a mono type that can be expressed through lattice-based, recursive aggregation in a Datalog system. Thus, mono types inherit their fixpoint guarantees from the underlying aggregation algorithms, such as DRed_L [Szabó et al. 2018] or LADDDER [Szabó et al. 2021]. We also decouple the partial order from the underlying data type, with the user defining the partial order by adhering to the mono interface.

The frontend language of the Soufflé Datalog system [Jordan et al. 2016] provides a component system on top of Datalog to promote code reuse. A component can contain relation or type declarations, rules, facts, and nested components. While components can be instantiated, instantiation is static and Soufflé creates distinct namespaces for each instantiation at compile time.

9 Conclusion

We propose OODL, an object-oriented language that deeply integrates OOP and Datalog. In particular, OODL allows object-oriented programs to express fixpoint computations. We carefully translate OODL programs to Datalog in a way that preserves key OOP features: dynamic allocation, object identity, dynamic dispatch, and mutation. However, the side effects of allocation and mutation

conflict with Datalog's fixpoint semantics. To this end, we design extensions of OODL that permit object creation and mutation in a limited form, namely through algebraic case classes and our novel abstraction of *mono types*. We have implemented OODL and shown in case studies how it allows developers to combine OOP design patterns with fixpoint computations.

Acknowledgments

We thank the anonymous reviewers for their effort and helpful suggestions. This work has been funded by the German Research Foundation (DFG) (Project numbers 451545561 & 508316729) and the European Research Council (ERC) under the European Union's Horizon 2023 (Grant Agreement ID 101125325).

References

Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. 2005. Diagnosis of asynchronous discrete event systems: datalog to the rescue!. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, Chen Li (Ed.). ACM, 358–367. https://doi.org/10.1145/1065167.1065214

Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. 1993. Methods and Rules. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. ACM Press, 32–41. https://doi.org/10.1145/170035.170044

Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (Pre-) Semirings. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Philadelphia, PA, USA) *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/3517804.3524140

Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, Christine Morin and Gilles Muller (Eds.). ACM, 223–236. https://doi.org/10.1145/1755913.1755937

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011a. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011b. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

Michael Arntzenius and Neel Krishnaswami. 2020. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.* 4, POPL (2020), 22:1–22:28. https://doi.org/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, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 214–227. https://doi.org/10.1145/2951913.2951948

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

Catriel Beeri and Raghu Ramakrishnan. 1991. On the power of magic. *The Journal of Logic Programming* 10, 3 (1991), 255–299. https://doi.org/10.1016/0743-1066(91)90038-Q Special Issue: Database Logic Progamming.

Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 141:1–141:31. https://doi.org/10.1145/3428209

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25.

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends Databases* 5, 2 (2013), 105–195. https://doi.org/10.1561/1900000017

Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 1213–1216. https://doi.org/10.1145/1989323.1989456

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. 2023. Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. *PACMPL* 7, ICFP (2023).

Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-Based Data Structures for Deterministic Parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

Magnus Madsen and Ondrej Lhoták. 2020. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 125:1–125:28. https://doi.org/10.1145/3428193

Magnus Madsen, Ming-Ho Yee, and Ondrej 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, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

Eva Magnusson, Torbjorn Ekman, and Gorel Hedin. 2007. Extending Attribute Grammars with Collection Attributes–Evaluation and Applications. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 69–80. https://doi.org/10.1109/SCAM.2007.13

David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. 2018. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM / Morgan & Claypool, 3–100. https://doi.org/10.1145/3191315.3191317

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 222)*. Schloss Dagstuhl, 7:1–7:28.

André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

Max Schäfer, Pavel Avgustinov, and Oege de Moor. 2017. *Algebraic datatypes.* https://codeql.github.com/docs/ql-language-reference/types/#algebraic-datatypes Accessed: 2024-06-26.

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Programming Language Design and Implementation (PLDI)*. ACM.

K. Tuncay Tekle and Yanhong A. Liu. 2010. Precise Complexity Analysis for Efficient Datalog Queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (Hagenberg, Austria) *(PPDP '10)*. Association for Computing Machinery, New York, NY, USA, 35–44. https://doi.org/10.1145/1836089.1836094

K. Tuncay Tekle and Yanhong A. Liu. 2019. Extended Magic for Negation: Efficient Demand-Driven Evaluation of Stratified Datalog with Precise Complexity Guarantees. In *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019 (EPTCS, Vol. 306)*, Bart Bogaerts, Esra Erdem, Paul Fodor, Andrea Formisano, Giovambattista Ianni, Daniela Inclezan, Germán Vidal, Alicia Villanueva, Marina De Vos, and Fangkai Yang (Eds.). 241–254. https://doi.org/10.4204/EPTCS.306.28

Received 2024-04-03; accepted 2024-08-18