A Typed Multi-level Datalog IR and Its Compiler Framework

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

The resurgence of Datalog in the last two decades has led to a multitude of new Datalog systems. These systems explore novel ideas for improving Datalog's programmability and performance, making important contributions to the field. Unfortunately, the individual systems progress at a much slower pace than the overall field, because improvements in one system are rarely ported to other systems. The reason for this rift is that each system provides its own Datalog dialect with specific notation, language features, and invariants, enabling specific optimization and execution strategies.

This paper presents the first compiler framework for Datalog that can be used to support any Datalog frontend language and to target any Datalog backend. The centerpiece of our framework is a novel typed multi-level Datalog IR that supports IR extensions and guarantees executability. Existing Datalog systems can provide a compiler frontend that translates their Datalog dialect to the extended IR. The IR is then progressively lowered toward core Datalog, allowing optimizations at each level. At last, compiler backends can target different Datalog solvers. We have implemented the compiler framework and integrated 4 Datalog frontends and 3 Datalog backends, using 16 IR extensions. We also formalize the IR's flexible type system, which is bidirectional, flow-sensitive, bipolar, and uses three-valued typing contexts. The type system simultaneously validates type compatibility and precisely tracks bindings of logic variables while permitting IR extensions.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; **Type theory**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: Datalog, type system, multi-level IR, compiler framework

ACM Reference Format:
David Klopp, Sebastian Erdweg, and André Pacak. 2024. A Typed Multi-level Datalog IR and Its Compiler Framework. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 327 (October 2024), 29 pages. https://doi.org/10.1145/3689767

1 Introduction

Datalog was invented in 1977, five years after C and Prolog, three years after SQL, and two years after Scheme [Gallaire and Minker 1978]. However, only in the last decade or so, Datalog has been recognized as a *programming language* rather than an expressive query language for databases. While early Datalog programs implemented relatively simple recursive database queries, Datalog programs nowadays span many thousand lines of code and, for example, are used to implement state-of-the-art program analyses [Bravenboer and Smaragdakis 2009; Hajiyev et al. 2006]. This revival of Datalog as a programming language has led to a wide array of novel Datalog systems that improve Datalog's programmability and its performance.

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-ART327
https://doi.org/10.1145/3689767

The expressiveness and programmability of textbook Datalog is limited, since Datalog amounts to relational algebra with recursion. Language extensions such as negation, arithmetics, and aggregation have long been standard in Datalog systems, but recent research explores many new ideas. For example, Soufflé [Scholz et al. 2016] extends Datalog with algebraic data types and records, Formulog [Bembenek et al. 2020] extends Datalog with functions and SMT constraints, IncA [Pacak et al. 2022; Szabó et al. 2016] extends Datalog with pattern functions and structural EDB relations, Dedalus [Alvaro et al. 2010] extends Datalog with stateful relations, and egglog [Zhang et al. 2023] extends Datalog with equality saturation. Other systems forego Datalog and provide separate languages that execute like Datalog, such as the object-oriented QL [Avgustinov et al. 2016] and the functional Datafun [Arntzenius and Krishnaswami 2016]. At the same time, recent research on Datalog execution has made significant progress. For example, there is automatic index selection [Subotic et al. 2018] and feedback-directed join optimization [Arch et al. 2022] in Soufflé, differential data flow [McSherry et al. 2013] in DDLog [Ryzhyk and Budiu 2019], incremental recursive aggregation in IncA [Szabó et al. 2018, 2021], macro-based lowering to Rust in Ascent [Sahebolamri et al. 2022], and many domain-specific optimizations in the literature.

While these advances are promising, progress on Datalog is inhibited by the lack of a common language infrastructure. Specifically, each of the systems mentioned above comes with its own Datalog dialect with specific notation, language features, and invariants, enabling specific optimization and execution strategies. For example, a Soufflé program cannot be run with Formulog, a Formulog program cannot be run with IncA, an IncA program cannot be run with Dedalus, and so on. This has many downsides, which we summarize below:

(1) There is no portability between systems because language extensions in one Datalog dialect cannot be used in other Datalog systems.
(2) Optimizations implemented in one Datalog system cannot be exploited in another Datalog system. The effect of optimizations can also not be easily compared between systems.
(3) Datalog modules written for different systems cannot be linked and thus cannot interoperate.

In this paper, we present the first compiler framework for Datalog, outlined in Figure 1. Our framework adopts the architecture of LLVM's MLIR [Lattner et al. 2021] to Datalog: Datalog dialects can provide compiler frontends that translate their code into an intermediate representation (IR). Analysis and optimization passes can then operate on the IR independent of the input Datalog dialect. And finally, compiler backends can translate the optimized IR to emit code for different Datalog engines. This architecture readily solves the latter two issues discussed above: optimizations operate on the shared IR and are not specific to any Datalog system, and Datalog programs can be linked after translation into the IR. But, how can we address the wide variety of language features supported (and required) by existing Datalog dialects?

The key contribution of this work is the design of a typed multi-level IR for Datalog. A multi-level IR consists of a core IR and any number of IR extensions. Datalog dialects that require language features outside the realm of standard Datalog can add (shared) IR extensions and then target the extended IR. IR extensions generally implement *lowerings* that desugar the extended IR nodes. IR extensions can also be stacked on top of each other, such that *progressive lowering* eventually produces core IR code. Optimizations can act on any level of abstraction rather than on core IR only. And not all IR extensions have to be lowered: compiler backends may provide built-in support for any number of IR extensions.

The most important conceptual contribution of this work is the design of a novel type system for Datalog. We want the Datalog IR to be typed so that (i) optimizations can soundly assume valid IR as input and (ii) bugs in generated code are detected before subsequent compiler passes transform the code. However, existing type systems for Datalog do not guarantee type safety because they

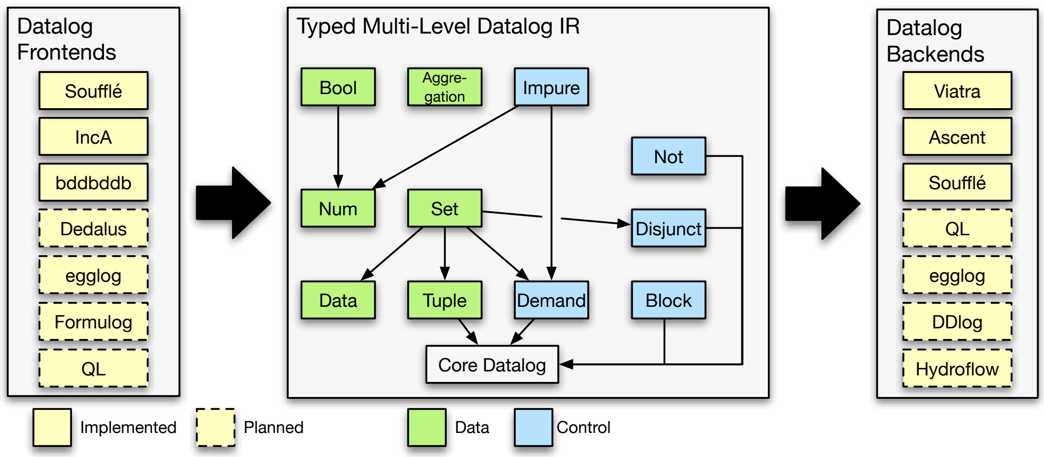

Fig. 1. Compiler framework for Datalog using a typed multi-level IR that permits optimizations at each level.

neglect variable bindings, or are overly restrictive and preclude IR extensions [Bembenek et al. 2020; Zook et al. 2009]. For example, the following rule aims to produce pairs (X, Y), but Y is not bound in the rule's body, because A is not bound, because the second call is negative.

$$\text{broken\_path}(X, Y) :- \text{edge}(X, Z), \neg\text{edge}(X, A), Y = A.$$

The property violated here is known as *range restriction* in the Datalog literature, but there are other cases where variable resolution can lead to run-time failure. To ensure type safety, we design a bidirectional, bipolar, and flow-sensitive type system for Datalog IR, using three-valued typing contexts. The type system is extensible, so that IR extensions can provide their own static semantics, allowing us to validate IR code at all levels. We have implemented the compiler framework and developed 16 typed IR extensions whose static semantics has unprecedented precision. For example, one IR extension implements the magic set transformation, which has a substantial effect on variable bindings, all covered by our type system.

In summary, this paper makes the following contributions:

- We analyze challenges for generating Datalog and show how a multi-level IR helps (Section 2).
- We present a core Datalog IR and its extensible type system (Section 3).
- We develop IR extensions with complex lowerings and capture their static semantics (Section 4).
- We connect 4 Datalog frontends and 3 Datalog backends to our framework (Section 5.1).

2 Why We Need a Multi-level Datalog IR

Our goal is to design a compiler framework for Datalog that supports any Datalog dialect and can target any Datalog solver. In this section, we introduce a motivating example to highlight typical issues with generating Datalog code.

Our motivating example stems from the domain of program analysis, which is a frequent use case of Datalog. For example, Doop defines points-to analysis for JVM bytecode in Datalog [Bravenboer and Smaragdakis 2009]. Imagine we want to improve the precision of Doop by adding an abstract interpreter that predicts the sign of numbers to decide whether a conditional jump can occur or not. Doop is implemented in Soufflé and it certainly should be possible to implement an abstract interpreter in Soufflé, too. However, since the abstract interpreter is a recursive function, we would much rather implement it in a functional Datalog dialect, compile it to the Datalog IR, and link it with the Doop points-to analysis written in Soufflé.

```
data Sign = Top | Pos | Neg | Zero
data AbsVal = Bottom | Num(Sign)
def absEval(e: Exp): AbsVal = e match
  case AddExp(e1, e2) => (absEval(e1), absEval(e2)) match
    case (Num(s1), Num(s2)) => Num(add(s1, s2))
    case _ => Bottom
  case NumExp(i) =>
    if (i < 0) Num(Neg)
    else if (i > 0) Num(Pos)
    else Num(Zero)
def add(v1: Sign, v2: Sign): Sign = (v1, v2) match
  case (Top, _) | (_, Top) | (Neg, Pos) | (Pos, Neg) => Top
  case (_, Zero) => v1
  case (Zero, _) => v2
  case (Neg, Neg) => Neg
  case (Pos, Pos) => Pos
```

Fig. 2. Sign analysis written in a functional language. How can we compile this code to Datalog?

```
absEval(e: Exp, s: AbsVal)
absEval(e, s) :-
  input(e), ?AddExp(e, e1, e2),      // Match and destruct Add expressions
  absEval(e1, v1), absEval(e2, v2),  // Evaluate both arguments
  ?Num(v1, s1), ?Num(v2, s2),        // Match and destruct evaluation results
  add(s1, s2, a), s = !Num(a).       // Compute and return abstract addition result
absEval(e, s) :-
  input(e), ?AddExp(e, e1, e2),
  absEval(e1, v1), absEval(e2, v2),
  ?Bottom(v1),                  // The first evaluation result is not a Num
  s = !Bottom.                  // Error state
absEval(e, s) :-
  input(e), ?AddExp(e, e1, e2),
  absEval(e1, v1), absEval(e2, v2),
  ?Num(v1, _), ?Bottom(v2),     // The first result is a Num, but the second result is not
  s = !Bottom.                  // Error state
absEval(e, s) :-
  input(e), ?NumExp(e, i),      // Match and destruct Num expressions
  i < 0,                        // Only proceed if the integer literal is smaller than 0
  sign = !Neg, s = !Num(sign)   // Return an abstract Num result with Neg sign
absEval(e, s) :-
  input(e), ?NumExp(e, i),
  i >= 0,                       // Mutually exclusive rules, by negating the first condition
  i > 0,                        // Check the second condition
  sign = !Pos, s = !Num(sign).  // Return an abstract Num result with Pos sign
absEval(e, s) :-
  input(e), ?NumExp(e, i),
  i >= 0,                       // Negate the first condition
  i <= 0,                       // Negate the second condition
  sign = !Zero, s = !Num(sign). // Return an abstract Num result with Zero sign
```

Fig. 3. Result of compiling function `absEval` from Figure 2 to Datalog with ADTs.

