Mono Types — First-Class Containers for Datalog

Runqing Xu
JGU Mainz, Germany

David Klopp
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

We propose mono types, a new abstraction for programming Datalog. Mono types behave like first-class containers that can be stored in relations and to which elements can be added decentrally. But, mono types are more than just containers: They provide a `read` operation that can yield any result as long as it monotonically grows with each added element and is independent of the order in which elements are added to the container. This design permits a wide range of mono types (e.g., sets, maps, and aggregations), yet guarantees mono types can be integrated into Datalog without jeopardizing Datalog's least fixed-point semantics. We develop a theory for mono types, which includes constructions for complex mono types, equivalence relation for mono types, and properties about semantics-preserving mono-type transformations. This theory ensures sound Datalog integration and justifies crucial compiler optimizations for mono types. Together, these techniques demonstrate that mono types provide abstraction without regret: We demonstrate in two case studies that programs become easier to write with mono types, while their performance also improves drastically.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Datalog, compiler optimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.5

Funding This work is supported by the German Research Foundation (DFG) (Project numbers 451545561 & 508316729) and ERC grant AutoInc (`https://doi.org/10.3030/101125325`).

Acknowledgements We thank the anonymous reviewers for their effort and helpful suggestions.

1 Introduction

Datalog is a programming language for processing structured and cyclic data. Datalog's distinguishing feature is its efficient fixed-point semantics: A program runs until no more deductions are possible, exploiting that programs behave monotonically. Therefore, when processing graphs (as in data-flow or social-network analysis), Datalog programmers can forgo data structures for graph traversals (e.g., visited nodes, queues, stacks) and explicit termination conditions. Instead, the termination is managed by the fixed-point semantics. However, this does not mean all Datalog programs are equally efficient.

Consider a Datalog program that collects the variables shared between two syntax trees:

```
sharedVar(v) :- e1 = Plus(Plus(Var("x1"),Var("x2")),Var("y")),
                e2 = Plus(Var("y"),Plus(Var("z1"),Var("z2"))),
                hasVar(e1, v), hasVar(e2, v).
hasVar(e, v) :- ?Var(e,v).
hasVar(e, v) :- ?Plus(e,lhs,rhs), hasVar(lhs,v).
hasVar(e, v) :- ?Plus(e,lhs,rhs), hasVar(rhs,v).
```

Predicate `sharedVar` constructs two syntax trees `e1` and `e2`, computes their variables using `hasVar`, and returns those variables `v` that occur in both expressions. Predicate `hasVar` traverses

the syntax tree to collect variables. We write `?Var(e,v)` to pattern match `e` as a `Var`, extracting the variable name `v`. In the end, `hasVar` computes a relation that associates each expression with each variable contained in the expression. Thus, `hasVar` will derive 8 tuples for each of `e1` and `e2`, even though they only contain 3 variables each. In general, an expression tree of size n will contain at most $(n+1)/2$ variables, yet `hasVar` computes up to $((n+1) \cdot log\,(n+1))/2$ tuples. That is, our program performs asymptotically worse than necessary.

The problem of our program is that it propagates variables along the tree, generating a separate tuple for an intermediate node. But the problem is more general than that. For example, consider a graph analysis of a social network, where we want to find Alice's most popular friend.

```
mostPopFriend(p, max((p, n))) :- n = count(connectedWith(p,_)).
mostPopFriend(p1, max((p, n))) :- connectedWith(p1, p2), mostPopFriend(p2, (p, n)).
main(p) :- mostPopFriend("Alice", (p, _)).
```

This program uses recursive aggregation to select the person `p` with the most connections `n`. Moreover, since `connectedWith` may span a cyclic graph, the program relies on Datalog's fixed-point semantics, producing a single most popular person in the end. Again this program derives many more tuples than necessary. Initially, `mostPopFriend` contains tuples `(p, (p,n))` for each person `p` and only gradually will the most connected person become known throughout the graph.

In this paper, we propose a Datalog extension that not only eliminates this overhead but also improves the programmability of Datalog. Specifically, we introduce *first-class containers for Datalog* that enable a new programming pattern: Rather than redundantly propagating information from one node to another, we can propagate a single container to collect all information. Conceptually, our containers are mutable collections that can grow over time. Let us revisit the first example:[1]

```
sharedVar(v) :- e1 = Plus(Plus(Var("x1"),Var("x2")),Var("y")),
                e2 = Plus(Var("y"),Plus(Var("z1"),Var("z2"))),
                hasVar(e1, v), hasVar(e2, v).
hasVar(e, v) :- col = new set[string], collectVar(e,col), read(col) = v.
collectVar(e, col) :- ?Var(e,v), col += v.
collectVar(e, col) :- ?Plus(e,lhs,rhs),collectVar(lhs,col),collectVar(rhs,col).
```

Our extension comprises three new constructs. First, we can create distinct containers such as `new set[string]` in `hasVar`. Second, we can add elements to the container such as `col += v` in `collectVar`. Third, we can read from a container `read(col)`. When executing this program, only the container needs to be propagated but not the variables, leading to considerably fewer tuples. But, actually, our containers are much more flexible than this example may suggest.

Our extension supports a wide variety of "containers", which only need to adhere to a single monotonicity property for a user-defined partial order $\sqsubseteq$.

$$\forall c.\, \forall i.\ \mathsf{read}(c) \sqsubseteq \mathsf{read}(\mathsf{c} \mathrel{+}= \mathsf{i})$$

[1] Datalog requires range-restrictedness: Each variable in the head of a rule must be positively bound in the body of the rule. In this paper, example Datalog programs sometimes violate range-restrictedness when a relation's head variable is intended to receive input from the relation's call-sites. This is the case for head variable `col` of `collectVar`, which is bound by the call in `hasVar`. Our Datalog code can be systematically rewritten to satisfy range-restrictedness using the magic set transformation. We take the liberty of showing the "wrong" code for readability.

That is, adding elements i to a container c will only increase the subsequent reads. This property is sufficient for ensuring that containers are compatible with Datalog's fixed-point semantics. For the `set` container from above, the monotonicity property holds because `read` produces the set of added elements. But we do not need to collect elements, we may also aggregate them. For example, a sum aggregation of natural numbers satisfies the monotonicity property: `read` produces the sum, which grows with every addition. Since monotonicity is the only constraint, we call our container extension *mono types*. Mono types not only can improve performance, they also provide a new form of abstraction. For example, in our social-network program, we can add people with their friendship count to a mono type, and separately decide which mono-type definition to use:

```
collectFriend(p, col) :- col += (p, count(connectedWith(p,_))).
collectFriend (p1, col) :- connectedWith(p1, p2), collectFriend(p2, col).
mostPopFriend(p, pop) :- max = new retain_max, collectFriend(p, max), read(max) = pop.
leastPopFriend(p, pop) :- min = new retain_min, collectFriend(p, min), read(min) = pop.
```

We first demontrate the algebraic theory of mono types. A well-founded theory of mono types is crucial because violations of monotonicity will lead to undefined Datalog behavior. In particular, we develop mono-type constructions that enable us to develop complex mono types based on simpler ones. For example, a map mono `Map[K,V]` is parametric in a mono type for values `V` and combines all values that have the same key. This way we can construct mono types for multi maps `new map[K, Set[V]](new set)` or for grouped maxima `new map[K, int](new max)`. Our theory ensures that all map constructions yield valid mono types. We have formalized the algebraic theory of mono types and verified the correctness of mono-type constructions in the proof assistant Rocq, and the code is available open-source.[1]

We then show how mono types can be integrated into Datalog systems. Technically, we compile mono types to Datalog aggregations. However, this induces a considerable overhead for mono types that are containers, such as sets and maps. Therefore, we develop the theory of mono-type equivalence and mono-type functors, allowing us to convert between equivalent mono-type definitions. We use this theory to prove certain optimizations sound, and add them to our Datalog integration. Finally, we evaluate the programmability and performance of mono types using case studies. In summary, we present the following contributions:

- We develop and mechanize the algebraic theory of mono types and mono-type constructions in Rocq.
- We integrate mono types into Datalog by translation to user-defined aggregations.
- We formalize the theory of mono-type equivalence and mono-type functors and use it to develop sound compiler optimizations for the set and map mono types.
- We show that mono types improve Datalog's programmability and evaluate their performance.

2 A Theory of Mono Types

In the introduction, we have seen some motivating examples for using mono types: to collect values and to aggregate them. Since mono types can occur as part of Datalog's fixed-point semantics, it is important to ensure Datalog remains well-founded. To this end, we develop and formalize a specification for mono types and instantiate the specification with various examples. We then present generic constructions for mono types, allowing us to build up

[1] `https://gitlab.rlp.net/plmz/artifacts/mono-types-ecoop25`

complex mono types from simpler ones. We have mechanized the theory of mono types and the constructions in Rocq and use Rocq's notation in our presentation.

2.1 Mono Types

Conceptually, mono types are stateful containers. However, mono-type containers are restricted in three important ways: First, elements can only be added to a mono type but never removed or replaced. Second, the content of a mono type can only be observed monotonically. And third, the observation of a mono type is independent with the order of added elements. We represent the interface of mono types as a type-level record in Rocq:

```
Record MonoAPI M I O `{PartialOrder O} : Type :=
{ create : M I O ;  add : M I O → I → M I O ;  read : M I O → O
; mono_law : forall m i, read m ⊑ read (add m i)
; ord_indep : forall m i1 i2, read (add (add m i1) i2) = read (add (add m i2) i1)
}.
```

Mono instances have type `M I O`, where `I` is the type of added inputs, and `O` is the type of the read observations. Mono instances can only be created through functions `create` and `add`, which means all mono instances are sequences:

$$\texttt{m}_0 = \texttt{create}, \qquad \texttt{m}_1 = \texttt{add m}_0\ \texttt{i}_1, \qquad \cdots \qquad \texttt{m}_n = \texttt{add m}_{n-1}\ \texttt{i}_n$$

Users can observe the state of a mono instance at any time using `read`, but with two restrictions:

(i) Reads on a later mono instance produce larger values than reads on an earlier mono instance from the same sequence:

$$\forall\, i, j.\quad i < j \implies \texttt{read m}_i \sqsubseteq \texttt{read m}_j$$

In the interface of mono types above, we ensure this property by requiring a monotonicity law `mono_law`. This law states that each individual `add` increases the observable output with respect to a fixed partial order for `O`.

(ii) Swapping the order of inputs does not affect the updated value of a mono instance:

$$\forall\, i_1, i_2.\quad \texttt{read}\ (\texttt{add}\ (\texttt{add}\ \texttt{m}_j\ i_1)\ i_2) = \texttt{read}\ (\texttt{add}\ (\texttt{add}\ \texttt{m}_j\ i_2)\ i_1)$$

We ensure the order independence through the law `ord_indep` in the interface.

The monotonicity law, order independence law, and the lack of any removal operations are the reason that mono types integrate well with Datalog as we will discuss below.

One notable omission from `MonoAPI` is the internal state of mono instances. While it is clear that mono types must use internal state to carry information from an `add` to a `read`, the internal state cannot be observed by users directly. Instead, the internal state is only accessible while defining a mono type, for which we provide a dedicated interface:

```
Record Mono ST I O `{PartialOrder O} : Type :=
{ init : ST ; add : ST → I → ST ; read : ST → O
; mono_law : forall (st : ST) (i : I), read st ⊑ read (add st i)
; ord_indep : forall s a1 a2, result (add (add s a1) a2) = result (add (add s a2) a1)
}.
```

In contrast to `MonoAPI`, interface `Mono` has an additional type parameter `ST`, which exposes the type of the internal state. Otherwise, the two interfaces are almost identical and each `Mono` gives rise to an implementation of `MonoAPI` by setting `M I O = ST` and using `init` for `create`. All

our examples and constructions use the `Mono` interface, but it is important to remember that the internal state will not be visible to users of `MonoAPI`, which means we can exchange the implementation internals of a mono type without affecting users.

As first example, consider a mono type `nat_sum_mono` that computes the sum over naturals:

```
Definition nat_sum_mono : Mono nat nat nat :=
  {| init := 0 ; add s n := s + n ; read s := s |}.
```

For `nat_sum`, the internal state, input, and output all are of type `nat`. We start with an initial state `0`, add each new input to it, and allow the sum to be observed directly. This mono type satisfies the `mono_law` for the natural order of `nat`.

Our second example is `retain_max_mono`, which we used in Section 1 for the most popular person in a social network:

```
Definition retain_max_mono A : Mono (option (A, nat)) (A, nat) (option (A, nat)) :=
{| init := None
 ; add s i := match s, i with
   | None, _ ⇒ Some i
   | Some (a1,n1), (a2,n2) ⇒ if n1 >? n2 then Some (a1,n1) else Some (a2,n2)
   end
 ; read s = s
|}.
```

This mono type is more interesting and requires distinct types for input and output. The input is a pair `(A,nat)`, which can be understood as a weighted `A`. The mono type retains that input `A` with the highest weight. However, initially no `A` value is known yet, which is why we employ an option type for the internal state and output. Mono type `retain_max` satisfies the `mono_law` for the ordering where `None` $\sqsubseteq$ `Some` (a, n) for all a and n, and `Some` $(a_1, n_1) \sqsubseteq$ `Some` (a_2, n_2) if $n_1 \leq n_2$.

In both previous examples, the `read` function was a simple identity. We include `read` in our design because it allows us to isolate a post-processing step. For example, we can define a set mono that collects and produces a set of all inputs. However, internally this mono type uses a list representation for constant-time `add` and only converts the list to a set in `read`:

```
Definition set_mono1 A : Mono (list A) A (set A) :=
{| init := nil ; add s a := a :: s ; read s := set_from_list s |}.
```

The internal state has no obvious ordering, but the output set satisfies the `mono_law` based on the partial order ($\subseteq$) of set inclusion. Technically, it is not necessary to isolate the `read` projection, but it makes mono types easier to define and reason about. In principle, we can merge the post-processing done by `read` into the other functions if we use the internal state to also retain the post-processed output:

```
Definition set_mono2 A : Mono (list A, set A) A (set A) :=
{| init := (nil, set_from_list nil)
 ; add st a := let (l,_) := st in (a :: l, set_from_list (a :: l))
 ; read st := let (_,s) := st in s
|}.
```

Here, the internal state contains both the list of inputs and the projected set representation. We update the projected set after each `add`, but only use it in `read`. For this particular mono type, we can actually improve the definition by fusing `add` and `set_from_list`:

```
Definition set_mono A : Mono (set A) A (set A) :=
  {| init := empty_set ; add s a := set_add a s ; read s := s |}.
```

This is our final mono type for sets (*set mono* for short). We start with an empty set and add input elements to that set directly. While we could have started with this mono type, note this: All three mono types implement the same `MonoAPI`. Semantically, the three mono types are interchangeable because they only differ in their internal state, which is not observable. We will exploit such interchangeability in Section 4 to optimize usages of mono types.

The set mono is actually a special case of an entire class of mono types. A bounded semi-lattice has a bottom element and a least upper bound operator `lub`. Each bounded semi-lattice uniquely determines a partial order:

$$\mathtt{a}_1 \sqsubseteq \mathtt{a}_2 \iff \mathtt{lub}\ \mathtt{a}_1\ \mathtt{a}_2 = \mathtt{a}_2$$

Each bounded semi-lattice gives rise to mono type, which satisfies the `mono_law`:

```
Definition lattice_mono A `(BoundedSemiLattice A) : Mono A A A :=
  {| init := bottom ; add := lub ; read a := a |}.
```

This is an important class of mono types because lattices are heavily exercised in Datalog. In particular, lattice-based aggregation [8, 13] allows Datalog programs to implement data-flow analyses over custom lattices. The lattice mono can also easily accomodate widening, for example, using a unary widening operator in `read`.

Lastly, we want to provide an example where the internal state of a mono type is not monotone, but the output is. Consider the following mono type where the internal state changes in accordance with the collatz function:

```
Definition collatz_fun n : {n : nat | 0 < n} := if is_even n then n / 2 else 3 * n + 1.
Definition collatz_mono A (n : {n : nat | 0 < n}) : Mono nat A bool :=
{| init := n ; add st _ := if st =? 1 then 1 else collatz_fun st ; read st := st =? 1 |}.
```

The `collatz_mono` starts at a given positive number `n`. At each call of `add`, we forward the state to the next collatz number, unless we already reached `1`. The output of the `collatz_mono` is `true` if the number of calls to `add` is equal to or exceeds the total stopping time of `n`, that is, if the sequence has reached `1` yet. For example, `collatz_mono 12` will exercise the following sequence for the first 9 calls of `add`: $12, 6, 3, 10, 5, 16, 8, 4, 2, 1$. Any subsequent call of `add` will keep the state at 1. Nonetheless, `add` is not monotone: the state fluctuates up and down indeterminately. However, the output of mono type satisfies `mono_law` for the order `false` $\sqsubseteq$ `true`.

2.2 Mono-Type Constructions

We have seen a number of mono types in the previous subsection. However, it is not necessary to construct mono types individually, including the necessary proof of `mono_law` . Instead, users can use generic constructions that we discovered for creating complex mono types from simpler ones. We start with constructions that compose multiple mono types.

Sum mono. In Section 1, we used a mono type to collect all variables in an expression tree. Consider we also want to collect the integer literals the tree contains. Of course, we could traverse the tree twice, once collecting variables and once collecting literals, but this entails a memory and run-time overhead. It is more efficient to use a single mono type that can collect both kinds of data. We can construct such a mono type using the sum construction:[2]

[2] Note the record accessors in Rocq: `init m1`, `add m1`, and `read m1` refer to the respective fields of the mono-type record `m1`.

```
Definition sum_mono (m1 : Mono ST1 I1 O1) (m2 : Mono ST2 I2 O2)
  : Mono (ST1 * ST2) (I1 + I2) (O1 * O2) :=
{| init := (init m1, init m2)
 ; add s a := let (s1, s2) := s in match a with
              | inl a1  ⇒  (add m1 s1 a1, s2)
              | inr a2  ⇒  (s1, add m2 s2 a2)
              end
 ; read s := let (s1, s2) := s in (read m1 s1, read m2 s2)
|}.
```

A sum mono is parametric in mono types `m1` and `m2`, and it retains both of their internal states `ST1 * ST2`. The sum mono's input is the sum type `I1 + I2` and the `add` function dispatches to either `m1` and `m2`. We can use the sum mono in Datalog as follows:

```
main(e) :- col = new sum(new set[string], new set[int]), collectInfo(e, col).
collectInfo(e,col) :- ?Var(e,v), col += (inl v).
collectInfo(e,col) :- ?Num(e,n), col += (inr n).
collectInfo(e,col) :- ?Plus(e,lhs,rhs),collectInfo(lhs,col),collectInfo(rhs,col).
```

Product mono. While the sum mono dispatches its input, a product mono obtains inputs for both contained mono types. For example, in a social network, we may want to find the most popular person while also collecting all people in a set. The product mono `new prod(new retain_max, new set[Person])` can deliver this behavior:

```
Definition prod_mono (m1 : Mono ST1 I1 O1) (m2 : Mono ST2 I2 O2)
  : Mono (ST1 * ST2) (I1 * I2) (O1 * O2) :=
{| init := (init m1, init m2)
 ; add s a := let (s1, s2) := s in let (a1, a2) := a in (add m1 s1 a1, add m2 s2 a2)
 ; read s := let (s1, s2) := s in (read m1 s1, read m2 s2)
|}.
```

Note how each input pair `(a1,a2)` is split up and forwarded to `m1` and `m2` in `add`. The definitions `init` and `read` are identical to the sum mono.

Input functors. We can also construct various mono types through functors. The simplest class of functors only affects the input of a mono type:

```
Definition fmap_input_mono (f : I2 → I1) (m : Mono ST I1 O) : Mono ST I2 O :=
  {| init := init m ; add s a := add m s (f a) ; read s := read m s |}.
```

This functor wraps a mono type and applies `f` to all inputs prior to processing. For example, we can use the functor to define mono types that fork their input:

```
Definition fork_mono (m1: Mono ST1 I O1) (m2: Mono ST2 I O2): Mono (ST1 * ST2) I (O1 * O2)
  := fmap_input_mono (fun a ⇒ (a,a)) (prod_mono m1 m2).
```

The fork mono is like a product mono, but both contained mono types expect the same input type `I`. We can use an input functor to duplicate a given input and use the standard product mono to do the composition.