To keep our example tractable, we use a simple expression language here rather than bytecode:

$$\text{(programs)} \quad p ::= \overline{e} \qquad\qquad \text{(expressions)} \quad e ::= n \mid e + e$$

Figure 2 shows an abstract interpreter for this expression language. We assume that the expressions of our analysed language are encoded as an algebraic data type (ADT). To analyse an expression, we recursively call `absEval` to produce an abstract value: `Bottom` if the computation fails or a number `Num(s)` with its sign `s`. To analyze an `AddExp`, we abstractly evaluate both arguments and perform abstract addition of sign lattice values using the `add` function. We determine the sign of a numeric literal with an if expression consisting of three branches.

2.1 Why Generating Datalog Is Difficult

We need a multi-level Datalog IR because generating Datalog is difficult. To illustrate why, let us compile this abstract interpreter to Datalog. We encode functions as relations and use a standard Datalog extension for ADTs as supported by various Datalog engines, such as Soufflé [Scholz et al. 2015] and Viatra [Varró et al. 2016]. This compilation approach loosely follows the strategies described by [Pacak and Erdweg 2022]. We show the generated Datalog in Figure 3, but only for `absEval` because the Datalog code is bloated. Let us step through the translation process to identify the challenges involved. We start by compiling the outermost pattern match in `absEval` to Datalog and already encounter the first challenge:

Structural Mismatch. Most programming languages feature structured programming: nested control constructs to organize the program. For example, the abstract interpreter uses pattern matching and conditionals. However, Datalog consists of an entirely flat structure: relations consist of alternative rules, which consist of conjunctive atoms. This rigid structure makes Datalog an inconvenient compilation target. For example, Pacak and Erdweg [2022] present a compiler from a functional expression language to Datalog that has this signature:

$$\begin{aligned}&\mathsf{compile} : \mathit{exp} \longrightarrow \mathcal{P}(\mathit{term} \times \mathcal{P}(\mathit{atom}))\\&\mathsf{compile}(f(e_1, \ldots, e_n)) =\\&\quad \{(y, \{f(t_1, \ldots, t_n, y)\} \cup a_1 \cup \ldots \cup a_n) \mid (t_1, a_1) \in \mathsf{compile}(e_1), \ldots, (t_n, a_n) \in \mathsf{compile}(e_n)\}\end{aligned}$$

An expression is translated to a set of alternatives, each producing one value (encoded as a term) guarded by a set of atoms. We show their translation of function applications, which is fairly complex. Since each argument expression compiles to a set of alternatives, the compiler has to compute the Cartesian product of all alternatives in a set comprehension, and for each alternative, it has to merge all atoms from the compiled arguments. This is a lot of accidental complexity. Ideally, we would like the compiler to be much simpler:

$$\begin{aligned}&\mathsf{compile} : \mathit{exp} \longrightarrow \mathit{term}\\&\mathsf{compile}(f(e_1, \ldots, e_n)) = f(\mathsf{compile}(t_1), \ldots, \mathsf{compile}(t_n), y); y\end{aligned}$$

This is incompatible with Datalog's flat syntactic structure and the generated code shown in Figure 3. In particular, nested pattern matching and conditionals cause problems, because they need to be flattened into what amounts to disjunctive normal form. But that is not enough.

Control Mismatch. Most programming languages have control flow in the sense that the order in which instructions are executed matters. This is not the case for Datalog. Conceptually, in Datalog, all alternative rules of a relation are executed simultaneously and repeatedly until reaching a fixpoint. This makes targeting Datalog difficult. Consider again our example from Figure 2. The abstract interpreter `absEval` is order-dependent in three places.

First, the default pattern for the nested pattern match may only be considered after all other alternatives failed. In Figure 3, we can see that the first generated Datalog rule covers the successful case where `e1` and `e2` both evaluate to numbers. To encode the default pattern correctly, we must generate rules that check that no other pattern matches. The generated code does that by generating one rule for each combination of `(v1,v2)` that does not match `(Num(_),Num(_))`. Computing the correct set of rules quickly becomes complex. We would much rather compile the default pattern to a single rule guarded by `Not(?Num(v1,_),?Num(v2,_))`, which uses a generic negation IR extension.

Second, and similar to pattern matching, the branches of the conditional expressions may only be considered after evaluating all relevant conditions. In particular, the negated condition must be asserted in the else branch, and nested conditionals must evaluate all relevant conditions. To make sure the original execution order is correctly reflected in the generated code, the generated rules must be mutually exclusive so that there is no non-determinism. Rather than requiring each compiler to ensure this property, we would prefer a well-tested IR extension for Boolean expressions and conditionals that can be shared by different compilers.

Third, we cannot enumerate the result of `absEval` for all $e \in exp$ and should only consider expressions provided as arguments to `absEval` in some main function. Indeed, to produce valid Datalog code, the compiler must generate a relation `input` that enumerates all actual inputs of `absEval`. In Figure 3, we include a call to `input(e)` at the beginning of each rule to ensure `e` only ranges over the actual inputs. The `input` relation can be generated by a demand transformation [Tekle and Liu 2010] that each compiler must implement. Instead, we would like to provide a single IR extension that realizes this feature and that can be reused across compiler frontends.

Atoms Are Second-Class. Unlike terms, atoms do not produce a value and generally are second-class citizens in Datalog. Consequently, we cannot store the result of executing an atom, we cannot provide an atom as an argument in a call, and we cannot abstract over atoms. For example, in the generated code in Figure 3, numeric comparisons such as `i < 0` are encoded as atoms, and negation requires dedicated rewritings (e.g., `i < 0` becomes `i >= 0`) rather than negating the resulting Boolean value. Consider a slight extension of `absEval`, where we want to track a number's sign and parity:

```
val ev = even(i)
if (i < 0 && ev) Num(NegEven)
else if (i < 0) Num(NegOdd)
else if (i > 0 && ev) Num(PosEven)
else if (i > 0) Num(PosOdd)
else Num(Zero)
```

This has introduced various issues. First, negating the first condition yields a disjunction `i>=0 || !ev`, which means all subsequent rules will be duplicated after flattening because of the structural mismatch. Second, if Booleans are atoms without values, what should we store in variable `ev`? Third, it will be difficult to support user-defined Boolean connectives, because they cannot actually take or produce Boolean values.

Of course, we can also encode Booleans as terms rather than atoms, so that Boolean values have a run-time representation. The downside of this approach is the run-time overhead it induces. We believe a multi-level IR can support Boolean values while mitigating most overheads due to optimizations at different abstraction levels.

Logic Variables and Their Bindings. Datalog is a logic programming language and, as such, features logic variables. Logic variables are dynamically scoped and implicitly bound within a rule, which makes their binding difficult to reason about. For example, in the generated Datalog code of Figure 3, `input(e)` binds `e` so that `?AddExp(e, e1, e2)` subsequently binds `e1` and `e2`. But if we negate the pattern match `not ?AddExp(e, e1, e2)`, `e1` and `e2` remain unbound; we are only testing whether `e` is not an `AddExp`. That is, sometimes a term is allowed to bind variables, but other times it is only well-typed when variables have been previously bound. Compilers have to know and adhere to these binding rules when generating Datalog. This is difficult currently because type systems for Datalog either ignore variable bindings, which is unsafe, or they are overly restrictive, which precludes IR extensions.

The challenges described so far render the generation of correct Datalog code complex and error-prone. How can we streamline and simplify the code generation process?

2.2 How a Multi-level IR Helps Generating Datalog

Rather than generating Datalog code directly, we propose to treat Datalog as a core IR in a multi-level IR setup. This allows us to introduce IR extensions that make generating Datalog easier. For example, we can translate the `absEval` function to the following extended Datalog IR:

```
absEval(e: Exp@demanded, s: AbsVal)
absEval(e, s) :- e match
  case ?AddExp(e, e1, e2) => absEval(e1, n1),  absEval(e2, n2), (n1, n2) match
    case (?Num(s1), ?Num(s2)) => add(s1, s2, a), s = !Num(a).
    case _ => s == !Bottom
  case ?NumExp(e, i) =>
      { (i < 0) = true, s = !Num(!Neg) }
    ∨ { (!(i < 0) && i > 0) = true, s = !Num(!Pos) }
    ∨ { (!(i < 0) && !(i > 0)) = true, s = !Num(!Zero) }
```

The type signature of `absEval` declares `e` to be demanded, meaning it is considered an input to the relation definition. The type annotation `@demanded` stems from an IR extension. We then use pattern matching (another IR extension) on algebraic data types (also an IR extension). Instead of flattening the nested conditionals, we use a disjunction (an IR extension) over blocks (also an IR extension) and encode the conditions as Boolean terms (another IR extension).

Importantly, it is not necessary to support all of these IR extensions in compiler backends in our framework. Instead, our multi-level IR progressively lowers IR extensions to simpler forms until we are left with core Datalog features. For example, pattern matching lowers to IR code that uses disjunctions and negation, but that does not need pattern matching anymore. We can optimize the IR code *before each* lowering as to implement abstraction-specific optimizations. In the end, compiler backends only need to support core Datalog features and primitive values.

3 A Typed IR for Core Datalog

The centerpiece of our compiler framework is a typed and extensible Datalog IR. In this section, we introduce the core IR and its type system. The core Datalog IR resembles textbook Datalog with negation: a program consists of rules, which define a head $p(\overline{X})$ and body $\overline{a}$ consisting of atoms. Each atom is either a call $p(\overline{t})$ that queries a relation p or an equation $t_1 = t_2$.

$$\begin{array}{llll@{\qquad}llll}
\text{(programs)} & D & ::= & \overline{sig}\ \overline{r} & \text{(atoms)} & a & ::= & p^{\#}(\overline{t}) \mid t =^{\#} t \\
\text{(signatures)} & sig & ::= & p : T \times \cdots \times T & \text{(terms)} & t & ::= & X \\
\text{(rules)} & r & ::= & p(\overline{X}) :- \overline{a}. & \text{(polarity)} & \# & ::= & + \mid - \\
\end{array}$$

$$X \in \mathbf{Var} \qquad T \in \mathbf{Type} \qquad v \in \mathbf{Val}$$

Core Datalog differs from textbook Datalog in two ways. First, textbook Datalog usually permits constants as terms, but there is only one kind of term in core Datalog: variables. That is, core Datalog programs cannot create values v; all values must be drawn from the predefined fact base (i.e., the EDB). This is because we allow (and expect) IR extensions to introduce value domains and the corresponding literal terms (cf. value extensions in Figure 1). For the time being, we treat values v and their types T as opaque placeholders.

The second difference from textbook Datalog is that we mark the polarity of calls and equations explicitly in core Datalog. In particular, $p^{-}(\bar{t})$ marks a negative call and $t_1 =^{-} t_2$ denotes an inequality constraint. We write $t_1 =^{\#} t_2$ with $\# \in \{+, -\}$ to abstract from a specific polarity. Calls and equations without explicit polarity annotation are assumed to be positive.

3.1 Type Safety and Evaluation Order

Well-typed programs may not go wrong. In Datalog, programs can go wrong in two ways. First, a term may produce a value whose type is incompatible in the term's context. For example, after extending the IR with integer addition, we need to ensure that its operands produce integer values. But even in the core IR, rules must produce tuples that are compatible to the relation's signature. We will ensure type compatibility with minimal type annotations using local type inference and, in particular, a bidirectional type system.

The second kind of run-time error is seemingly simpler, but actually more difficult to handle: variable resolution. In Datalog, there are neither explicit variable declarations nor explicit assignments. Instead, variables are bound through predicate calls and equational constraints. For example, here are three equivalent variants of the recursive path rule:

$$\begin{array}{lcl}
\mathsf{path}(A, B) & :- & \mathsf{edge}(A^{\circ}, C^{\circ}), \mathsf{path}(C^{\bullet}, B^{\circ}). \\
\mathsf{path}(A, B) & :- & \mathsf{edge}(A^{\circ}, C^{\circ}), \mathsf{path}(D^{\circ}, B^{\circ}), C^{\bullet} = D^{\bullet}. \\
\mathsf{path}(A, B) & :- & \mathsf{edge}(A^{\circ}, C^{\circ}), C^{\bullet} = D^{\circ}, \mathsf{path}(D^{\bullet}, B^{\circ}).
\end{array}$$

We have decorated the rule definitions with binding information. A variable $X^{\bullet}$ is a bound occurrence of a variable, that is, the variable has previously been assigned a value, which is retrieved at this occurrence. In contrast, a variable X° is a binding occurrence of a variable, that is, we are assigning the variable its value(s) at the present occurrence.[1]

In the first rule, we can see that A and C are binding occurrences in the call to edge: We are reading their values from the edge relation. Subsequently, C is a bound occurrence in the call to path whereas B is a binding occurrence: the rule computes an equi-join between edge and path on C. We can equivalently express an equi-join using an equational constraint as the second and third rules illustrate. The second rule asserts $C = D$ at the very end, when both variables have already been bound. Instead, the third rule assigns C to D before querying path. Note that direction of the assignment $C = D$ is determined by the boundedness of C and D in the equation, not by their position: for any pair of terms, $t_1 = t_2$ and $t_2 = t_1$ have the exact same run-time behavior.