Map mono. Finite maps are used ubiquitously in computing, and program analysis applications in Datalog are no exception. For example, even a flow-insensitive analysis usually tracks abstract values for each variable `map[string,V]`, where `V` forms a lattice and multiple entries for the same variable are joined. A flow-sensitive analysis computes such a map for each control-flow node `map[Node,map[string,V]]`. To support this important class of Datalog applications, we developed a construction for map monos:

A map mono accepts input pairs `K * V`, where `K` is a key and `V` is a value. However, it does not store the values for keys directly. Instead, all values associated with the same key are added to a mono type `m` with input `V`. The map mono then stores the state of `m` for each key:

```
Definition map_mono `{Bounded O} (m: Mono ST V O): Mono (map K ST) (K * V) (map K O) :=
{| init := empty_map
 ; add s ka := let (k, a) := ka in
               map_add_with s k (fun st ⇒ add m st a) (add m (init m) a)
 ; read s := map_map_values (read m) s
|}.
Definition map_add_with {K V} : map K V → K → (V → V) → V → map K V.
Definition map_map_values {K V1 V2} : (V1 → V2) → map K V1 → map K V2.
```

Initially, the state of a map mono is the empty map; we initialize the state for new keys on demand in `add`. To this end, `add` uses a helper function `map_add_with s k f v` that adds key `k` to map `s`. If the key already exists in `s`, `map_add_with` uses `f` to update the old value using `add` of `m`. But if the key is new `s`, `map_add_with` inserts its value `v`. This is where `add` lazily initializes the entry for `k` with `init m` before adding `a` to it. The result of the map mono is another map, which contains the outputs of `m` for each key in the map.

To prove that `map_mono` produces valid mono types that satisfy the `mono_law`, we need to introduce a partial order on maps. The constraint `Bounded V` means `V` has a partial order and a bottom element.

```
Instance map_order {K V} `{Bounded V}: PartialOrder (map K V) :=
  { s1 ⊑ s2 := forall k, map_find_default k bottom s1 ⊑ map_find_default k bottom s2 }.
```

Function `map_find_default k v s` yields the value of `k` in `s`, or `v` if `s` does not contain `k`. This means we compare two maps by treating all undefined keys as being assigned to bottom. Thus, `s1 ⊑ s2` holds if for all keys `k` in `s1`, either `s1(k) = bottom` or `s2` also contains `k` and `s1(k) ⊑ s2(k)`. Based on this order, we prove `mono_law` for `map_mono` by induction on its internal state. We can now use the map mono to construct various useful mono types, such as:

```
Definition multi_map_mono K V := map_mono K (set_mono V).
Definition var_sign_mono := map_mono string (lattice_mono sign).
Definition stmt_var_interval_mono :=
  map_mono stmt (map_mono string (lattice_mono interval)).
```

The `var_sign_mono` can be used for a flow-insensitive sign analysis, tracking the sign of each variable. The `stmt_var_interval_mono` nests two map monos to track the range of each variable for each statement separately. These mono types can improve the programmability of Datalog.

3 Integrating Mono Types into Datalog

The theory from the previous section establishes that reading from a mono type yields monotonically growing observations about its state. In the present section, we explain how this property enables us to integrate mono types into Datalog. Syntactically, the integration is straightforward as shown in Figure 1 (you can ignore the parts in blue for now). This follows our examples from the paper: We add an atom $t \mathrel{+}= t$ to add a value to a mono type, **new** M to create a new mono instance, and **read**(t) to read the result of a mono type. The present section explains how these constructs can be integrated into Datalog semantically.

Our strategy for integrating mono types into Datalog is to treat them as a generalized form of aggregation. In Datalog, aggregation is often written as an annotation on the rule head:

$$
\begin{array}{rcll}
p & \in & \mathit{Prog} & ::= \overline{r} \\
r & \in & \mathit{Rule} & ::= P(\overline{t}) \coloneq \overline{a}. \mid P(\overline{t}, \alpha(\overline{t})) \coloneq \overline{a}. \\
a & \in & \mathit{Atom} & ::= P(\overline{t}) \mid t = t \mid t \mathrel{+}= t@\overline{t} \\
t & \in & \mathit{Term} & ::= c \mid x \mid \textbf{new}\ M(\overline{m})@\overline{T} \mid \textbf{read}(t) \\
m & \in & \mathit{MonoTerm} & ::= \textbf{new}\ M(\overline{m}) \mid c \\
\multicolumn{4}{c}{c \in \textbf{Val} \quad x \in \textbf{Var} \quad M \in \textbf{MonoType} \quad \alpha \in \textbf{AggOp}}
\end{array}
$$

Figure 1 Datalog syntax extended with constructs for creating, adding to, and reading from a mono type.

```
R(x, α(y)) :- P(x, y).
P("a",1). P("a",2). P("b",1). P("b",3). P("c",10).
```

Here, α is an aggregation operator, `α(y)` is the aggregation results, and all bindings for `y` in the body of `R` are the aggregation inputs. Note that we obtain a separate aggregation result for each `x` in `R(x,α(y))`. For example, if α implements a sum aggregation, we obtain:

```
R("a",3). R("b",4). R("c",10).
```

This form of aggregation is prevalent in existing Datalog systems and we want to reuse it for mono types. However, mono types are first-class containers, and aggregation happens within each container. To embed mono types in Datalog, we have to solve 3 conceptual challenges first:

- The above example uses strings to distinguish aggregation results. Mono instances also need to be distinguishable, so that each one can provide their own aggregation result.
- The above example collects all aggregation inputs in a relation `P`. Mono instances also need to collect their inputs, even though they are provided through scattered calls of `add`.
- The above example uses a simple aggregation operator $\alpha : 2^{\texttt{int}} \to \texttt{Y}$ that takes a set of input elements of type `int` and yields a value of some type `Y`. Mono types use `add` and `read` functions to produce aggregation results, which does not fit the signature of α.

In the remainder of this section, we describe these challenges in detail and propose concrete solutions. In the end, we present translation rules that compile mono types to Datalog and give a declarative semantics for mono types.

3.1 Representing Mono Instances

Each instance of a mono type represents a distinct container. Adding an element to one mono instance may not affect other mono instances. That is, mono instances need to be identifiable and distinguishable. Consider the following example, reproduced from Section 1:

```
sharedVar(v) :- e1 = Plus(Plus(x1,x2),y), e2 = Plus(y,Plus(z1,z2)), hasVar(e1,v), hasVar(
    e2,v).
hasVar(e, v) :- m = new set[int], collectVar(e,m), read(m) = v.
collectVar(e, m) :- ?Var(e,v), m += v.
collectVar(e, m) :- ?Plus(e,lhs,rhs), collectVar(lhs,m), collectVar(rhs,m).
```

In our example, `sharedVar` invokes `hasVar` twice, once for `e1` and once for `e2`. Accordingly, `hasVar` must create two instances with `new set[int]`, one for each expression. In contrast to object-oriented programming languages, Datalog does not have side-effects, which is necessary for creating unique object IDs. We need a different approach.

Our solution is based on an observation: Predicate `hasVar` creates a distinct mono instance for each expression `e` provided as `input`. By default, the atoms that occur before `new set[int]`

in the body of `hasVar` determine how many mono instances we need: one for each unique binding of variable `e`. In general, if a rule $P(\bar{t}) :- a_1, \ldots, a_n.$ has an atom a_i that contains **new** M, then we need a distinct instance of M for each substitution produced by $a_1, \ldots, a_{i-1}$. For each substitution $\sigma = \{X_1 \mapsto c_1, \ldots, X_k \mapsto c_k\}$ produced by $a_1, \ldots, a_{i-1}$, we uniquely represent the associated mono instance as a tuple $(c_1, \ldots, c_n)$. We also support a more explicit form **for** $\bar{t}$: **new** $M(\overline{m})$, which lists relevant variables explicitly and does not rely on the order of atoms in the rule. Note that nested mono constructors such as the two sets in `new sum(new set[string], new set[int])` are arguments to the surrounding `sum` mono; they are not values in Datalog and do not require a Datalog representation.

3.2 Collecting Mono-Type Inputs

Datalog aggregation is defined over relations: A relation collects all relevant aggregands to be processed by an aggregation operator. In contrast, the aggregands of mono types are scattered throughout the program: the invocations of `+=` determine the relevant aggregands, but there is no central relation that collects them. For example, consider again the following program from Section 2, where we add values to a mono type `m` in two different places:

```
main(e) :- m = new sum(new set[string], new set[int]), collectInfo(e, m).
collectInfo(e,m) :- ?Var(e,v), m += (inl v).
collectInfo(e,m) :- ?Num(e,n), m += (inr n).
collectInfo(e,m) :- ?Plus(e,lhs,rhs),collectInfo(lhs,m),collectInfo(rhs,m).
```

To enable aggregation for mono types, we must collect the aggregands in a relation. We found that we can use a magic set transformation to achieve this. A magic set transformation rewrites the Datalog program to collect the relevant arguments of a relation [16]. If we treat `m += a` as a call to a synthesized relation `Aggregand(m, a)`, we can use a magic set transformation to obtain the following rules:

```
Aggregand(m, a) :- ?Var(e,v), a = inl v.
Aggregand(m, a) :- ?Num(e,n), a = inr n.
```

These rules are derived from the previous program by copying the relevant atoms for each `m += a`. This way, the `Aggregand` relation enumerates the added aggregand values `a` for each mono instance `m`.

By collecting all added values in a relation, we implicitly submit to Datalog's set semantics. Specifically, even when executing `m += a` multiple times, relation `Aggregand` will only enumerate `(m,a)` $\in$ `Aggregand` at most once; multiple occurrences of the same tuple are collapsed. While this is fine for the set monos used in the example above, other mono types are less permissive. For example, consider we want to infer how often each variable occurs in an expression:

```
main(e) :- m = new map[string,nat](new nat_sum), collectInfo(e, m).
collectInfo(e,m) :- ?Var(e,v), m += (v,1).
collectInfo(e,m) :- ?Plus(e,lhs,rhs),collectInfo(lhs,m),collectInfo(rhs,m).
```

Even if an expression contains the same variable or numeric literal multiple times $x + y + x$, the `Aggregand` relation will only contain two tuples: `(m, ("x",1))` and `(m,("y", 1))`. That is, the second insertion of `(m, ("x",1))` does not show up, because Datalog employs its set semantics for the relation. As a consequence, we would incorrectly compute that both `"x"` and `"y"` only occur once in the expression.