The binding of variables crucially depends on the order, in which the atoms from a rule's body are evaluated. This is known as sideways information passing in the Datalog literature. Some Datalog systems provide a non-deterministic semantics that leaves the evaluation order of atoms undefined, which entails non-deterministic binding information. While sensible for a declarative language, our Datalog IR is designed to be generated, not handwritten. As such, we expect compiler frontends and optimizations want to be as explicit about the generated code's semantics as possible. Therefore, we define that the **evaluation order of atoms in the Datalog IR is left-to-right**. All examples shown have used this evaluation order already.

During the evaluation of Datalog rules, there are certain points where variables must be bound occurrences. We call these points **strictness points**. For example, the following four rules violate

[1]Datalog has set semantics, where all bound variables $(X_1, \ldots, X_n)$ together are assigned a set of tuples during run time.

different strictness points and would fail at run time:

$$\begin{array}{lll} \mathsf{yes}(A,B) & :- & B^\circ = \text{“yes”}. \\ \mathsf{sqrt4}(X) & :- & 4 = X^\circ * X^\circ. \\ \mathsf{path}(A,B) & :- & A^\circ = B^\circ. \\ \mathsf{unreachable}(A,B) & :- & \mathsf{path}^-(A^\circ, B^\circ). \end{array}$$

The first rule violates what is known as *range restrictedness*: Each head variable of a rule must be positively bound in its body. Using our binding annotations, this means each head variable V must be bound $V^\bullet$ at the end of the rule's body. This ensures Datalog can indeed enumerate all derivable tuples. Rule yes fails this condition because A° is unbound at the end of the body. The second rule uses the unbound X as an arithmetic operand. This fails because arithmetic operators can only be applied to values; they cannot be used to bind variables. The third rule fails because at least one side of an equation $A = B$ must be bound to a value. In the fourth rule, the negative call corresponds to a check $(A, B) \notin \mathsf{path}$, which is strict in (A, B) and cannot be used to bind variables.

Our goal is to design a type system for Datalog IR that prevents all of the above run-time errors.

Definition 1 (Type safety.). A Datalog program is type-safe if

(1) all terms have types compatible with the context in which they appear, and
(2) no unbound variables occur at strictness points.

Our type system concretizes this definition by stipulating when exactly a term has a compatible type and where strictness points are precisely. Most prior work focuses on refinement types for logic variables to enable optimizations, but does not guarantee type safety since it excludes precise variable resolution. Formulog [Bembenek et al. 2020] features a safe type system, but it imposes overly restrictive constraints on variable bindings that preclude many useful IR extensions (cf. the detailed discussion in Section 6).

3.2 Typing the Core Datalog IR

We present the first extensible type system for Datalog that ensures type safety. Ensuring type safety for Datalog and, in particular, resolving the boundedness of variables correctly requires a combination of advanced type system features:

Bidirectional. A bidirectional type system checks terms against an expected type whenever possible, but infers a term's type when insufficient contextual information is available. While we use type inference for terms in equations $t_1 = t_2$, we use type checking for calls like $\mathsf{path}(A, C)$ with an unknown variable C, whose type is drawn from the signature of path.

Flow-sensitive. In languages with lexical scoping, the typing context is passed top-down in environment-passing style. However, Datalog's logic variables are neither lexically declared nor bound. Therefore, we define a flow-sensitive type system that passes the typing context in store-passing style to communicate new variable bindings from one atom to the next.

Bipolar. Type-safe Datalog requires strictness points where no binding may take place. To this end, we formulate the type system in a bipolar style: Each typing judgment has a polarity $\# \in \{+, -\}$ that determines whether bindings are allowed + or disallowed −.

The type system uses the following four judgments:

$$\begin{array}{llll} \text{(term inference)} & \Gamma \vdash^\# t \Rightarrow T \dashv \Gamma & \text{(atom checking)} & \Gamma \vdash^\# a \text{ ok} \dashv \Gamma \\ \text{(term checking)} & \Gamma \vdash^\# t \Leftarrow T \dashv \Gamma & \text{(rule checking)} & \vdash p(\overline{X}) :- \overline{a}. \checkmark \end{array}$$

The type of terms can be inferred $t \Rightarrow T$ or checked $t \Leftarrow T$ as known from bidirectional type systems. Since variable bindings in terms and atoms are flow-sensitive, the typing context is threaded

$$\frac{\Gamma(X) = T^\bullet}{\Gamma \vdash^{\#} X \Leftarrow T \dashv \Gamma}\ \text{C-Var}^\bullet \qquad \frac{\Gamma(X) = T^\circ}{\Gamma \vdash^{+} X \Leftarrow T \dashv \Gamma; X{:}T^\bullet}\ \text{C-Var}^\circ \qquad \frac{X \notin \Gamma}{\Gamma \vdash^{+} X \Leftarrow T \dashv \Gamma; X{:}T^\bullet}\ \text{C-Var}^{\notin}$$

$$\frac{\Gamma(X) = T^\bullet}{\Gamma \vdash^{\#} X \Rightarrow T \dashv \Gamma}\ \text{I-Var}^\bullet \qquad \frac{\Gamma(X) = T^\circ}{\Gamma \vdash^{+} X \Rightarrow T \dashv \Gamma; X{:}T^\bullet}\ \text{I-Var}^\circ$$

$$\frac{p : T_1 \times \cdots \times T_n \qquad \forall i.\ \Gamma_i \vdash^{\#_1 \cdot \#_2} t_i \Leftarrow T_i \dashv \Gamma_{i+1}}{\Gamma_1 \vdash^{\#_1} p^{\#_2}(t_1, \ldots, t_n)\ \text{ok} \dashv \Gamma_{n+1}}\ \text{A-Call}$$

$$\frac{\Gamma_1 \vdash^{-} t_1 \Rightarrow T \dashv \Gamma_2 \qquad \Gamma_2 \vdash^{\#_1 \cdot \#_2} t_2 \Leftarrow T \dashv \Gamma_3}{\Gamma_1 \vdash^{\#_1} t_1 =^{\#_2} t_2\ \text{ok} \dashv \Gamma_3}\ \text{A-EqL}$$

$$\frac{\Gamma_1 \vdash^{-} t_2 \Rightarrow T \dashv \Gamma_2 \qquad \Gamma_2 \vdash^{\#_1 \cdot \#_2} t_1 \Leftarrow T \dashv \Gamma_3}{\Gamma_1 \vdash^{\#_1} t_1 =^{\#_2} t_2\ \text{ok} \dashv \Gamma_3}\ \text{A-EqR}$$

$$\frac{p : T_1 \times \cdots \times T_n \qquad \Gamma_1 = X_1{:}T_1^\circ; \ldots; X_n{:}T_n^\circ \qquad \forall i.\ \Gamma_i \vdash^{+} a_i\ \text{ok} \dashv \Gamma_{i+1} \qquad \forall j.\ \Gamma_{r+1} \vdash^{-} X_j \Leftarrow T_j \dashv \_}{\vdash\ p(X_1, \ldots, X_n) :- a_1, \ldots, a_r.\ \checkmark}\ \text{Rule}$$

Fig. 4. A bidirectional, flow-sensitive, and bipolar type system for core Datalog that ensures type safety.

as input and output in these judgments. Terms and atoms are checked under a given polarity #, allowing or disallowing new variable bindings. In particular, typing rules can declare strictness in a term or atom by setting its polarity to -, which prohibits new bindings. Note that in the core Datalog IR, atoms are only checked positively. However, some IR extensions introduce nested atoms, where it is necessary to specify the polarity explicitly.

The typing context must distinguish three states for variables. First, a variable can be undefined $X \notin \Gamma$, meaning we have no information about X. Second, a variable can be declared $\Gamma(X) = T^\circ$, meaning we know its type but no value has been assigned to it yet. And third, a variable can be bound $\Gamma(X) = T^\bullet$, meaning we know its type and may read its values. We define the syntax for typing contexts accordingly:

$$\text{(contexts)} \quad \Gamma ::= \varepsilon \mid \Gamma; X{:}T^\circ \mid \Gamma; X{:}T^\bullet$$

Figure 4 shows the typing rule for core Datalog. Most rules are concerned with variable handling as expected. There are $2 * 2 * 3 = 12$ configurations to be considered for checking variables because the type system is bidirectional, bipolar, and the typing context distinguishes three variable states. The typing rules only list valid cases; the tables below define validity for all combinations:

direction	+/-	$\Gamma(X)$	valid?	rule
check	+	●	yes	C-Var$^\bullet$
check	+	○	yes	C-Var$^\circ$
check	+	$\notin$	yes	C-Var$^{\notin}$
check	-	●	yes	C-Var$^\bullet$
check	-	○	no	
check	-	$\notin$	no	

direction	+/-	$\Gamma(X)$	valid?	rule
infer	+	●	yes	I-Var$^\bullet$
infer	+	○	yes	I-Var$^\circ$
infer	+	$\notin$	no	
infer	-	●	yes	I-Var$^\bullet$
infer	-	○	no	
infer	-	$\notin$	no	

Note that references to bound variables $\bullet$ are always valid. However, declared variables $\circ$ may only be referred to in binding contexts that have + polarity. Undefined (local) variables can occur at any time in Datalog, but are only valid in binding contexts where the expected type is known.

The typing rule for calls $p^{\#_2}(t_1, \dots, t_n)$ is largely standard: We check each argument term t_i against the corresponding parameter type T_i, with a threaded typing context. What is novel here is how the polarity of the argument check is computed, namely by combining the polarity of the call $\#_2$ with the polarity of the judgment $\#_1$. Polarities can be multiplied like arithmetic signs to determine whether a check is allowed to bind variables or not.

$$\#_1 \cdot \#_2 = \begin{cases} +, & \text{if } \#_1 = + \text{ and } \#_2 = + \\ -, & \text{if } \#_1 = + \text{ and } \#_2 = - \\ -, & \text{if } \#_1 = - \text{ and } \#_2 = + \\ +, & \text{if } \#_1 = - \text{ and } \#_2 = - \end{cases}$$

In a positive context, arguments of positive calls are positive (can be bound by the call), but arguments of negative calls are negative (must be bound prior to the call). We get the inverse in a negative context: arguments of positive calls are negative, but arguments of negative calls are positive. That is, terms are positive if they are wrapped in an even number of negations. Or conversely, calls are strict in their arguments when they are wrapped in an odd number of negations. For example, not (not $\mathsf{p}^-(X)$) checks X in a negative context, because there are three negations.

There are two symmetric rules for handling equations. An equation is valid if at least one of the terms only refers to bound variables (- polarity). Hence, equations are strict in one of their arguments, but its position is not predetermined. The type of the other term can be checked against the inferred type. Like for calls, we multiply polarities to determine whether an equation is positive (an equality) or negative (an inequality), and use the corresponding polarity for term checking.

Finally, we define the validity of Datalog rules. We construct an initial context for the rule's body that marks all head variables as declared $X_i : T_i^\circ$ but not bound. We then check all atoms left-to-right by threading the context. We have carefully set up the type system such that we can now check for range restrictedness: In the final typing context of a rule Γ_{n+1}, all head variables must be bound. That is, rules are strict in the tuples $(X_1, \dots, X_n)$ they produce.

3.3 Typing Examples

Consider we want to type check the second rule of the standard path example:

$$\mathsf{path}(A, B) :- \mathsf{edge}^+(A, C), \mathsf{path}^+(C, B).$$

The second rule has two head variables, which we initialize in the context as declared variables and use it to type the first atom of the rule:

$$\dfrac{\dfrac{}{A{:}T^\circ; B{:}T^\circ \vdash^+ A \Leftarrow T \dashv A{:}T^\bullet; B{:}T^\circ}\ \text{C-Var}^\circ \qquad \dfrac{}{A{:}T^\bullet; B{:}T^\circ \vdash^+ C \Leftarrow T \dashv A{:}T^\bullet; B{:}T^\circ; C{:}T^\bullet}\ \text{C-Var}^{\notin}}{A{:}T^\circ; B{:}T^\circ \vdash^+ \mathsf{edge}^+(A, C)\ \mathsf{ok} \dashv A{:}T^\bullet; B{:}T^\circ; C{:}T^\bullet}\ \text{A-Call}$$

This derivation tree validates the call to edge. Argument A is declared and becomes bound through the call. Argument C is undefined but also becomes bound through the call. Since the type system is flow-sensitive, the second atom is checked in an updated context resulting from the first atom:

$$\dfrac{\dfrac{}{\dots \vdash^+ C \Leftarrow T \dashv A{:}T^\bullet; B{:}T^\circ; C{:}T^\bullet}\ \text{C-Var}^\bullet \qquad \dfrac{}{\dots \vdash^+ C \Leftarrow T \dashv A{:}T^\bullet; B{:}T^\bullet; C{:}T^\bullet}\ \text{C-Var}^\circ}{A{:}T^\bullet; B{:}T^\circ; C{:}T^\bullet \vdash^+ \mathsf{path}^+(C, B)\ \mathsf{ok} \dashv A{:}T^\bullet; B{:}T^\bullet; C{:}T^\bullet}\ \text{A-Call}$$

The final context Γ satisfies the range restriction of Datalog rules: Each head variable is positively bound in the rule's body: both $\Gamma \vdash^- A \Leftarrow T \dashv \_$ and $\Gamma \vdash^- B \Leftarrow T \dashv \_$ succeed. Had the second atom call been $\mathsf{path}(C, A)$ instead, B would not be bound at the end of the rule and $\Gamma \vdash^- B \Leftarrow T \dashv \_$ would have failed.

Polarities specify strictness points of the language, that is, places where no bindings may occur. For example, consider again the unreachable rule from Section 3.1:

$$\mathsf{unreachable}(A, B) \ \mathrel{:-}\ \mathsf{path}^-(A^\circ, B^\circ).$$

Recall that this rule is *not* type-safe because evaluation gets stuck, when the arguments of negative calls are unbound. Indeed, we cannot construct a typing derivation for the negative call, even though A and B appear in the initial context as declared. Let $\Gamma = A{:}T^\circ; B{:}T^\circ$ be the initial context:

$$\dfrac{\dfrac{\lightning\Gamma \vdash^- A \Leftarrow T \dashv \Gamma\lightning \qquad \lightning\Gamma \vdash^- B \Leftarrow T \dashv \Gamma\lightning}{\Gamma \vdash^+ \mathsf{path}^-(A, B)\ \mathsf{ok} \dashv \Gamma}\ \text{A-Call} \qquad \lightning\Gamma \vdash^- A \Leftarrow T \dashv \_\lightning \qquad \lightning\Gamma \vdash^- B \Leftarrow T \dashv \_\lightning}{\vdash\ \mathsf{unreachable}(A, B)\ \mathrel{:-}\ \mathsf{path}^-(A^\circ, B^\circ).\ \checkmark}\ \text{Rule}$$

Overall, based on stuck sub-derivations, we obtain four typing errors for this example: both arguments of path^- are illegal in a negative context because there is no inference rule that resolves to a declared (but not bound) variable in a negative context, and both head variables A and B fail range restrictedness because they lack a positive binding. Our type system correctly rejects this and other unsafe Datalog programs that would fail at run time.

3.4 Type Soundness

We prove soundness for the type system of Core Datalog based on a recent structural operational semantics of Datalog [Pacak and Erdweg 2023]. This semantics models a top-down evaluation of Datalog, which is equivalent to the standard bottom-up evaluation, implemented in most Datalog systems. The top-down semantics is a good match for our type system because it makes bindings explicit in what is known as *supplementary tables* in Datalog literature [Green et al. 2013]. A supplementary table has the form $\mathsf{table}(\overline{X}, V)$ and contains a set of rows V that bind the free variables $\overline{X}$ of a rule. As evaluation progresses through a rule and binds more variables, the supplementary table is extended with additional columns using the *bind* operation:

$$\mathit{bind}(X, V, \sigma) = \sigma \bowtie \mathsf{table}(X, V)$$

We can use the *bind* to tightly relate our typing contexts to sets of value tables it governs:

$$[\![\Gamma]\!] = \begin{cases} \{\mathsf{table}(\varepsilon, ())\}, & \text{if } \Gamma = \varepsilon \\ [\![\Gamma']\!], & \text{if } \Gamma = \Gamma'; X{:}T^\circ \\ \{\mathit{bind}(X, V, \sigma) \mid V \subseteq [\![T]\!], V \neq \emptyset, \sigma \in [\![\Gamma']\!]\}, & \text{if } \Gamma = \Gamma'; X{:}T^\bullet \end{cases}$$

The empty context corresponds to the unit table with no columns and a single row. Variables, which are declared but not bound in the context, do not add bindings to a value table. Only bound variables have bindings in the value table and the value must be of the appropriate type $[\![T]\!]$. This semantic specification of contexts nicely defines how declared variables differ from bound variables. It is also easy to provide a syntactic relation that validates whether a given value table adheres to a context, which we elided here.

We can now formulate the standard theorems of syntactic type soundness for atoms:

Theorem 2 (Preservation). *If* $\Gamma_1 \vdash^\# a_1\ \mathsf{ok} \dashv \Gamma_2$ *and* $\sigma_1 \in [\![\Gamma_1]\!]$ *and* $\sigma_1 \vdash a_1 \ \longrightarrow^A\ a_2 \dashv \sigma_2$*, then* $\Gamma_1 \vdash^\# a_2\ \mathsf{ok} \dashv \Gamma_2$ *and* $\sigma_2 \in [\![\Gamma_2]\!]$.

Theorem 3 (Progress). *If* $\Gamma_1 \vdash^\# a_1\ \mathsf{ok} \dashv \Gamma_2$ *and* $\sigma_1 \in [\![\Gamma_1]\!]$ *and* a_1 *is not a value, then there is* a_2 *and* σ_2 *such that* $\sigma_1 \vdash a_1 \ \longrightarrow^A\ a_2 \dashv \sigma_2$.

There are a few complications in actually conducting these proofs using the semantics by Pacak and Erdweg [2023]. First, the reduction relation for atoms does not actually yield bindings σ_2. The updated bindings are computed in a surrounding relation for reducing rules, using a *merge* function. However, it is easy to refactor the reduction rules such that the *merge* happens in atom reduction already. Second, calls reduce to an intermediary subquery form, which recursively evaluates until reaching a fixed-point. Since we are following a syntactic typing discipline, we need to provide a typing rule for this intermediate form. The syntax of subqueries is fairly involved to allow fixed-point iterations, and its typing rule follows step. We present a somewhat simplified typing rule that covers the main usage of subqueries:

$$\frac{\begin{array}{ccc} p : X_1{:}T_1; \dots ; X_n{:}T_n & \sigma_a \in \Pi_?(\llbracket \Gamma \rrbracket) & \forall i.\ \vdash r_i \checkmark \\ \Gamma = X_1{:}T_1^\bullet; \dots ; X_n{:}T_n^\bullet & \sigma_r \in \llbracket \Gamma \rrbracket_\emptyset & \forall i.\ \Gamma_i \vdash^{\#_1 \cdot \#_2} t_i \Leftarrow T_i \dashv \Gamma_{i+1} \end{array}}{\Gamma_1 \vdash^{\#_1} \mathbf{sq}(p^{\#_2}(\bar{t}), \sigma_a, \sigma_r, \sigma, r_1 \vee \dots \vee r_n) \text{ ok} \dashv \Gamma_{n+1}} \text{ A-Subquery}$$

Given the schema of relation p (including column names), we define Γ to bind variables for all columns of p. The tables σ_a and σ_r must adhere to Γ. Here, σ_a comprises the given arguments of the subquery, which must be a sub-table of a table in $\llbracket \Gamma \rrbracket$ (i.e., a projection of a well-typed result table). On the other hand, σ_r collects the results of p and must be empty or contained in $\llbracket \Gamma \rrbracket$. The subquery will execute the rules $r_1 \vee \dots \vee r_n$, which must be well-typed. During execution, the leftmost rule may be partially evaluated already and has to be well-typed under the type of σ, a detail we elided for clarity in the typing rule above. The result Γ_{n+1} of a subquery is computed by iterating over the arguments t_i, which is why the original call is part of the subquery.

Proof: Type preservation for atoms. By induction on the typing relation.

- $\Gamma_1 \vdash^{\#_1} t_1 =^{\#_2} t_2 \text{ ok} \dashv \Gamma_2$: Equality atoms reduce in a single step yielding a filtered table or a table that binds one extra column. We proceed by inversion of the preconditions for t_1 and t_2 and confirm: When $\Gamma_1 = \Gamma_2$, the reduction semantics indeed only filters the value table. And when $\Gamma_2 = \Gamma_1; X{:}T^\bullet$, the reduction semantics indeed binds X in the value table.
- $\Gamma_1 \vdash^{\#_1} p^{\#_2}(t_1, \dots, t_n) \text{ ok} \dashv \Gamma_{n+1}$: Calls either reduce to an immediate result if the query was evaluated before or reduce to a subquery of p. In the first case, the result table has type Γ_{n+1} given that the fixed-point cache is well-typed, which we have to establish as separate invariant inductively. In the second case, it is easy to confirm that the generated subquery preserves the type of the call.
- $\Gamma_1 \vdash^{\#_1} \mathbf{sq}(p^{\#_2}(\bar{t}), \sigma_a, \sigma_r, \sigma, r_1 \vee \dots \vee r_n) \text{ ok} \dashv \Gamma_{n+1}$: There are four reduction rules for subqueries. Two rules are responsible for reducing the leftmost rule r_1 and adding its result to σ_r, which preserves the type of the subquery. One rule yields a value table when the subquery reached a fixed-point, which is equivalent to a call that yields immediately, as discussed above. Finally, one rule restarts a subquery when it finished but did not reach a fixed-point yet. The resulting subquery restarts with an empty σ_r and is otherwise equal to the original subquery. Thus, reduction of subqueries preserves types.

□

Proof: Progress for atoms. By induction on the reduction relation.

- Equalities are stuck if both terms are free variables. Case distinction reveals that this cannot happen for well-typed equations.
- Inequalities are stuck if either term is a free variable. Again, case distinction shows this cannot happen for any well-typed inequality.
- Calls are never stuck because they either yield an immediate result or spawn a subquery.

- For subqueries, we distinguish two cases. If there are no more rules to execute, the subquery result is either unstable (new tuples found) or stable. An unstable subquery is restarted, whereas a stable subquery reduces to a value table. If there are more rules to execute, we either have progress in the leftmost rule or the leftmost rule is a result table, which is merged with σ_r and then discarded from the rules. In all cases, we have progress.

□

We also need to prove preservation and progress for rule reduction, which reduces a rule's leftmost atom stepwise until it yields a value table and can be merged. Atoms are removed from a rule's body when their evaluation finishes, and this repeats until the rule's body is empty. To reason about the soundness of this reduction, we provide a typing rule for partially evaluated Datalog rules:

$$\frac{\forall i.\ \Gamma_i \vdash^{+} a_i \text{ ok} \dashv \Gamma_{i+1}}{\Gamma_1 \vdash\ p(X_1, \ldots, X_n) :- a_1, \ldots, a_r.\ \checkmark \dashv \Gamma_{n+1}}\ \text{Rule'}$$

Preservation and progress hold for rule reduction, because reducing the leftmost atom has progress and preserves types. Formally, the type soundness for atom reduction and rule reduction has to be verified simultaneously, since they are mutually dependent. In summary, we conclude that our type system for Core Datalog is sound with respect to the small-step reduction semantics. As such, Core Datalog can serve as the basis for type-safe IR extensions, where we have to ensure that lowerings preserve typing [Lorenzen and Erdweg 2013, 2016]. In this paper, we focus on the foundational design for a type multi-level Datalog IR and introduce IR extensions without soundness proofs.