This issue is well-known for aggregations in Datalog, and we adapt the standard solution [9]. To distinguish different occurrences of aggregands, we introduce auxiliary columns that help distinguish them. Figure 1 showed the necessary syntactic constructs in blue font. For our example, we can use the IDs of expressions for this purpose:

```
main(e) :- m = new map[string,nat](new nat_sum)@Exp, collectInfo(e, m).
collectInfo(e,m) :- ?Var(e,v), m += (v,1)@e.
collectInfo(e,m) :- ?Plus(e,lhs,rhs),collectInfo(lhs,m),collectInfo(rhs,m).
Aggregand(m, kv, id) :- ?Var(e,v), kv = (v,1), id = e.id.
```

Note how we augmented the mono construction with the type and the `add` call with the value of the auxiliary columns. For $x + y + x$ we obtain the following aggregands, assuming each variable has a distinct ID `Var#i`.

```
Aggregand(m, ("x",1), Var#1). Aggregand(m, ("y",1), Var#2). Aggregand(m, ("x",1), Var#3).
```

Importantly, aggregation will ignore the auxiliary columns. It will query `Aggregand(m, kv, _)` and call `add` for each individual entry. The next subsection explains how aggregation works in detail.

3.3 Mono Types as Generalized Recursive Aggregation

Conceptually, Datalog's aggregation operators are defined over sets of inputs and have the signature $\alpha : 2^{\mathtt{X}} \to \mathtt{Y}$. That is, given a set of inputs of type X, aggregation yields a single output of type Y. In practice, Datalog systems compute aggregation results iteratively and require aggregation operators of the following form: $\beta : \mathtt{Y} \times (\mathtt{X} \to \mathtt{Y} \to \mathtt{Y})$. An aggregation operator β defines an initial result β_1 that corresponds to $\alpha(\emptyset)$ and an update function β_2 that incorporates another input. Then, for $X = \{x_1, \ldots, x_n\}$, we have $\alpha(X) = \beta_2(x_n, (\ldots, \beta_2(x_1, \beta_1)))$.

As a first try, we could integrate mono types as iterative aggregation operators $\beta = (\mathtt{init}, \mathtt{add})$. The `read` function is then only applied after obtaining the mono-type state through aggregation. This leads to code of the following form, where `x` are mono-type inputs, `y` are mono-type states, and `z` are mono-type outputs:

```
Aggregand(m, v) :- ... .
Aggregate(m, β(x)) :- Aggregand(m, x).
P(z) :- Aggregate(m,y), read(y) = z.
```

But this approach faces a semantic issue: it does not work for recursive aggregation.

If recursive aggregation occurs the Datalog program (indirectly) feeds the output y of an aggregation back as input x. Many interesting use cases of aggregation require recursive aggregation, including program analysis and network analysis. For example, consider again the social-network analysis from Section 1, which finds the most popular friend:

```
mostPopFriend(p, max((p, n))) :- n = count(connectedWith(p,_)).
mostPopFriend(p1, max((p, n))) :- connectedWith(p1, p2), mostPopFriend(p2, (p, n)).
main(p) :- mostPopFriend("Alice", (p, _)).
```

This program uses recursive aggregation, because the second rule obtains an aggregation output `(p,n)` of `p2` and uses it as aggregation input for `p1`. Recursive aggregation is only well-defined if the aggregation operator is monotone: $\forall X.\, \forall Y.\, \alpha(X) \sqsubseteq \alpha(X \cup Y)$ for traditional aggregation, or $\forall x.\, \forall y.\, y \sqsubseteq \beta_2(x, y)$ for iterative aggregation.[3] Monotonicity of aggregation is important because new aggregation results *replace* prior aggregation results (in contrast to regular columns, which accumulate all derived values in a set). But our mono types do not

[3] Note that the initial result β_1 does not need to be a least element because Datalog yields *no result* when there are no aggregands. Internally, Datalog embeds aggregations results in `option` Y and yields `None` initially, but `Some` y subsequently.

guarantee monotonicity for the states aggregated by `add`; we only ensure monotonicity for the resulting outputs. This renders the above approach incorrect because the second column of `Aggregate(m,`β`(x))` does not grow monotonically. Consequently, we lose the property that every Datalog program has a minimal model, and a fixed-point does not need to exist.

We can fix the situation by considering mono types as generalized aggregation operators

$$\gamma : \mathtt{Y} \times (\mathtt{X} \to \mathtt{Y} \to \mathtt{Y}) \times (\mathtt{Y} \to \mathtt{Z}).$$

This directly corresponds to the interface of mono types, with $\gamma = (\mathtt{init}, \mathtt{add}, \mathtt{read})$ and extends iterative aggregation by incorporating the output projection `read`. While this may seem negligible, it significantly alters the semantics of Datalog programs using such aggregation:

```
Aggregand(m, v) :- ... .
Aggregate(m, γ(x)) :- Aggregand(m, x).
P(z) :- Aggregate(m,z).
```

Compared to the first approach, `Aggregate` now contains output values `z` rather than internal states `y`. And since mono types guarantee monotonicity of the projected outputs through the `mono_law`, we can use them as a generalized form of aggregation. For example, this generalized aggregation is necessary to correctly embed the `collatz_mono` into Datalog, because its output is monotone even though its state fluctuates. We have used generalized aggregations in an existing Datalog system (see Section 5) and it was easy to replace β-aggregation by γ-aggregation.

3.4 Formal Translation and Mono Dispatch

The previous subsections explained the ideas for encoding mono types in Datalog. To make these encodings precise and reproducible, we provide a formal translation from Datalog with mono types to Datalog with aggregation. Figure 2 shows the translation rules, which we explain one by one subsequently. The reason we define the semantics of mono types through translation to recursive aggregation is that the semantics of recursive aggregation is well understood [9] and supported by modern Datalog engines.

A program consists of rules $r_1 \cdots r_n$, which we translate compositionally $[\![r_1]\!] \cdots [\![r_n]\!]$. However, to support mono types, we must also generate additional rules for each mono construction **new** $M(\overline{m})@\overline{U}$ in the program. Note that $\overline{U}$ are the types of the auxiliary columns used to distinguish different occurrences of added inputs as described in Subsection 3.2. For each mono construction, we derive three new rules:

- Aggregand: Collects all aggregands of a mono type, added through invocations of $m \mathrel{+}= t$. As explained in Subsection 3.2, we rely on a magic-set transformation to synthesize the rules for this predicate; our translation only indicates the head of these rules.
- $\mathsf{Aggregate}_{M(\overline{m})}$: Performs the aggregation for the fully instantiated mono type $M(\overline{m})$ in the head of the rule. The body of the rule draws the aggregation inputs from Aggregand, ignoring the auxiliary columns.
- Aggregate: Since different mono types can occur in the same program, we need to carefully select the correct aggregation operation (an issue we have ignored so far). To this end, we use Aggregate to *dispatch* over the current mono type. Specifically, if m is a mono instance of $M(\overline{m})$, we dispatch to $\mathsf{Aggregate}_{M(\overline{m})}$. Note that these dispatch alternatives must be generated statically to identify the predicate named $\mathsf{Aggregate}_{M(\overline{m})}$.

The rest of the translation is compositional; Figure 2 only shows the rules for rewritten atoms. First, consider the construction of a new mono instance of $M(\overline{m})$. We represent

$$
\begin{array}{lcl}
[\![r_1 \cdots r_n]\!]_{prog} & = & [\![r_1]\!]_{rule} \cdots [\![r_n]\!]_{rule} \cdot [\![\textbf{new } M(\overline{m})@\overline{T} \in (r_1 \cdots r_n)]\!]_{agg} \\
\\
[\![\textbf{new } M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} & = & \mathsf{Aggregand}(m, t, (u_1, \ldots, u_k)) :- \ \textit{by magic set trans.} \\
 & & \mathsf{Aggregate}_{M(\overline{m})}(m, M(\overline{m})(t)) :- \mathsf{Aggregand}(m, t, \_). \\
 & & \mathsf{Aggregate}(m, t) :- \ ?\mathsf{MonoVal}(m, M(\overline{m}), \_), \\
 & & \qquad\qquad \mathsf{Aggregate}_{M(\overline{m})}(m, t). \\
\\
[\![a_0 :- a_1, \ldots, a_n]\!]_{rule} & = & a_0 :- [\![a_1]\!]_{atom}, \ldots, [\![a_n]\!]_{atom} \\
\\
[\![m = \textbf{for } \overline{t} : \textbf{new } M(\overline{m})@\overline{U}]\!]_{atom} & = & m = !\mathsf{MonoVal}(M(\overline{m}), \overline{t}) \\
[\![m \mathrel{+}= t@\overline{u}]\!]_{atom} & = & \mathsf{Aggregand}(m, t, \overline{u}) \\
[\![x = \textbf{read}(m)]\!]_{atom} & = & \mathsf{Aggregate}(m, x)
\end{array}
$$

Figure 2 Formal translation of Datalog with mono types to Datalog with generalized aggregation.

mono instances as algebraic data in Datalog: $m = !\mathsf{MonoVal}(M(\overline{m}), \overline{t})$ constructs a mono instance and $?\mathsf{MonoVal}(m, M(\overline{m}), \overline{t})$ deconstructs m again. Each mono instance contains its fully instantiated mono type $M(\overline{m})$ and a list of values computed by $\overline{t}$ that uniquely identifies each mono instance as explained in Subsection 3.1.

An add operation $m \mathrel{+}= t@\overline{u}$ translates to a call of relation Aggregand. The magic set transformation will use this call to generate a rule for Aggregand that enumerates the arguments of the call. Here, the terms $\overline{u}$ serve to distinguish different occurrences of t. Finally, we read the aggregation result of a mono instance by calling relation Aggregate.