4 Typed Extensions of Datalog IR

The core Datalog IR and its type system only support the most basic Datalog operations. When generating Datalog, it is useful to introduce higher-level IR features (i) to simplify the frontend compilers and (ii) to enable optimizations at different abstraction levels. In this section, we demonstrate how our Datalog compiler framework supports typed multi-level IR extensions. We consider IR extensions in two categories as highlighted in Figure 1: adding new kinds of data and adding control constructs. We have implemented 16 IR extensions so far: Here we focus on the mechanism that makes the IR extensible, presenting parts of selected extensions only. We invite the reader to study the other IR extensions in the open-source code repository.[2]

4.1 A First Example: IR Extension for Arithmetics

The core Datalog IR does not provide any values or operations to act on them. We define an IR extension for arithmetics that adds IR nodes for integer and floating-point constants as well as standard operations to compare and compute with numbers. We present the additional IR nodes as an extension of the IR syntax, only mentioning extended and new nonterminals:

$$\begin{array}{lll}
\text{(atoms)} & a & ::= \ldots \mid t\ \mathit{compareOp}\ t \\
\text{(terms)} & t & ::= \ldots \mid \mathit{int} \mid \mathit{dbl} \mid t\ \mathit{binOp}\ t \\
\text{(types)} & T & ::= \ldots \mid \mathsf{Int} \mid \mathsf{Double}
\end{array}$$

$$\mathit{compareOp} \in \{<, \leq, >, \geq\} \quad \mathit{binOp} \in \{+, -, *, /, \mathit{min}, \mathit{max}, \ldots\} \quad \mathit{int} \in \mathbf{Int} \quad \mathit{dbl} \in \mathbf{Double}$$

Datalog frontends that feature arithmetics can target the Datalog IR extended with these arithmetic terms and atoms. The frontend does not need to know how arithmetics is handled downstream, whether it is lowered to other IR nodes or supported by Datalog backends directly (it's the latter). However, frontends need to generate valid IR. Each IR extension must define typing rules to govern

[2]https://gitlab.rlp.net/plmz/inca-scala

how the extended IR nodes may be used. For arithmetics, this is rather straightforward; we'll show a few representative rules only:

$$\frac{}{\Gamma \ \vdash^{\#} \ \mathit{int} \Rightarrow \mathsf{Int} \ \dashv \ \Gamma}\ \text{I-Int} \qquad \frac{\Gamma_1 \vdash^{-} t_1 \Rightarrow T \dashv \Gamma_2 \qquad \Gamma_2 \vdash^{-} t_2 \Leftarrow T \dashv \Gamma_3 \qquad T \in \{\mathsf{Int}, \mathsf{Double}\}}{\Gamma_1 \ \vdash^{\#} \ t_1 \ \mathit{binOp} \ t_2 \Rightarrow T \ \dashv \ \Gamma_3}\ \text{I-BinOp}$$

The first rule states that an integer literal always has type Int. The second rule assigns type T to an addition if both operands have type T and T is a numeric type. Note that the rule addition also defines strictness points for its operands: Both operands are checked in a negative context $\vdash^{-}$, which asserts all variables occurrences in t_1 and t_2 are bound occurrences. The inference rule for comparisons and the checking rules are analogous.

The arithmetic IR does *not* lower to core Datalog nor to another IR. Therefore, frontend compilers that (transitively) require the arithmetic IR extension can only be composed with compiler backends that also support this extension. Since most Datalog solvers have built-in support for arithmetics, this is not a limitation. But, what if a language feature is not supported by existing Datalog solvers?

4.2 Multi-level IR Extension for Booleans

Languages that compile to Datalog often feature conditional constructs based on Boolean conditions. However, core Datalog does not provide abstractions for Boolean values or their operations. Rather than requiring each compiler frontend to encode Booleans as numbers (previous extension), we introduce a dedicated IR extension for Booleans that lowers to arithmetics. Not only does this reduce the effort for compiler frontends, it also enables Boolean optimizations as we detail in item 4.6. We add the following IR nodes:

$$\begin{array}{lll} \text{(terms)} & t & ::= \ldots \mid \mathsf{true} \mid \mathsf{false} \mid t \ \mathit{boolBinOp} \ t \mid \neg t \mid \mathsf{asBool}\ a \\ \text{(types)} & T & ::= \ldots \mid \mathsf{Bool} \\ \multicolumn{3}{l}{\mathit{boolBinOp} \in \{\&\&, ||, \ldots\}} \end{array}$$

Besides standard Boolean literals and operations, we also introduce an IR node to convert an atom into a Boolean term: asBool a. The reverse operation that encodes a Boolean term t as an atom can be easily encoded: $t = \mathsf{true}$. The typing rules for Boolean IR nodes are analogous to the rules for arithmetics. However, Boolean IR nodes do not need to be supported by compiler backends, we can lower them to other IR nodes instead.

We describe lowerings as a recursive translation function $[\![\cdot]\!]$ that maps IR nodes of an extension to other IR nodes. We provide the following lowering for the Boolean IR:

$$\begin{array}{lll} [\![\mathsf{true}]\!] = 1 & [\![\mathsf{false}]\!] = 0 & \\ [\![t_1 \,\&\&\, t_2]\!] = [\![t_1]\!] \ \mathit{min} \ [\![t_2]\!] & [\![t_1 \,||\, t_2]\!] = [\![t_1]\!] \ \mathit{max} \ [\![t_2]\!] & [\![\neg t]\!] = 1 - [\![t]\!] \\ \multicolumn{3}{l}{[\![\mathsf{asBool}\ a]\!] = \mathsf{fresh}\ R\ \mathsf{in}\ \{\{a, R = 1\} \vee \{\mathsf{not}\ a, R = 0\}; R\}} \end{array}$$

We encode Boolean literals as 1 and 0, respectively. Conjunction becomes *min*, disjunction becomes *max*, and negation becomes subtraction from 1. But what is going on in the last equation? To encode an atom as a Boolean, we must test two alternatives: atom a succeeds or it fails. Based on this observation, we set a fresh variable R to 1 or 0 accordingly. In place of the original IR node, this variable R can then be used as a Boolean term.

The lowering for asBool is quite involved and uses three additional IR extensions not introduced yet: an extension for negating atoms (not), an extension for blocks that allows atoms to appear within terms, and an extension for disjunctions that supports nested alternatives (rather than top-level rules only). Before discussing these extensions, we should inspect the typing rule for asBool to validate that the lowering preserves typing:

$$\frac{\Gamma_1 \vdash^{-} a \text{ ok} \dashv \Gamma_2}{\Gamma_1 \vdash^{\#} \text{asBool } a \Rightarrow \text{Bool} \dashv \Gamma_2} \text{ I-AsBool}$$

The typing rule specifies that asBool computes a Boolean value. Indeed, the lowering yields x of type Int, which is the type Bool lowers to. Other than that, the typing rule requires that a is valid, namely in a negative context. This strictness point is crucial because the lowering applies not to a. For example, consider asBool$(X^{\circ} = 5)$ with an unbound X. This code is only valid in a positive context that can bind X. Had we checked a under a positive polarity, this example would be well-typed but the lowered code would not: fresh R in $\{\{X =^{+} 5, R = 1\} \vee \{X =^{-} 5, R = 0\}; R\}$. Since X is unbound, the negative equation in the second alternative is invalid. This is a good example, illustrating why variable resolution needs to be validated together with type compatibility: The binding rules are complex such that generating valid code is not easy. Our validation procedure runs after each lowering step such that we can identify erroneous lowerings.

Another multi-level IR extension we have implemented is first-class sets. In Datalog, sets are encoded as relations, which are always second-class entities at the top-level. We added an IR extension for creating and computing with set values. This is inspired by Pacak and Erdweg [2022], who compile a functional programming language to Datalog. They eliminate first-class sets by defunctionalization, which creates top-level relations that dispatch on the set constructor. We realized that their approach is not specific to their particular frontend language, but useful and realizable across frontends in the IR.

4.3 Structural IR Extensions

The structure of Datalog is rigid, making it unnecessarily complicated to generate Datalog code:

- Negation is a flag on calls and equations, not an operation on atoms.
- Atoms can only occur at the top-level of rules, but not be nested.
- Alternatives can only be encoded as top-level rules, but not within a single rule.

The literature on code generation has long recognized that such structural constraints create tension for code generators. For example, expression-based languages such as Racket benefit from having only a few syntactic categories [Flatt et al. 2023]: definitions, bindings, expressions. To target more rigid languages such as Java, where statements may not appear within expressions, Bravenboer and Visser [2004] introduced *expression blocks* $\{s_1; \ldots; s_n; e\}$ that desugar to standard Java by lifting the statements and leaving the expression in place. We follow this line of work and introduce auxiliary IR extensions that improve the targetability of Datalog IR.

$$\begin{array}{lrcl} \text{(atoms)} & a & ::= & \ldots \mid \text{not } a \mid \{\overline{a}\} \mid a \vee a \\ \text{(terms)} & t & ::= & \ldots \mid \{a; t\} \end{array}$$

We allow not for negating atoms, atoms can be grouped into a block $\{a_1, \ldots, a_n\}$, and we can describe disjunctions inline. Lastly, we allow terms to contain blocks, akin to expression blocks. The lowering of these extensions is fairly simple. First, we transform the program into disjunctive normal form as required by core Datalog. That is, disjunctions become top-level alternative rules that contain (possibly negated) atoms, which do not contain any more nested atoms. Second, eliminate negation based on an extensible mapping of supported atoms: flip the polarity of calls and equations, eliminate double negations. We don't show the definition of the lowerings here, but instead focus on the typing rules for these extensions.

The typing rules for our structural IR extensions not, block, and disjunction are non-trivial. Again, this is because we need to capture variable resolution precisely. We have deliberately set up our type system to accommodate structural IR extensions and their typing:

$$\frac{\Gamma_1 \vdash^{-\cdot\#} a \text{ ok} \dashv \Gamma_2}{\Gamma_1 \vdash^{\#} \text{not } a \text{ ok} \dashv \Gamma_2}\ \text{A-Not}$$

$$\frac{\forall i.\ \Gamma_i \vdash^{\#} a_i \text{ ok} \dashv \Gamma_{i+1}}{\Gamma_1 \vdash^{\#} \{a_1, \ldots, a_n\} \text{ ok} \dashv \Gamma_{n+1}}\ \text{A-Block} \qquad \frac{\Gamma_1 \vdash^{+} a \text{ ok} \dashv \Gamma_2 \qquad \Gamma_2 \vdash^{\#} t \Rightarrow T \dashv \Gamma_3}{\Gamma_1 \vdash^{\#} \{a; t\} \Rightarrow T \dashv \Gamma_3}\ \text{I-Block}$$

$$\frac{\Gamma \vdash^{\#} a_1 \text{ ok} \dashv \Gamma_1 \qquad \Gamma \vdash^{\#} a_2 \text{ ok} \dashv \Gamma_2}{\Gamma \vdash^{\#} a_1 \vee a_2 \text{ ok} \dashv \Gamma_1 \sqcap \Gamma_2}\ \text{A-Or}$$

The typing rule for not a may appear trivial, but note the polarity of the premise $- \cdot \#$. Recall from Section 3 that polarities can be multiplied like signs. Therefore, $- \cdot \#$ inverts the sign of #. The negated atom a may bind variables if and only if the not a appears in a negative context. In particular, note that the polarity of not (not a) is equal to the polarity of a; double negation is eliminated:

$$\frac{\dfrac{\Gamma_1 \vdash^{\#} a \text{ ok} \dashv \Gamma_2}{\Gamma_1 \vdash^{-\cdot\#} \text{not } a \text{ ok} \dashv \Gamma_2}\ \text{A-Not}}{\Gamma_1 \vdash^{\#} \text{not (not } a) \text{ ok} \dashv \Gamma_2}\ \text{A-Not}$$

Consequentially, a term may bind variables if it is wrapped in an even number of negations.

The rule for atom blocks $\{a_1, \ldots, a_n\}$ should be unsurprising. However, the rule for term blocks $\{a; t\}$ is interesting, because it unconditionally checks the embedded atom under a positive polarity. This is because the lowering will eventually lift the atom to the top of a rule, where all atoms are checked positively (cf. Rule in Figure 4).