This concludes the integration of mono types into Datalog. In the translated program, the aggregation operator $M(\overline{m})$ is both order-independent and monotonic under bag inclusion, based on the algebraic properties of mono types. Thus, our integration does not jeopardize the least-fixed point semantics of Datalog with recursive aggregation [13], as long as there is no infinite ascending chain in the aggregation domain. We can use our translation to execute Datalog programs with mono types in any Datalog engine that supports generalized aggregation, such as Viatra/IncA [17], Ascent [10], and Flix [8].

3.5 An Example of Mono-type Translation

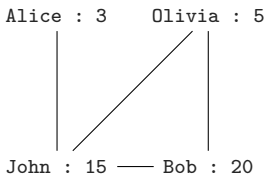
Even with above translation rules, understanding the execution of Datalog programs with mono types can be challenging, especially for readers who are unfamiliar with recursive aggregation. To aid in comprehension, we present a concrete example of the translation and the execution of a Datalog program with mono types.

Consider a program that determines the number of followers for the most-followed user (we call it *maximum follower count* in the follows) in Alice's social network. This program uses a map mono, which associates each user's name with a `nat_max` mono, to track the maximum follower count for each individual.

```
visit(m) :- followers(p, n), m += (p,n).
visit(m) :- connected(p, q), visit(q,m), map = read(m), m += (p,map(q)).
main(f) :- m = new map[string, nat](new nat_max), visit(m), f = read(m)("Alice").
```

Here is the translation of above program. For simplicity, we use ... to leave the body generated by magic set transformation, and inline the $\mathsf{Aggregate}_{M(\overline{m})}$ rule as there is only one mono-type instance created.

```
visit(m) :- followers(p, n), Aggregand(m,p,n).
visit(m) :- connected(p, q), visit(q,m), Aggregate(m,map), Aggregand(m,p,map(q)).
main(f) :- m = !MonoVal(map_mono(nat_max)), visit(m), Aggregate(m,d), f = d("Alice").
Aggregand(m,p,n) :- ...
```



(a) An instance of social-network.

	Aggregand(m,p,n)	Aggregate(m, d)
Iter. 1	(m, "Alice", 3), (m, "John", 15)*) (m, "Olivia", 5), (m, "Bob", 20)	(m, Map("Alice" -> 3, "John" -> 15, "Olivia" -> 5, "Bob" -> 20))
Iter. 2	(m, "Alice", 15), (m, "John", 20) (m, "John", 5), (m, "John", 3) (m, "Olivia", 15), (m, "Olivia", 20) (m, "Bob", 5), (m, "Bob", 15)	(m, Map("Alice" -> 15, "John" -> 20, "Olivia" -> 20, "Bob" -> 20))
Iter. 3	(m, "Alice", 20)	(m, Map("Alice" -> 20, "John" -> 20, "Olivia" -> 20, "Bob" -> 20))

*) We use `m` to represent the mono-type value `map_mono(nat_max)` for brevity.

(b) The execution trace of the translated program.

Figure 3 A concrete example of the execution of Datalog programs with mono types.

```
Aggregate(m,map_mono(nat_max)(p,n)) :- Aggregand(m,p,n).
```

We illustrate the execution of the translated program using a simple example in Figure 3. Figure 3a depicts a sample social network that includes Alice, where `Alice : 3` indicates that Alice has three followers. We detail the execution trace for both the `Aggregand` and `Aggregate` rules in Figure 3b. The aggregated argument of the `Aggregate` relation is a map that associates each user with the maximum follower count in her network. In the second iteration, John's maximum user count is updated by 20, which triggers the tuple `Aggregand(m, "Alice", 20)` derived in the third iteration. Besides, we can observe that the number of followers for each user in the map mono grows monotonically. We do not show the fourth iteration because no new tuples are derived and the program terminates. This example shows that performing mono add after mono read will lead to recursive aggregation in the translated program, but mono types abstract from the complications this involves.

4 Mono-Type Equivalence and Optimization

Mono-type aggregation can have a considerable performance overhead in Datalog systems because we are leaving the realm of Datalog's highly optimized relational algebra. In this section, we develop a theory for the equivalence of mono types that allows us to replace mono types without changing the semantics of a Datalog program. We then explain how this theory establishes optimizations for mono types and how they can reduce the need for aggregation in important cases.

4.1 Mono-type Equivalence

Intuitively, we consider two mono types to be equivalent if they produce the same outputs given the same sequence of inputs. There is no need to compare the states of mono types since they are invisible in the Datalog programs. For example, we introduced different version of the set mono in Section 2 (shown below): `set_mono1` uses a list internally and converts it to a set on `read`, while `set_mono` maintains a set of added elements internally and yields that set on `read`. These two mono types are equivalent because they produce the same sets as output, independent of states. However, sometimes we want to compare mono types that have different output types. For example, we would expect a multimap mono `map_mono K (set_mono V)` is equivalent to set of pairs `set_mono (K * V)`, because their output types `map K (set V)` and `set (K * V)` are isomorphic. Accordingly, we formalize the equivalence on mono types through a bidirectional embedding. A function f embeds mono type m_1 into m_2 if (i) f maps the initial output of m_1 to the initial output of m_2 and (ii) f is closed

under `add`. That is, instead of running m_2, we can run m_1 and embed its outputs to obtain results equivalent to m_2. Two mono types then are *equivalent* if there exist embeddings in both directions. We formalized mono-type equivalence in `mono_equiv` below and proved it is a reflexive, symmetric, and transitive relation in Rocq:

```
Definition reachable (m : Mono ST I O) (st : ST) :=
  exists (l : list I), List.fold_left (add m) l (init m) = st.
Definition mono_embed (m1 : Mono ST1 I O1) (m2 : Mono ST2 I O2) (f : O1 → O2) : Prop :=
  f (read m1 (init m1)) == read m2 (init m2) ∧
  forall (s1 : ST1) (i : I), reachable m1 s1 → exists (s2 : ST2),
    reachable m2 s2 ∧ f (read m1 s1) == read m2 s2 ∧
    f (read m1 (add m1 s1 i)) == read m2 (add m2 s2 i).
Definition mono_equiv (m1 : Mono ST1 I O1) (m2 : Mono ST2 I O2)
    (f : O1 → O2) (g : O2 → O1) : Prop := mono_embed m1 m2 f ∧ mono_embed m2 m1 g.
```

We showed `mono_equiv` holds for the multimap mono `map_mono K (set_mono V)` and `set_mono (K * V)`. To this end, we defined embedding functions for their outputs and verified they satisfy `mono_embed`.

4.2 Mono-type Functors

With mono-equivalence in place, we can now study how to transform between equivalent mono types. If we inspect the specification of mono types and their equivalence relation, we can find two significant ways to transform between equivalent mono types. First, we can post-process the result of `read` to produce an alternative output. And second, we can post-process the result of `add` to produce (and maintain) an alternative state. We call both kinds of transformations *mono-type functors* and provide sufficient criteria to ensure they preserve equivalence. This means that no manual equivalence proof will be necessary for mono types where one is derived from the other through mono-type functors.

We design output functors to change the output type and value of a mono type. To this end, we specify `OutputFunctor`, which employs a function `f` to adjust the `read` operation and leave `init` and `add` untouched. The result of `OutputFunctor` is a new mono type whose output changed from type `O1` to `O2`. In order to preserve the `mono_law`, we require `f` to be a monotone function.

```
Definition OutputFunctor (f : O1 → O2) (_ : monotone f) (m : Mono ST I O1)
  : Mono ST I O2 := {| init := init m ;  add := add m ; read s := f (read m s) |}.
```

For example, we can derive a flattened version of the multimap mono:

```
Definition flatten_multimap {A B} : map A (set B) → set (A * B).
Definition flat_multimap_mono : Mono (map K (set V)) (K * V) (set (K * V)) :=
  OutputFunctor flatten_multimap _ (map_mono K (set_mono V)).
```

Observe how `flat_multimap_mono` yields outputs of type `set (K * V)` rather than `map K (set V)`. This way, output functors allow us to create many interesting mono types by projection of the output. For example, we can use output functors to obtain a mono that only contains the unique keys of a multimap, or only those entries that satisfy a filter function.

```
Definition multimap_keys_mono : Mono (map K (set V)) (K * V) (set K) :=
  OutputFunctor map_keys _ (map_mono K (set_mono V)).
Definition multimap_size_mono : Mono (map K (set V)) (K * V) nat :=
  OutputFunctor set_size _ flat_multimap_mono.
```

Here, the last example shows that we can stack applications of `OutputFunctor`: We use a flattened multimap and take its size. This way, `multimap_size_mono` computes the number of key-value pairs.

But, when are the resulting mono types equivalent to the original ones? We have proved that, if `f` has an inverse function `g`, the mono type generated from an output functor is equivalent to the original mono type. That is, the effect of `OutputFunctor` must be reversible:

```
Theorem ob_functor_equiv : forall (m: Mono ST I O1) f (f_mono: monotone1 f) g,
    bijective f g → mono_equiv m (OutputFunctor f f_mono m) f g.
```

Not all uses of `OutputFunctor` satisfy this property. But, we can use this lemma to guarantee a multimap mono is equivalent to its flattened version:

```
Definition unflatten_multimap : set (K * V) → map K (set V).
Lemma bij_flatten_multimap : bijective flatten_multimap unflatten_multimap.
Lemma flat_multimap_equiv : mono_equiv (map_mono K (set_mono V)) flat_multimap_mono.
```