For disjunctions $a_1 \vee a_2$, we again have to ensure variables are correctly resolved. In particular, we must account for situations where one alternative binds a variable but the other does not. This is usual for flow-sensitive type systems, where the bindings of alternative branches have to be merged. We extend the merging to our typing contexts, which distinguish three variable states: undefined, declared, and bound. Specifically, typing contexts form a lattice $X \mapsto T \times \{\notin, \circ, \bullet\}$, where unbound variables default to $\bot^{\notin}$. The merge operation then is the standard meet operation on this lattice, given the order $\notin\, < \circ < \bullet$ on variable bindings. This means, a variable is only bound after a disjunction if it is bound in both branches, it is unbound if it is unbound in either branch, and it is declared in all other cases. Should the resulting context map a variable to a bottom type $\bot^{\circ}$ or $\bot^{\bullet}$, this indicates a type error: The branches assigned conflicting types to the variable.

4.4 Demanded Parameters, a Magical IR Extension

The demand transformation (aka., magic set transformation) is an optimization technique for Datalog, where the domain of relations is narrowed to fit its queries, avoiding the derivation of irrelevant tuples [Tekle and Liu 2010]. But in generating Datalog code, the demand transformation fulfils another purpose: It can be used to encode the control flow of a frontend language [Pacak and Erdweg 2022]. Technically, the demand transformation enforces input/output modes on the columns of all relations R. It does that by generating an auxiliary input relation input_R, which enumerates all inputs provided to R. Rules of R are then augmented with an additional guard $\mathsf{R}(X_{In})$, binding all input variables R. For example, this makes it feasible to encode functions with a infinite domain as Datalog relations:

$$\mathsf{inc}(I, R) :- \mathsf{input\_inc}(I), R = I + 1.$$

Without the call to input_inc, this rule fails to type check, because I is not bound in the addition.

We want to support the demand transformation as an IR extension. To this end, we extend relation signatures to allow annotations of input and output positions:

$$\begin{array}{lll} \text{(signatures)} & sig & ::= \dots \mid p : PT \times \dots \times PT \\ \text{(parameter types)} & PT & ::= T^{\circ} \mid T^{\bullet} \end{array}$$

Parameter type T° is the default and corresponds to a regular, non-demanded parameter. In contrast, parameter type $T^{\bullet}$ is a demanded parameter that must be provided as input to the relation. We can directly use these parameter types for the initial context used for checking Datalog rules:

$$\frac{p : PT_1 \times \dots \times PT_n \qquad \Gamma_1 = X_1{:}PT_1; \dots; X_n{:}PT_n \qquad \forall i.\ \Gamma_i \vdash^{+} a_i \text{ ok} \dashv \Gamma_{i+1} \qquad \forall j.\ \Gamma_{n+1} \vdash^{-} X_j \Leftarrow T_j \dashv \_}{\vdash\ p(X_1, \dots, X_n) :- a_1, \dots, a_r.\ \checkmark}$$

$$\frac{p : PT_1 \times \dots \times PT_n \qquad \forall i.\ \ \Gamma_i \vdash^{min(\#_1 \cdot \#_2, sign(PT_i))} t_i \Leftarrow T_i \dashv \Gamma_{i+1}}{\Gamma_1 \vdash^{\#_1} p^{\#_2}(t_1, \dots, t_n) \text{ ok} \dashv \Gamma_{n+1}}$$

We also adopted the rule for calls to account for parameter types with demand. Essentially, we make sure that arguments for demanded parameters always have negative polarity. To this end, we compute the polarity for an argument through a sequence of helper functions:

$$min(\#_1, \#_2) = \begin{cases} - & \text{if } \#_1 = - \text{ or } \#_2 = - \\ +, & \text{otherwise} \end{cases} \qquad sign(PT) = \begin{cases} + & \text{if } PT = T^{\circ} \\ - & \text{if } PT = T^{\bullet} \end{cases}$$

The typing rules predict the variable binding in the lowered code. A rule $p(X_1, X_2) :- a_1, \dots, a_r$. with demanded parameter X_1 transforms to $p(X_1, X_2) :- input\_p(X_1), a_1, \dots, a_r$. with no demanded parameters. Still, the initial typing context $X_1{:}T_1^{\bullet}; X_2{:}T_2^{\circ}$ of the original rule is valid for $a_1, \dots, a_r$ because the added guard binds X_1. Moreover, the first argument from all calls to p is enumerated in *input_p*, which is safe because they were required to be bound.

This IR extension really highlights the flexibility of our multi-level IR and the expressiveness of its type system. To the best of our knowledge, the impact of the demand transformation on variable resolution has never been captured by a static semantics before.

4.5 Impure Datalog

Datalog frontend languages sometimes provide mutable state or other impure Datalog features. For example, Dedalus features stateful relations for encoding messages between distributed actors [Alvaro et al. 2010]. And one of our case studies supports dynamic object allocation, which is also stateful. We provide a generic IR extension to support these and other impure language features.

$$\begin{array}{l} \text{(atoms)} \quad a \ ::= \dots \mid \mathsf{impure}[IK, T]\ \overline{a} \text{ with } X := t \\ IK \in \textbf{ImpurityKind} \end{array}$$

The impure atom allows Datalog programs to encode state of type T, locally bound to X, and updated by t. Conceptually, an impure atom of kind k reads and writes to a global variable ST_k:

$$\begin{array}{lcl} \mathsf{impure}[k, \mathsf{Int}]\ P(S) \text{ with } S := S + 1 & \rightsquigarrow & S := ST_k, \quad P(S), \quad ST_k := S + 1 \\ \mathsf{impure}[k, \mathsf{Int}]\ Q(R) \text{ with } R := 0 & & R := ST_k, \quad Q(R), \quad ST_k := 0 \end{array}$$

Technically, the lowering encodes the global state by state passing for each impurity kind: throughout all relations that (transitively) interact with an impure atom of a given impurity kind, the containing relation is extended with an input and an output parameter for that state. Within each such relation, the state is threaded from one atom to the next, writing the updated state to fresh

Datalog variables.

$$\frac{\Gamma_1 = \Gamma; X{:}T^{\bullet} \qquad \forall i.\ \ \Gamma_i \vdash^{\#} a_i \text{ ok} \dashv \Gamma_{i+1} \qquad \Gamma_{n+1} \vdash^{-} t \Leftarrow T \dashv \Gamma_{n+2} \qquad \Gamma' = \begin{cases} \Gamma_{n+2}; X{:}\Gamma(X) & \text{if } X \in \Gamma \\ \Gamma_{n+2} - X & \text{if } X \notin \Gamma \end{cases}}{\Gamma \vdash^{\#} \text{impure}[k, T]\ a_1, \ldots, a_n \text{ with } X := t \text{ ok} \dashv \Gamma'}$$

Our type system can precisely capture the contract of the impure atom. Within the impure context, the local state variable X is bound as defined by Γ_1. After processing the atoms, t must yield an updated state of type T. In the end, since X is only accessible locally, we restore the binding of X from the original context Γ.

4.6 Summary

We have implemented 16 typed IR extensions and presented a subset of them in this section. In particular, we have demonstrated that IR extensions can gracefully extend the type system of the Datalog IR to support validity checking. It should be noted that almost all IR extensions interact with variable resolution, but for different reasons:

(1) The extension defines strictness points (e.g., arithmetics, Boolean, data, set, tuples, aggregation).
(2) The lowering generates auxiliary variables that may not shadow existing ones [Erdweg et al. 2014] (e.g., Boolean, set, tuples, impure).
(3) The extension moves or negates atoms to change their polarity (e.g., asBool, negation, blocks).

The design of a type system for Datalog IR that can accurately describe variable bindings is a key contribution of this paper. We have shown the type system not only captures the static semantics of core Datalog, but can be extended to specify complex constraints for various IR extensions. This way, our compiler framework can check the validity of generated IR code at each level of the IR. That is, we can detect invalid IR code where it has been generated and blame the corresponding generator: a compiler frontend, an IR lowering, or an optimization.

Our compiler framework supports the description of compiler pipelines that interleave optimizations and IR lowerings. This paper is really focused on the design of an extensible Datalog IR and its type system, and we do not have space to describe all optimizations at length. However, our implementation already supports some optimizations, which can either run at any IR level or at specific IR levels. In the next section, we will first extend our framework with multiple frontends and backends and then discuss some of these level-specific optimizations.

5 Evaluation

We first implement our compiler framework and extend it with multiple frontend and backend systems as indicated by Figure 1. Afterward, we show that the multi-level nature of our IR enables new possibilities for optimizations, that are only possible as long as an IR extension is not yet lowered. That is, existing backend systems, such as Soufflé are unable to optimize the code on their own without our compiler framework.

5.1 Connecting Existing Datalog Frontends and Backends

We claim that the typed Datalog IR establishes a solid foundation for a generic compiler framework for Datalog. As such, a compiler framework using the IR should be able to target various existing Datalog solvers, and it should be possible to build compiler frontends for various existing Datalog dialects. To evaluate this claim, we have implemented an actual compiler framework for Datalog based on the IR described here and we have built 4 compiler frontends and 3 compiler backends.

Implementation of the IR. We have implemented the extensible IR as described in this paper as open data types in Scala. We encoded the nonterminals of the IR as interfaces and the concrete

IR nodes as classes implementing them. Each IR node can extend the core type system, and we use mix-in inheritance to compose the various extensions into a single type checker. Overall, the implementation is relatively close to the formal development presented in the paper. The code is available open source.[3]

Building Compiler Frontends. We evaluated if our compiler framework and the Datalog IR are general enough by developing 4 compiler frontends for existing Datalog dialects. First, we support the Datalog dialect of bddbddb, a Datalog engine that is known for its binary decision diagram usage [Whaley and Lam 2004]. Second, we developed a frontend for Soufflé, which already required quite a few IR extensions. Third, we support IncA's functional Datalog frontend and compile it to the IR, for which we reimplemented the compiler from Pacak and Erdweg [2022] in our framework, targeting a heavily extended Datalog IR. And finally, we have developed a compiler frontend for an object-oriented Datalog dialect that supports dynamic dispatch, dynamic object allocation, and mutable fields [Klopp et al. 2024].

Note that a pure Datalog frontend, extended with all of our IR extensions could also be beneficial for implementing logic programs directly. We have not implemented such a frontend yet and instead focused on compiling frontend dialects. However, the only thing missing for such a frontend is a suitable parser.

Frontend language	Impl. size	Used IR extensions
bddbddb	177 LoC	arithmetics
Soufflé	299 LoC	arithm., bool, alg. data, strings, not, block, or
Functional IncA	243 LoC	arithm., bool, alg. data + match, demand, sets, maps, strings, tuples, aggregate, not, block, or, polymorphism
Object-oriented	675 LoC	like IncA + impure + mono aggregate

The table above summarizes our experience: It is relatively easy to add compiler frontends. The code size for the IR extensions is not important here (and not included in Impl. size), because their implementations are shared across frontends. For example, the object-oriented Datalog dialect uses all of the same IR extensions as functional IncA, plus two additional extensions. We expect to discover more useful IR extensions as we connect additional Datalog dialects to our compiler framework.

Targeting Existing Solvers. We developed three compiler backends for our Datalog compiler framework. One backend targets Soufflé [Scholz et al. 2016], the other targets the Viatra solver used by IncA [Szabó et al. 2016], and the third targets the Ascent engine in Rust [Sahebolamri et al. 2022]. Target platforms support different feature sets; the compiler backend can choose which IR extensions it supports directly rather than using a lowering. For example, Soufflé has built-in support for arithmetics, strings, algebraic data, and non-recursive aggregation. The compiler backend for Soufflé translates the corresponding IR nodes directly to Soufflé code. Soufflé also supports nested disjunction and negation, but we opted to lower these features in our compiler framework for compatibility with other extensions. Technically, we realized the backend as a tree-to-tree transformation from the IR to a Soufflé AST in 100 LoC in Scala, supported by 277 LoC for the AST definition and printer.

The second compiler backend targets IncA's incremental Datalog solver Viatra [Varró et al. 2016]. Viatra is a Java API that can be used to create *graph patterns*, which correspond to Datalog relations. Viatra does not support any Datalog extensions directly, but it can execute any JVM code to produce or modify values. To target Viatra, we added an auxiliary IR extension that embeds Scala types

[3]https://gitlab.rlp.net/plmz/inca-scala

and code into Datalog. We then first lower the IR extensions for arithmetics, strings, algebraic data types, and aggregation to the Scala IR extension. And then we translate Datalog with embedded Scala to Scala code that invokes the Viatra API in 271 LoC.

Lastly, the third compiler backend generates Rust code that exercises the Ascent Datalog engine. An Ascent Datalog program is a Rust program that uses Rust macros to declare Datalog rules. We modeled a subset of the Rust AST and translate the Datalog IR to that AST in 213 LoC, supported by 408 LoC for the AST and its printer.

There are two lessons to take away from the 3 compiler backends we developed. First, developing compiler backends is relatively low effort, because backends can choose which features they want to support directly. Even for Viatra, which does not provide a Datalog-like input language, we were able to realize the backend in a few hundred lines of code. This indicates that it is feasible to connect a wide range of existing Datalog solvers to our compiler framework. Second, our compiler framework enables us to run the same Datalog program against multiple existing Datalog solvers. This is a cornerstone for the development of Datalog optimizations, making it possible to evaluate the effect of an optimization on different solvers. For example, we were able to execute a Doop analysis written in Soufflé with all three Datalog backends through our compiler framework.

Engine	Soufflé	Soufflé parallel	Viatra	Ascent	Ascent parallel
Running time (s)	10.29	8.42	14.62	29.96	18.13

Notably, all runs stem from a single Datalog source file, processed by a single compiler frontend, passed through a single optimization pipeline, and then handed off to three different backends for execution. The performance of Datalog engines has never been more easy to compare.

5.2 Exploiting Higher-level Abstractions for Optimizations

Besides supporting multiple frontends and backends, the multi-level IR of our compiler framework also opens up the possibility for new optimizations. We can leverage domain-specific knowledge at each abstraction level to apply optimizations before lowering them. Conversely, the optimization potential for low-level code that does not use high-level abstractions is rather small. For instance, the Doop analyses written in Soufflé are low-level and hard to optimize because they only leverage three extensions: strings, arithmetics, and nested disjunctions. However, when compiling complex frontend languages to Datalog as illustrated in Section 2, we often use high-level abstractions to decrease the implementation effort. Therefore, generated Datalog code can often be optimized when using our multi-level IR.

We showcase two optimizations our compiler framework enables: a Boolean optimization and an optimization for sets. These optimizations are only applicable before lowering because, after lowering, the necessary algebraic properties are no longer evident.

Optimizing Boolean Expressions. Consider we want to compile a functional program to Datalog (as done by Pacak and Erdweg [2022]) and optimize it using our compiler framework. We show a complete example in Figure 5, where function `f` uses conditionals over two input parameters `x` and `y` to select a sub-computation f_i. We compile this program to Datalog using the Boolean IR extension as shown in the top-left Datalog code in Figure 5: Each sub-computation is guarded by the conditions along its path, negating the `if` condition when selecting the `else` path. Observe that sub-computation f_2 is actually unreachable since it is not possible for `¬x && y` and `x && y` to be true simultaneously. However, without optimization this property gets lost. The bottom-left Datalog excerpt in Figure 5 shows the lowered code without optimization. In particular, we obtain `min(min(1-x, y), min(x, y))` for sub-computation f_2, which cannot be reduced to `0` (representing `false`). Thus, the rule's guard has to be evaluated at run time, and sub-computation f_2 is reachable.

```
// A functional program to be compiled to Datalog
def f(x: Bool, y: Bool) = {
  if (x) {
    if (x && y) { f₀ }
    else { f₁ }
  } else if (y) {
    if (x && y) { f₂ }
    else { f₃ }
  } else {
    f₄
  }
}
```

Compiled without optimization

```
f(x, y) :- (x && (x && y)) = true, f₀.
f(x, y) :- (x && ¬(x && y)) = true, f₁.
f(x, y) :- ((¬x && y) && (x && y)) = true, f₂.
f(x, y) :- ((¬x && y) && ¬(x && y)) = true, f₃.
f(x, y) :- (¬x && ¬y) = true, f₄.
```

Compiled with optimization

```
f(x, y) :- (x && y) = true, f₀.
f(x, y) :- (x && ¬y) = true, f₁.
f(x, y) :- false = true, f₂. // fails
f(x, y) :- (¬x && y) = true, f₃.
f(x, y) :- (¬x && ¬y) = true, f₄.
```

Lowered without optimization

```
f(x, y) :- min(x, min(x, y)) = 1, f₀.
f(x, y) :- min(x, 1-min(x, y)) = 1, f₁.
f(x, y) :- min(min(1-x, y), min(x, y)) = 1, f₂.
f(x, y) :- min(min(1-x, y), 1-min(x, y)) = 1, f₃.
f(x, y) :- min(1-x, 1-y) = 1, f₄.
```

Lowered with optimization

```
f(x, y) :- min(x, y) = 1, f₀.
f(x, y) :- min(x, 1-y) = 1, f₁.

f(x, y) :- min(1-x, y) = 1, f₃.
f(x, y) :- min(1-x, 1-y) = 1, f₄.
```

Fig. 5. A functional program with conditionals and the corresponding generated Datalog code in our multi-level IR. The generated code contains booleans that we lower to arithmetics. On the left, we lower the Boolean code without optimization; on the right, we optimize the Boolean code before lowering.

We have implemented an optimization for the Boolean IR that rewrites Boolean formulas according to their algebraic properties. Using this optimization, we obtain the top-right Datalog code in Figure 5 before lowering. We illustrate the effect of the optimization using colors: red marks deletions in the unoptimized code and blue marks insertions in the optimized code. This way, we can see that all but the last rule have been improved by the optimization. The subsequent lowering yields the bottom-right Datalog program, where the third rule was eliminated due to the contradiction `0 = 1`. This shows how our multi-level Datalog IR allows optimizations to target the appropriate abstraction in a way that would not be possible if all optimizations happened after lowering to core Datalog.

Optimizing Set Unions. Our multi-level Datalog IR enables optimizations that have a significant performance impact. To this end, we designed, implemented, and micro-benchmarked an optimization for set operations to show the performance impact that extension-specific optimization can have. As shown by Functional IncA [Pacak and Erdweg 2022] and DataFun [Arntzenius

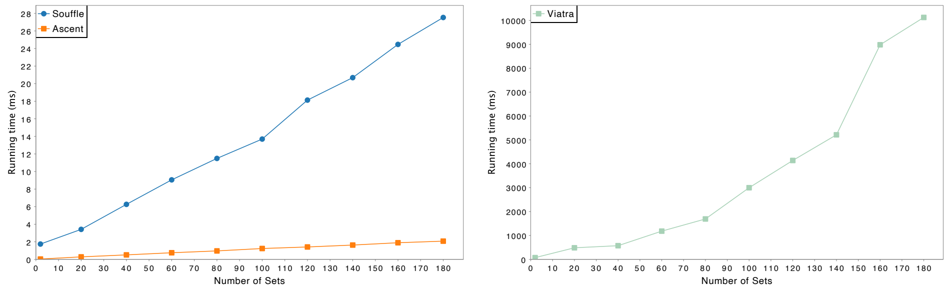

Fig. 6. Measuring the performance impact of computing the union over an increasing number of sets.

and Krishnaswami 2016], sets are a natural extension for Datalog since they enable higher-order relations: Relations that take other relations (encoded as sets) as inputs.

Our compiler framework supports an IR extension for creating and operating on first-class sets as indicated in Figure 1. We lower sets by defunctionalization as described by Pacak and Erdweg [2022]: We generate algebraic data types (ADTs) whose constructors identify the expression that constructs the set, and we generate a relation that enumerates the elements of a set given its ADT value. For example, each set expression $a \cup b$ in the program yields a unique constructor $\texttt{S}_i$ and the expression is rewritten to $\texttt{S}_i\texttt{(A,B)}$, where `A` and `B` are the defunctionalized encodings of `a` and `b`. The relation that enumerates the elements of $\texttt{S}_i\texttt{(A,B)}$ then features two dedicated rules to propagate the elements of `A` and `B`:

```
S_elem(s, x) :- ?S_i(s, A,B), S_elem(A, x).
S_elem(s, x) :- ?S_i(s, A,B), S_elem(B, x).
```

In principle, Datalog backends could recognize this propagation logic to optimize the generated code. Unfortunately, this is beyond their capabilities as we show empirically using a micro benchmark $(a \cup b) \cup \cdots \cup (a \cup b)$. If we repeat the union of n times, existing Datalog backends require time linear in n. We confirm this analysis by measuring the running times for Soufflé, Ascent, and Viatra for increasing n, as shown in Figure 6. Note that Viatra is in its own diagram because it is orders of magnitude slower, yet it supports incremental Datalog evaluation, which we did not consider here.

We have implemented an optimization for the Set IR extension that automatically optimizes this repeated union to $a \cup b$, which requires constant time independent of n. When repeating the measurements after optimization, the backends indeed take constant time: Soufflé requires approximately 2 ms, Ascent roughly 0.05 ms, and Viatra 70 ms for evaluating the optimized program. This illustrates how our multi-level Datalog IR enables optimizations that significantly improve the running times of Datalog programs. Future work can develop many more interesting optimizations based on the foundations developed in this paper.

6 Related work

We propose a novel extensible, typed multi-level IR for Datalog to build a flexible compiler framework for Datalog. This is in stark contrast to the state of the art, where each Datalog system provides its own isolated compiler pipeline.

The idea of a multi-level IR was first popularized by the LLVM compiler framework [Lattner et al. 2021]. Originally, LLVM was designed with a single IR in mind to represent and type check code language-independently [Lattner and Adve 2004]. There are more than 40 LLVM IR extensions (called dialects), including extensions for basic arithmetic, vectors, tensors, higher-order functions, and asynchronous computations.[4] In general, a multi-level intermediate representation (IR) is defined by distinctive design principles [Lattner et al. 2021] which we follow closely:

(1) *Little Builtin, Everything Customizable.* The IR is based on a minimal number of fundamental concepts to facilitate customization. These abstractions are easy to extend and, in our framework, include modules, module entries, atoms, terms, and types.
(2) *Region-Based Data and Control Structures.* To maintain low compiler complexity, the IR supports data and control structures that are transformed into normalized representations. We introduce such structures in the form of top-level disjunctions, blocks, and pattern matching on algebraic data.
(3) *Progressivity.* The compiler framework progressively lowers abstractions to a core IR, supporting analysis and optimizations at each level. Our design achieves this by building IR extensions on top of a core Datalog IR.
(4) *Declaration and Validation.* Transformations should be expressed as rewriting rules, separate from the IR nodes. The framework must validate the generated code after each transformation. We enable rewritings by exploiting the visitor pattern and type the generated IR code after each transformation.

Below, we first discuss related work on extensible IRs, followed by work on type systems and compilers for Datalog.

Extensible IRs. There is no universally accepted definition for an *extensible intermediate representation* in the literature. Nonetheless, various approaches describe mechanisms to extend their intermediate representation with new features. However, none of these approaches implement progressive lowerings and declarative transformations, which are essential for a multi-level IR.

Graal IR [Duboscq et al. 2013] is an intermediate representation for a Java just-in-time compiler. New IR nodes are defined declaratively and usually remain during compilation. However, nodes can choose to define a fixed rewriting in terms of other nodes. This is in contrast to the design of multi-level IRs, where IR nodes are progressively lowered in multiple passes to a core IR. Lowerings in a multi-level IR are more flexible and enable different transformations for the same node type.

Thorin [Leißa et al. 2015] is an extensible, higher-order IR based on a typed lambda calculus that enables optimizations at higher levels of abstraction. Users can declare new operations, called axioms, which can be opaque functions, type operators, or any other entity with a type. Multiple axioms are grouped into a dialect. Optimizations operate on multiple axioms, potentially across multiple dialects, to ultimately generate LLVM IR code. In contrast to a multi-level IR, Thorin lacks progressive lowering to express one dialect in terms of another.

INSPIRE [Jordan et al. 2013] is a unified, high-level, parallel intermediate representation designed for parallel applications. To accommodate changing requirements over time, INSPIRE emphasizes extensibility. In particular, the IR consists of a fixed set of core constructs that provide abstractions for types and operators, which are used to define extensions to the IR. Extensions can be newly defined or derived from existing constructs by composing previously defined elements. In contrast, a multi-level IR supports progressive lowering, where an extension is lowered and optimized through multiple passes to a core IR. Additionally, we decouple the lowerings from the IR node definitions, allowing for alternative lowerings for the same IR nodes.