State functor. We formulate state functors to derive new mono types that operate on a modified state. Unlike output functors, state functors use a pair of functions `f` and `g` that convert between `ST1` and `ST2` and back. This is necessary because the state is both argument and result of `add`. While `add` operates on `ST1` in the original mono type, we aim to derive `add` that operates on `ST2`. To this end, we map the argument state from `ST2` to `ST1` to apply the original `add`, and then map its result back to `ST2`. To ensure the resulting definition satisfies the `mono_law`, we require function `f∘g` to be both extensive and increasing in domain `ST1`. Moreover, we impose another restriction `indep` to guarantee the transformed mono types meet the `ord_indep` law.

```
Definition extensive (f : ST1 → ST1) (m : Mono ST1 I O) :=
  forall s : ST1, result m s ⊑ result m (f s).
Definition increasing (f : ST1 → ST1) (m : Mono ST1 I O) :=
  forall s1 s2 : ST1, result m s1 ⊑ result m s2 → result m (f s1) ⊑ result m (f s2).
Definition ord_indep_abs (f : ST2 → I → ST2) (g : ST2 → O) :=
  forall s a1 a2, g (f (f s a1) a2) = g (f (f s a2) a1).
Definition StateFunctor (f : ST1 → ST2) (g : ST2 → ST1) (m : Mono ST1 I O)
  (_ : extensive (f ∘ g) m) (_ : increasing (f ∘ g) m)
  (indep : ord_indep_abs (fun s a ⇒ f (add m (g s) a)) (read m ∘ g) : Mono ST2 I O :=
  {| init := f (init m) ; add s a := f (add m (g s) a) ; read s := (read m ∘ g) s |}.
```

The state functor derives mono types that operate on different states internally. For example, we can transform the multimap mono to use flattened sets as states:

```
Definition multimap_flat_state_mono : Mono (set (K * V)) (K * V) (map K (set V)) :=
  StateFunctor flatten_multimap unflatten_multimap _ _ (map_mono K (set_mono V)) _.
```

Note how the derived mono type uses sets of pairs internally, although its output is still a multimap for now. But is the resulting mono type also equivalent to the original one? We have proved that, if `f` is a bijection and `g` is its inverse, `StateFunctor` preserves the equivalence of mono types:

```
Theorem st_functor_equiv {ST1 ST2 I O} `{PartialOrder O, PartialOrder ST1,
  PartialOrder ST2} : forall (m: Mono ST1 I O) (f : ST1 → ST2) (g : ST2 → ST1),
    bijective f g → mono_equiv m (StateFunctor f g _ _ m _) id id.
```

Composing functors. We can compose mono-type functors by stacking them. The order in which we compose multiple `StateFunctor` or multiple `OutputFunctor` instances matters, because one transforms the output of the other. We have seen an example of this in `multimap_size_mono`, which stacks two `OutputFunctor`: The first converts a multimap to a set of pairs, the second counts the number of entries. Consequently, the order of their application cannot be changed.

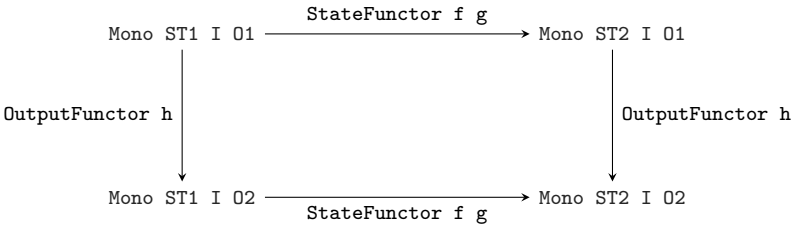


Figure 4 Commuting diagram for `OutputFunctor` and `StateFunctor` under `mono_equiv`.

The story is different when combining a `StateFunctor` with an `OutputFunctor`: One changes the state, the other changes the output. In that sense, we could try to apply them in any order. Indeed, we were able to prove that the order of their application is irrelevant for `mono_equiv`. Figure 4 shows the corresponding commuting diagram.

4.3 Optimization by Mono-Type Replacement

The equivalence relation `mono_equiv` indicates that we can use one mono type as a drop-in replacement for another mono type. Thus, if one implementation is more efficient than the other, we may opt for the better one. Mono-type functors help us establish such equivalences.

We illustrate the mono-type replacement for the multimap mono. So far, we have established a number of mono types that are provably equivalent to the multimap mono.

```
Definition flat_multimap_mono : Mono (map K (set V)) (K * V) (set (K * V)) :=
  OutputFunctor flatten_multimap _ (map_mono K (set_mono V)).
Definition multimap_flat_state_mono : Mono (set (K * V)) (K * V) (map K (set V)) :=
  StateFunctor flatten_multimap unflatten_multimap _ _ (map_mono K (set_mono V)) _.
Definition flat_multimap_flat_state_mono : Mono (set (K * V)) (K * V) (set (K * V)) :=
  OutputFunctor flatten_multimap _ multimap_flat_state_mono.
```

The last mono type shown above composes the state functor used in `multimap_flat_state_mono` with an output functor. The implementation of this mono type is not efficient, because it is converting back and forth between maps and sets all the time. But its signature suggests a way forward: it uses a set of pairs both as state and output. This is interesting, because we have seen another mono type that admits this type:

```
Definition set_pair_mono : Mono (set (K * V)) (K * V) (set (K * V)) := set_mono (K * V).
```

Since the state and output functor have done most of the heavy lifting, it is easy to prove that the set mono on pairs is equivalent to the last multimap mono from above:

```
Lemma set_multimap_equiv : mono_equiv flat_multimap_flat_state_mono set_pair_mono id id.
```

Hence, we can replace any multimap mono by a set mono in Datalog programs. We can follow the same approach to prove further equivalences and use them to opt for more efficient mono types.

4.4 Optimization by Mono-Type Elimination

This paper introduces mono types as a new abstraction for Datalog programmers. However, not all mono types are necessary at run time because *Datalog provides some mono types as built-ins*. We can eliminate those mono types that are built into Datalog from the program.

First, most Datalog systems support a number of primitive aggregations functions, such as `count`, `max`, or `sum`. Of course, we can reuse them instead of using `nat_sum`, for example. But this

is unlikely to affect the performance of the program much, because the built-in aggregations have to do the exact same work as our mono types and the aggregations resulting from them.

The second class of built-in mono types is much more important: Datalog has a number of relational mono types built-in. For example, Datalog relations essentially implement the `set_mono`: they collect tuples monotonically. If we can adopt such built-in relational mono types, we can eliminate the aggregation that the standard semantics of mono types imposes. Aggregation can have a considerable performance overhead in Datalog systems because we are leaving the realm of Datalog's highly optimized relational algebra. So this is likely to have a large performance impact.

Since the set mono is built into Datalog, we can optimize the translation rule:

if `mono_equiv` $M(\overline{m})$ (`set_mono` A) for some A:
$[\![\textbf{new}\ M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} = \mathsf{Aggregate}_{M(\overline{m})}(m, t) :\!-\ \mathsf{Aggregand}(m, t, \_).$
else:
$[\![\textbf{new}\ M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} = \mathsf{Aggregate}_{M(\overline{m})}(m, M(\overline{m})(t)) :\!-\ \mathsf{Aggregand}(m, t, \_).$

If $M(\overline{m})$ is equivalent to a set mono, the aggregation can be skipped: Note how the head of the generated rule simply forwards the aggregands. In contrast, other mono types have to aggregate the aggregands using $M(\overline{m})$. For example, we have established that the mono type for multimaps is equivalent to the mono type for sets. Therefore, multimaps do not require aggregation using the optimized translation rules.

We found another important relational mono type that is built into Datalog and which we call *functional relations*. A functional relation is a finite relation (i.e., set of tuples) with a functional dependency, meaning that some columns uniquely determine other columns. Datalog allows functional relations to aggregate the uniquely determined columns. Therefore, we can faithfully capture functional relations as the following mono type:

```
Definition fun_dep (s : set (A * B)) :=
  forall (a : A) (b b' : B), set_In (a, b) s → set_In (a, b') s → b = b'.
Definition funrel A B := {s : set (A * B) | fun_dep s}.
Definition funrel_mono K (m : Mono ST I O) : Mono (funrel K ST) (K * I) (funrel K O) :=
{| init := empty_set
 ; add s ka := let (k, a) := ka in
               funrel_add_with s k (fun st ⇒ add m st a) (add m (init m) a)
 ; read s := funrel_map_values (read m) s
|}.
```

Here, `funrel_add_with` uses the first function argument if `k` already is present in `s`, and the second the function argument for new `k`. Functional relations still require aggregation, but only on the uniquely determined columns rather than on all columns of the relation. We can exploit this in a dedicated translation rule:

if `mono_equiv` $M(\overline{m})$ (`funrel_mono` m') for some A:
$[\![\textbf{new}\ M(\overline{m})@(U_1, \ldots, U_k)]\!]_{agg} = \mathsf{Aggregate}_{M(\overline{m})}(m, k, m'(t)) :\!-\ \mathsf{Aggregand}(m, (k, t), \_).$

Note how the generated rule simply propagates the first half of the input k, and only requires aggregation of the remainder t. But this only holds for mono types $M(\overline{m})$ that are equivalent to a `funrel_mono`. We were able to establish such equivalence for the map mono:

```
Theorem map_funrel_mono_equiv : forall (m : Mono ST I O)
  mono_equiv (map_mono K m) (funrel_mono K m) map_funrel funrel_map.
```

We prove this property by first applying output and state functors to `map_mono`, such that the resulting mono type has the same signature as `funrel_mono`. The remainder of the proof is manageable and only took a few hours to complete.

In conclusion, Datalog programs that use a map mono only need to use aggregation on the values of that map. Moreover, since `map_mono K1 (map_mono K2 m)` is equivalent to `map_mono (K1 * K2) m`, actually only the most-nested value type has to be aggregated. This has a significant performance impact as we will evaluate in the subsequent section.

5 Implementation, Case Studies, and Performance

We have implemented mono types as an extension of the IncA Datalog system [14] and made the code available open-source.[2] IncA supports generalized recursive aggregations over user-defined functions, which makes it well-suited for mono types. In particular, we added built-in support for set monos, map monos, and a number of primitive monos on strings and numbers. We also provide an interface for user-defined mono types, which we used to integrate lattice-based mono types for example. We integrated mono types as described in Section 3 by translating into Datalog with aggregation. The translation for set monos and map monos is configurable to activate or deactivate the optimization described in Subsection 4.3 and 4.4. Overall, the integration of mono types required 550 LoC for IncA.

Next, we evaluate the usability and performance of mono types using two case studies: dependency analysis and interval analysis, each with and without using mono types. Program analysis is a common use case for Datalog because of its least fixpoint semantics. We show that mono types significantly improve the performance of these analysis compared to pure Datalog code.