Type Systems for Datalog. Only few studies on the typing of Datalog exist in the literature, and none of them considers a modularly extensible Datalog IR.

Formulog [Bembenek et al. 2020] extends Datalog by incorporating a first-order functional language and built-in support for SMT formulas. The language is statically typed and its type

[4]https://mlir.llvm.org/docs/Dialects/

system ensures type safety for Datalog: types are compatible and variables are bound before they are read. Formulog achieves safe variable handling by separating expressions from Datalog variables during typing. Datalog variables are checked using a flow-sensitive typing judgment for Datalog terms. However, if a term is not a variable, Formulog handles the term as an expression, which may only read but never bind variables. Moreover, Formulog specifies strictness points (e.g., in negative calls) by requiring the typing context to be unchanged. Both techniques work well for their setup, but precludes many IR extensions we have implemented.

For example, consider a tuple IR extension that allows equations such as $(X, Y) = (1, 2)$. Since (X, Y) is not a variable, Formulog would handle it as an expression that may not bind variables. Similarly, a block that binds variables within a term $3 + \{A = 1, B = 2, B - A\}$ is not supported by Formulog, because the outermost addition term moves the type checker into expression mode, which may not bind the variables A and B. Another problematic extension is atom negation, because $\mathrm{not}(\mathrm{not}P(X, Y))$ should bind variables, but Formulog's type system would reject this program. We also could not support the demand IR extension in Formulog, because Formulog rewrites $p(e_1, e_2)$ to $p(X, Y), X = e_1, Y = e_2$, which is only valid when neither X nor Y is demanded by p.

The fundamental issue is that there is no way to recover from a strictness point in Formulog: Once the type system shifts to expression mode, it stays there. Our solution is to forgo the distinction between terms and expressions, but to include a polarity flag instead. A term typed under negative polarity corresponds to Formulog's expression typing, whereas a positive polarity corresponds to Formulog's term typing. This makes our type system more uniform and allows us to switch from one mode to the other. For example, the IR extension for blocks can explicitly check the contained atoms in positive mode and the negation extension can flip the polarity.

While we and Formulog associate atomic types with terms (e.g., int, bool, set of int), other Datalog type systems have focused on identifying constraints between relations. For example, LogicBlox features a type system that can encode inclusion constraints on tuples, such as: if `parent(X,Y)` then `person(X)` and `person(Y)` [Zook et al. 2009]. This is useful for ensuring referential integrity in database systems, an issue orthogonal to the type-safety property we are after.

In the same vein, de Moor et al. [2008] use typing inference for optimizing Datalog programs. The idea is to trace which elements an (intermediary) relation can contain and to use this information for optimizations. In particular, a type error occurs when a relation is empty and the atom is known to fail. Follow-up work uses more complex type hierarchies that not only allow for subtyping checks, but also for disjointness and equality tests [Schäfer and de Moor 2010]. Datalog programmers can annotate such relationships in the program and type inference tracks corresponding dependencies between tuple elements. This line of work is quite different from ours in that it does not ensure type safety because variable bindings are imprecise, while we do not try to use typing for optimizations. Indeed, to integrate their type inference into our compiler framework, we would model it as a compiler pass rather than incorporating it into our type system.

Frontends/Backends. We provide a unified platform for different Datalog frontends and backends. Frontends can target our Datalog IR and we can compile the IR to existing backends.

Soufflé [Scholz et al. 2015] provides a Datalog dialect with an efficient Datalog solver. Soufflé provides a static type system for its Datalog dialect and a separate analysis to ensure range restrictedness. Nevertheless, Soufflé does not provide any formalization of the type system. Our Datalog IR verifies type compatibility and range restrictedness in a single, extensible type system. Soufflé allows disjunctions within rules, that are lowered to multiple rules. We have integrated Soufflé as a Datalog backend and frontend in our system. That way, programs that target our Soufflé backend can profit from optimizations we implement for various IR levels. For example, we provide specialized optimizations for boolean, sets or demanded parameters.

Functional IncA [Pacak and Erdweg 2022] is a statically typed, functional language with fixpoint computations that compiles to Datalog. It employs defunctionalization to encode first-class sets and encodes the control-flow of a functional program using the demand transformation [Tekle and Liu 2010]. Based on Functional IncA's defunctionalization approach, we designed an IR extension for first-class sets in Datalog. Instead of explicitly computing the demand transformation, we track demand through the type system and transform the program in a lowering step. As shown in our evaluation, our design is flexible enough to implement Functional IncA on top of our multi-level IR. In contrast to the monolithic compiler of Functional IncA, the re-implementation reuses many IR extensions which allows the compiler to reduce the compilation semantics complexity drastically.

Dedalus [Alvaro et al. 2010] is a language for distributed systems that reduces to Datalog with negation and aggregation. In these distributed system the state evolves with the execution. Dedalus introduces a notion of time to Datalog to reason about state. Our Datalog IR design can represent a similar notion of time in Datalog using the impure IR. Using impurity, we can thread a timestamp throughout a Datalog program that increases each time a change is made.

Egglog [Zhang et al. 2023] combines statically typed Datalog with equality saturation. The frontend language is parsed, typed, compiled and then executed with a custom solver. We support multiple frontends and backends for our multi-level IR and type check at each level. Egglog is a potentially interesting frontend and backend for our framework.

QL [Avgustinov et al. 2016] is an object-oriented language that is executed by an efficient Datalog-like solver. Since the solver specific details of QL are closed source, it is hard to assess if QL can be used as a possible backend. However, our multi-level IR should be expressible and flexible enough to implement QL's frontend language, as our object-oriented case study shows.

Flix [Madsen and Lhoták 2020; Madsen et al. 2016] is a functional-first programming language that incorporates Datalog programs as first-class values. Datalog expressions undergo type checking to ensure consistent types for predicate symbols and their terms throughout a Flix program, but the type system does not ensure range restrictedness. Given its features, Flix presents an interesting option as a frontend language for our multi-level IR. We envision the ability to compile and execute the Datalog portion of Flix to our multi-level IR to leverage the benefits of different backends.

DDLog [Ryzhyk and Budiu 2019] is a dialect of Datalog featuring a custom backend for automated incremental computations. Its dialect supports relations, variables, and functions. It includes a type system with Booleans, integers, bit-vectors, strings and tuples, but no recursive types are allowed. We support an extensible multi-level IR with type checking and we guarantee range restrictedness at each level. Our Datalog IR supports recursive data definitions in a dedicated IR extension.

Besides these combined frontend/backend solutions, there are multiple Datalog engines, that are interesting to support as Datalog backends in our IR. Viatra [Varró et al. 2016] is the incremental Datalog solver used by IncA, which we already support as a target. Ascent [Sahebolamri et al. 2022] is a macro-based Datalog solver written in Rust, that we also support as an additional backend. Hydroflow [Samuel 2021] is primarily a low-level compilation target for declarative cloud programming languages. It supports creating instances from Datalog code and therefore might become an interesting backend target in the future.

7 Conclusion

We designed the first compiler framework for Datalog. While existing Datalog systems are completely isolated, our compiler framework enables programs written in different Datalog dialects to interoperate at the IR level. It also allows optimizations to be formulated against the IR, which makes them reusable across Datalog compilers.

The compiler framework is based on a typed multi-level Datalog IR, which is the key conceptual contribution of this paper. IR extensions not only introduce new IR nodes and types, they also extend the type system and (optionally) provide a type-safe lowering. We carefully designed an extensible type system for the Datalog IR that is bidirectional, flow-sensitive, bipolar, and uses a three-valued typing context. This type system is precise enough to ensure valid variable bindings and flexible enough to support a wide range of extensions. Indeed, most of our IR extensions interact with variable binding and require the full expressiveness of our type system.

To validate our compiler framework, we implemented 4 compiler frontends and 3 compiler backends. These use 16 IR extensions that we developed, including extensions arithmetic, Booleans, and disjunctions. But our compiler framework really shines in supporting modular IR extensions whose precise static semantics has never been captured before, such as the IR extensions for demanded parameters and impure atoms. In future work, we will connect more compiler frontends and backends to the compiler framework and explore new IR extensions to support them. Additionally, we will explore optimizations for the core Datalog IR and its extensions systematically.

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

Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6702)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.). Springer, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotic, and Bernhard Scholz. 2022. Building a Join Optimizer for Soufflé. In *Logic-Based Program Synthesis and Transformation - 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13474)*, Alicia Villanueva (Ed.). Springer, 83–102. https://doi.org/10.1007/978-3-031-16767-6_5

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, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

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

Martin Bravenboer and Eelco Visser. 2004. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 365–383. https://doi.org/10.1145/1028976.1029007

Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. 2008. Type Inference for Datalog and Its Application to Query Optimisation. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Vancouver, Canada) *(PODS '08)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/1376916.1376957

Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation.

Sebastian Erdweg, Tijs van der Storm, and Yi Dai. 2014. Capture-Avoiding and Hygienic Program Transformations. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*

(Lecture Notes in Computer Science, Vol. 8586). Springer, 489–514.

Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. 2023. Rhombus: A New Spin on Macros without All the Parentheses. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023).

Hervé Gallaire and Jack Minker (Eds.). 1978. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977.* Plemum Press, New York. https://doi.org/10.1007/978-1-4684-3384-5

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

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. *codeQuest:* Scalable Source Code Queries with Datalog. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*, Dave Thomas (Ed.). Springer, 2–27. https://doi.org/10.1007/11785477_2

Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: the insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) *(PACT '13)*. IEEE Press, 7–18.

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

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2015), 202–212. https://api.semanticscholar.org/CorpusID:12679069

Florian Lorenzen and Sebastian Erdweg. 2013. Modular and automated type-soundness verification for language extensions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 331–342. https://doi.org/10.1145/2500365.2500596

Florian Lorenzen and Sebastian Erdweg. 2016. Sound type-dependent syntactic language extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 204–216. https://doi.org/10.1145/2837614.2837644

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

Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.7

André Pacak and Sebastian Erdweg. 2023. Interactive Debugging of Datalog Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 745–772. https://doi.org/10.1145/3622824

André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernhard Scholz and Yukiyoshi Kameyama (Eds.). ACM, 20–32. https://doi.org/10.1145/3564719.3568686

Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings, Vol. 2368)*, Mario Alviano and Andreas Pieris (Eds.). CEUR-WS.org, 56–67. http://ceur-ws.org/Vol-2368/paper6.pdf

Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. 2022. Seamless deductive inference via macros. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 77–88. https://doi.org/10.1145/3497776.3517779

Mingwei Samuel. 2021. *Hydroflow: A Model and Runtime for Distributed Systems Programming.* Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-201.html

Max Schäfer and Oege de Moor. 2010. Type Inference for Datalog with Complex Type Hierarchies. *SIGPLAN Not.* 45, 1 (jan 2010), 145–156. https://doi.org/10.1145/1707801.1706317

Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 196–206. https://doi.org/10.1145/2892208.2892226

Bernhard Scholz, Kostyantyn Vorobyov, Padmanabhan Krishnan, and Till Westmann. 2015. A Datalog Source-to-Source Translator for Static Program Analysis: An Experience Report. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*. IEEE Computer Society, 28–37. https://doi.org/10.1109/ASWEC.2015.15

Pavle Subotic, Herbert Jordan, Lijun Chang, Alan D. Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (2018), 141–153. https://doi.org/10.14778/3282495.3282500

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 *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. https://doi.org/10.1145/3453483.3454026

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

K. Tuncay Tekle and Yanhong A. Liu. 2010. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández (Eds.). ACM, 35–44. https://doi.org/10.1145/1836089.1836094

Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling* 15, 3 (01 Jul 2016), 609–629. https://doi.org/10.1007/s10270-016-0530-4

John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William W. Pugh and Craig Chambers (Eds.). ACM, 131–144. https://doi.org/10.1145/996841.996859

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492. https://doi.org/10.1145/3591239

David Zook, Emir Pasalic, and Beata Sarna-Starosta. 2009. Typed Datalog. In *Practical Aspects of Declarative Languages*, Andy Gill and Terrance Swift (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–182.

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