5.1 Dependency Analysis

We want to analyze the dependencies of a simple language with inter-dependent definitions. A program of the language consists of definitions that assign a name to an expression. Expressions are either numbers, additions, or variables:

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

We represent the analyzed program as an abstract syntax tree (AST) which is encoded with algebraic data types in Datalog. Each variable refers to exactly one definition, as illustrated with subscripts in the following example:

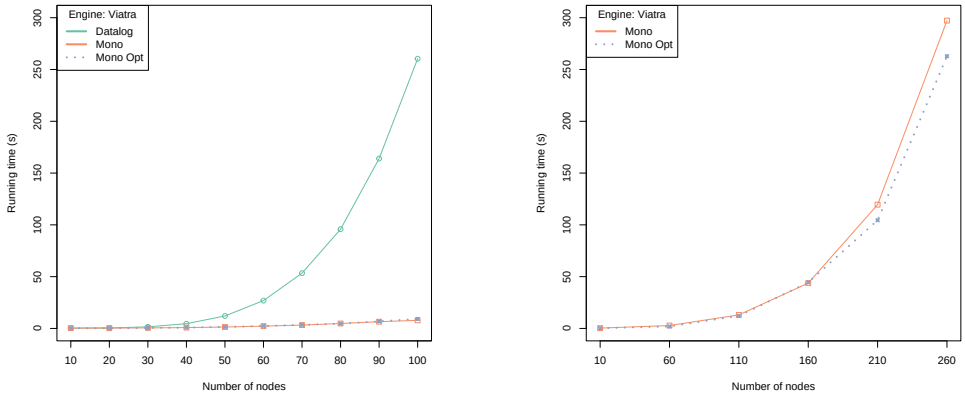
```
def a1 = c3; def b2 = 2; def c3 = a1; def main4 = c3
```

For a given definition, we compute all definitions it transitively depends on. In our example, $\texttt{main}_4$, $\texttt{a}_1$, and $\texttt{c}_3$ all transitively depend on $\texttt{a}_1$ and $\texttt{c}_3$, whereas $\texttt{b}_2$ is not used.

We implement this analysis in pure Datalog using three central relations:

```
edgesDefs(defs, from, to) :- ?Cons(defs, hd, tl), edgesDef(defs, hd, from, to).
edgesDefs(defs, from, to) :- ?Cons(defs, hd, tl), edgesDefs(tl, from, to).
edgesDef(defs, def, from, to) :- ?Def(def, _, e), refersTo(defs, e, to), from = def.
edgesDef(defs, def, from, to) :- ?Def(def, _, e), refersTo(defs, e, other),
                                 edgesDef(defs, other, from, to).
refersTo(defs, e, def) :- ?Var(e, name), findDef(defs, name, def).
refersTo(defs, e, def) :- ?Add(e, e1, e2), refersTo(defs, e1, def).
refersTo(defs, e, def) :- ?Add(e, e1, e2), refersTo(defs, e2, def).
```

[2] https://gitlab.rlp.net/plmz/inca-scala/-/tree/layered-ir/inca-ir/src/main/scala/inca/ir/extension/mono



(a) Viatra: Pure Datalog vs set mono.

(b) Viatra: Set mono vs optimized set mono.

Figure 5 Performance results for the dependency analysis.

We first traverse the list of all definitions in `edgesDefs`. For each definition, we compute all definitions it depends on using the two rules of `edgesDef`: A definition `def` depends on another definition `to` if (i) `def` contains a variable referencing `to`, or (ii) `def` contains a variable reference to `other`, which in turn depends on `to`. An expression `e` refers to a definition `def` if `e` contains a variable with the name of `def`. We accomplish this using the `refersTo` relation to recursively traverses an expression.

We now rewrite this program with mono types. Instead of collecting and propagating edges in `edgesDefs` and `edgesDef`, we create a single mutable mono set and pass it around. That way, `edgesDef` can directly write reachable pairs of definitions into the mono set when they are discovered:

```
main(from, to) :- m = new set[(Def, Def)], edgesDefs(defs, m), read(m) = (from, to).
edgesDefs(defs, m) :- ?Cons(defs, hd, tl), edgesDef(defs, hd, m), edgesDefs(tl, m).
edgesDef(defs, def, m) :- ?Def(def, _, e), refersTo(defs, e, trg),
                          m += (def, trg), edgesDef(defs, trg, m).
```

The changes required to the original dependency analysis are minimal. And the program became simpler. For example, we now only need a single rule for `edgesDef` that registers a dependency edge `(def, trg)` and also recurs. Similarly, `edgesDefs` only needs one rule that asks for the dependencies of `hd` and recurs on `tl` at once. But, most importantly for this case study, mono types improved the performance significantly.

In Figure 5, we show the execution time for the dependency analysis for different sized cyclic graphs on Viatra [17]. Figure 5a reveals that using a set mono reduces the running time considerably compared to pure Datalog. This is because the analysis using pure Datalog code has to propagate edges in `edgesDefs` and `edgesDef`, which produces orders of magnitudes more tuples than the analysis using mono types. Using a mono set eliminates this overhead. To highlight the difference between the optimized (Mono opt) and non-optimized set mono (Mono), we ran additional tests with bigger input graphs. As shown in Figure 5b, the optimization of set monos yields around 10% performance gains. For graphs exceeding 200 nodes this difference becomes more evident. The optimization does not have a bigger impact because, internally, the state of the non-optimized set mono is represented by an immutable Scala set. The aggregation operation needs to add elements to this internal state, which is a constant time operation. Since we are only reading the set mono once, in the `main` relation, this aggregation only happens a single time. We have additionally translated the dependency

analysis program into Soufflé [5] and Ascent [11]. Since neither of them supports user-defined aggregation, we can only compare the performance between plain Datalog and optimized set mono on these two engines. We found that the experimental results are almost identical to Viarta, optimized set mono's running time is significantly less than plain Datalog. This experiment result confirms mono types are an effective and general optimization technique for Datalog.

5.2 Interval Analysis

We also implement a flow-sensitive interval analysis for the While language in Datalog as idiomatically as possible. We then rewrite it to use nested map monos. Both implementations share common relations to compute the control flow graph and abstractly evaluate expressions.

Based on the control flow graph, the pure Datalog version uses two mutually recursive relations `intervalBefore` and `intervalAfter`. These relations define the interval value for a variable before and after the execution of a specific statement. To compute `intervalAfter`, we distinguish two different cases. If the current statement is an assignment of the current variable `var`, then `aeval` computes the value of `var` after the assignment. In all other cases, we propagate the information from `intervalBefore`. Note how `aeval` requires a reference to the current statement so that it can query `intervalBefore` for the value of referenced variables in `exp`. To compute `intervalBefore` for a variable `var`, we join the intervals of `var` that are valid after any predcessor statement.

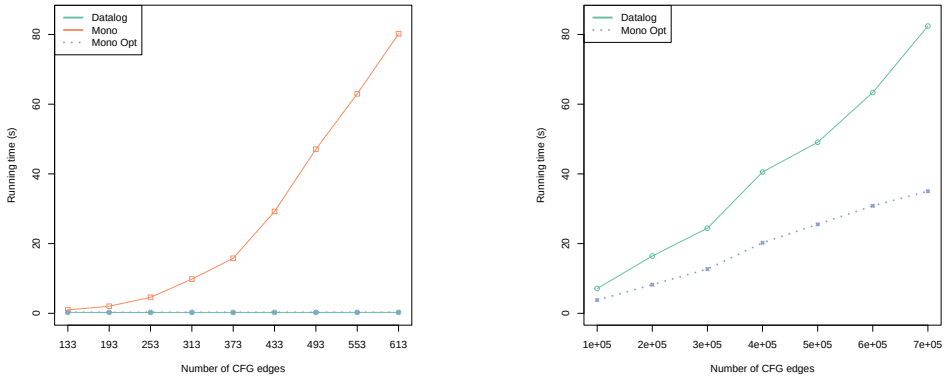
```
intervalAfter(stmt, var, iv) :- assignment(stmt, var, exp), aeval(stmt, exp, iv).
intervalAfter(stmt, var, iv) :- nonassignedVars(stmt, var), intervalBefore(stmt, var, iv).
aeval(stmt, exp, iv) :- ?Var(exp, v), intervalBefore(stmt, v, iv).
aeval(stmt, exp, iv) :- ...
predecessorIntervals(stmt, var, pred, iv) :- cflow(pred, stmt),
                                             intervalAfter(pred, var, iv).
intervalBefore(stmt, var, lub(iv)) :- predecessorIntervals(stmt, var, _, iv).
```

To implement the same analysis using mono types, we propagate a single nested mono map while traversing the control flow graph. The mono map yields a result of type `map[Stmt, map[Var, Interval]]`, which is the standard abstract domain used in flow-sensitive data-flow analysis: For each statement, we compute the abstract value of each variable. With this mono type, we do not need explicit aggregation in the Datalog code anymore. We initialize the nested mono map for the first statement in the analyzed code and then propagate it to all other statements. Note that we use the upper-case `Interval` for the type of interval values and the lower-case `interval` for the interval mono type.

```
traverse(stmt, m) :- isFirst(stmt), m = for stmt : new map[Stmt, map[Var, Interval]](
                                        new map[Var, Interval](new interval)].
traverse(stmt, m) :- cflow(pred, stmt), traverse(pred, m).
```

Similar to before, we need to handle two different cases when we process a statement. If the statement is an assignment of a variable `var` to some expression `exp`, we evaluate `exp` under an environment `env`. We obtain `env` from the map mono by first reading the mono type, which yields the full flow-sensitive analysis result as a nested map, and then performing a map lookup at `stmt`, which yields a variable map. This way, `aeval` does not depend on `intervalBefore` or the like, it simply reads variable values from the given environment. If the analyzed statement does not assign `var`, then we simply propagate its value to all successor statements:

```
transfer(stmt) :- traverse(stmt, m), assignment(stmt, var, exp), cflow(stmt, succ),
                  env = read(m)(stmt), aeval(env, exp, iv), m += (succ, (var, iv)).
```



(a) Pure Datalog vs map mono.

(b) Pure Datalog vs optimized map mono.

Figure 6 Performance results for the interval analysis.

```
transfer(stmt) :- traverse(stmt, m), nonassignedVars(stmt, var), cflow(stmt, succ),
                  env = read(m)(stmt), iv = env(var), m += (succ, (var, iv)).
aeval(env, exp, iv) :- ?Var(exp, v), iv = env(v).
aeval(env, exp, iv) :- ...
```

In our opinion, mono types enable a much cleaner definition of the interval analysis that is very close to the definitions found in program-analysis textbooks. In fact, the definition of `transfer` is most declarative: it describes equations that must hold for each statement. And since `transfer` is neither directly nor indirectly recursive, it can be understood as a specification without even considering a fixed-point semantics. Of course, solving this analysis problem still requires a fixed-point computation due to loops in the analyzed program. This is evident in the analysis definition implicitly: The values written to the map mono depend on the values read from the map mono, which will be translated into recursive aggregation. The mono type protected the programmer from this complication. But, is there a performance penalty for this abstraction?

Figure 6 shows the performance results for the interval analysis. On the left, we can see that the non-optimized mono version is orders of magnitude slower than using pure Datalog. But these are very small programs with only a few hundred CFG edges. We suspect that this slow down is due to the explicit map data structures that needs to be maintained by the non-optimized map mono. The optimization for map monos removes this overhead, by flattening the nested map mono and converting it to relations. Not only does this eliminate the overhead of the mono type abstraction, the optimized program actually scales much better to analyzing large programs. For example, to analyze programs with ~ 700.000 CFG edges, the optimized map mono requires less than 40s whereas the pure Datalog version requires more than 80s. This clearly shows that mono types provide abstraction without regret.

6 Related Work

We develop and mechanize the theory of mono types and their properties in Rocq. To the best of our knowledge, we are the first to introduce a mutable, monotonic container abstraction for Datalog. However, in the context of distributed computing and parallel programming, monotone data types have been studied in the past, which we discuss below. Besides a thorough theoretical exploration, we also integrate and evaluate mono types using the IncA

framework for Datalog. In recent years, numerous extensions and frontends for Datalog have been proposed in the literature. Some of these have features that are related to mono types, as the following discussion shows.

6.1 Datalog Dialects

Compared to all existing Datalog dialects, mono types bring three novel features. First, mono types generalize existing styles of aggregation, making aggregation more expressive. Second, mono types have a mechanized theory in Rocq, including constructions, functors, and an equivalence relation. And third, mono types behave like first-class containers that can be propagated and filled anywhere, which improves programmability and performance. In our discussion below, we focus on the first part, namely the expressiveness of aggregation. This is because no existing Datalog system has a mechanized aggregation theory nor supports anything like first-class containers.

Datafun [3] is a typed higher-order functional programming language that can express Datalog-style programs. Datafun supports aggregation over relations via the join operator of semi-lattices. Our mono types are a generalization of semi-lattices and permit `add` operations that are not computing a least upper bound, such as required for summation. Datafun's key feature is to track monotonicity with types by differentiating discrete and monotone variables. In contrast, we choose to develop and prove the theory of mono types in Rocq and use this foundation to let users design custom mono types. This is a classic trade-off between automation and expressiveness. Specifically, Datafun's type system has limited capabilities for establishing monotonicity and therefore only supports a few built-in semi-lattices, whereas our implementation supports user-defined mono types. Finally, Datafun ensures the termination of fixed-point computations by constraining them to finite semi-lattice types, which are tracked by its type system. With mono types, we do not guarantee termination but instead require the user to prevent infinite ascending chains, which is in conformance with lattice-based aggregation in existing Datalog systems. It would be interesting to study if Datafun can be faithfully extended to support mono types in place of semi-lattices while retaining its metatheory.

Flix [8] and IncA [13] are Datalog frameworks that support recursive aggregation over user-defined aggregation operators. While most aggregation operators in their case studies are semi-lattices, they can also express other monotonic operations, such as summation [15]. Mono types are more general still, because they support a post-processing through the `read` operation, and only the result of `read` must be monotonic. Since we deviated from the textbook semi-lattice aggregation, we established a theory of mono types formally in Rocq. Flix supports validating the monotonicity of user-defined aggregation operators using an SMT solver [7]. However, given the complexity of properties we needed to establish for mono types, it seems highly unlikely that an automated approach can establish the same guarantees. In particular, to enable optimizations of mono types, we had to prove a number of equivalences between mono types. These optimizations have a significant performance impact, as our evaluation shows. Existing Datalog systems cannot optimize aggregations.

BYODS [10] is a Datalog engine and dialect written in Rust with support for custom data structures to back Datalog relations. This not only improves performance, but also allows the user to extend the semantics of Datalog. Namely, BYODS represents relation as a triple $(D, \mathit{inj}, \gamma)$, where D is a lattice that represents the state of the custom relation. The injection function $\mathit{inj} : T \to D$ converts a tuple T to D, which can then be joined with the current state of the custom relation. And the concretization function $\gamma : D \to \mathcal{P}(T)$ extracts a set of tuples from the state of the custom relation. They have proved that if γ is a

monotonic function, the custom relation will not break the fixed point semantics of Datalog. The custom relation is similar to mono types, as D can be seen as the state type, *inj* and γ are close to the `add` and `read` operation respectively. However, mono types do not require the state type to be ordered and the output of mono types is not limited to powersets. In addition, BYODS can also be used to support user-defined lattices and recursive aggregations, which give rise to performing aggregation on map lattices. But they do not explore how to optimize aggregations, which requires an equivalence relation over user-defined types like ours. A promising future work is to combine BYODS with mono types to enable additional performance improvements after our optimizations.

Dedalus [2] introduces state to Datalog to reason about distributed systems. In particular, Dedalus introduces a notion of time to Datalog by annotating relations with a monotonically increasing timestamp. Each change happens at a specific point in time. To persist a fact over time, it is rederived in each subsequent iteration. Effectively, Dedalus evaluates Datalog over time, which enables strong updates by revoking or updating facts. While our mono types also introduce a notion of mutable state to Datalog, they do not permit strong updates. Even though we enforce no restriction on the state of a mono type, we require their output to grow monotonically to guarantee a unique minimal model. It is not obvious how we could support strong updates instead of monotone updates without conflicting with Datalog's fixed-point semantics. In general, mono types and Dedalus serve different purposes. Mono types provide a general purpose abstraction for Datalog to increase programmability and performance. Dedalus on the other hand, evaluates Datalog over time to reason about distributed systems.

6.2 Application of Monotonic Data Types

Conflict-free replicated data types (CRDTs) [12] is a framework for maintaining eventually consistent values in distributed systems. There are two types of CRDTs, operation-based CRDTs and state-based CRDTs. Mono types are more closely related to state-based CRDTs. Specifically, a state-based CRDT object consists of an initial state, a query method to read the current state, an update method to modify state, and a merge method to compute the least upper bound between local state and remote states. CRDT requires the state to grow monotonically after applying an update or merge method. Except for the merge method, we can find that state-based CRDTs and mono types exploit a similar design pattern. However, the monotonicity property serves different purposes in CRDTs and mono types. CRDTs require monotonicity to guarantee that all of the replicas reach the same state eventually. In contrast, mono types use monotonicity to ensure the well-definedness of recursive aggregation in Datalog [13].

Nevertheless, the similarity between CRDTs and mono types leads to a promising direction for future work: extending mono types to Datalog dialects that are used in the distributed system, such as Dedalus [2], Bloom [1], or Bloom^L [4]. Notably, Bloom^L allows passing lattice-based stateful objects across relations, and adopts similar ideas as mono types to collect the inputs of objects. However, mono types provide a more generalized form of lattice-based aggregation that differentiates between state and output, which is more flexible than Bloom^L. In addition, Bloom^L does not provide optimization for sets and maps.

LVars [6] (lattice vars) is a model for deterministic-by-construction parallel programming. Its interface only exposes a put operation (compute the `lub`) and a get operation (to read the state) to programmers. To guarantee determinism, LVars require the get operation to only return a state from a predefined threshold set Q, in which the `lub` of two distinct elements is $\top$ (the error state). With mono types, we do not need to associate a threshold set since we can rely on the eventual consistency of the fixed-point computation.

7 Conclusion

We propose mono types, a monotonic data structure for Datalog to improve its programmability and performance. Mono types support three essential operations: (1) we can instantiate them, (2) we can add elements to them, and (3) we can read monotonically increasing outputs from them. To guarantee well-defined behaviour for Datalog programs, we mechanize and prove the theory of mono types in Rocq. Given this theoretical foundation, we integrate mono types into Datalog by translating them to generalized aggregations over relations. We identify two central container mono types: set mono and map mono. Using a new concept of mono equivalence, we show that these containers are native to Datalog in the sense that we can profit from Datalog's set semantics to implement them. To validate the usability and performance improvements of mono types, we implement two case studies, a dependency analysis and an interval analysis. We show that mono types significantly improve the performance of these analysis compared to pure Datalog.

References

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

2 Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

3 Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional Datalog. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 214–227. ACM, 2016. `doi:10.1145/2951913.2951948`.

4 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. `doi:10.1145/2391229.2391230`.

5 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

6 Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC '13, page 71–84, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2502323.2502326`.

7 Magnus Madsen and Ondrej Lhoták. Safe and sound program analysis with flix. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 38–48. ACM, 2018. `doi:10.1145/3213846.3213847`.

8 Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 194–208. ACM, 2016. `doi:10.1145/2908080.2908096`.

9 Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In Moshe Y. Vardi and Paris C. Kanellakis, editors, *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, pages 114–126. ACM Press, 1992. doi:10.1145/137097.137852.

10 Arash Sahebolamri, Langston Barrett, Scott Moore, and Kristopher K. Micinski. Bring your own data structures to datalog. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1198–1223, 2023. doi:10.1145/3622840.

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

12 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3_29.

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

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

15 Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. Incremental overload resolution in object-oriented programming languages. In Julian Dolby, William G. J. Halfond, and Ashish Mishra, editors, *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, pages 27–33. ACM, 2018. doi:10.1145/3236454.3236485.

16 K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, page 35–44, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1836089.1836094.

17 Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Softw. Syst. Model.*, 15(3):609–629, 2016. URL: https://doi.org/10.1007/s10270-016-0530-4, doi:10.1007/S10270-016-0530-4.