Seamless Deductive Inference via Macros

Arash Sahebolamri
asahebol@syr.edu
Syracuse University
Syracuse, New York, USA

Thomas Gilray
gilray@uab.edu
University of Alabama at Birmingham
Birmingham, Alabama, USA

Kristopher Micinski
kkmicins@syr.edu
Syracuse University
Syracuse, New York, USA

Abstract

We present an approach to integrating state-of-art bottom-up logic programming within the Rust ecosystem, demonstrating it with Ascent, an extension of Datalog that performs well against comparable systems. Rust's powerful macro system permits Ascent to be compiled uniformly with the Rust code it's embedded in and to interoperate with arbitrary user-defined components written in Rust, addressing a challenge in real-world use of logic programming languages: the fact that logical programs are parts of bigger software systems and need to interoperate with other components written in imperative programming languages.

We leverage Rust's trait system to extend Datalog semantics with non-powerset lattices, much like Flix, and with user-defined data types much like Formulog and Souffle.

We use Ascent to re-implement the Rust borrow checker, a static analysis required by the Rust compiler. We evaluate our performance against Datafrog, Flix, and Soufflé using the borrow checker and other benchmarks, observing comparable performance to Datafrog and Soufflé, and speedups of around two orders of magnitude compared to Flix.

CCS Concepts: • **Theory of computation** → *Program analysis*; **Constraint and logic programming**; *Logic and databases*; **Description logics**; • **Software and its engineering** → *Automated static analysis.*

Keywords: Logic Programming, Datalog, Program Analysis, Static Analysis, Rust, Ascent

ACM Reference Format:
Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. 2022. Seamless Deductive Inference via Macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497776.3517779

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04…$15.00
https://doi.org/10.1145/3497776.3517779

1 Introduction

Modern software is often comprised of code written in a mix of multiple programming paradigms including imperative, functional, and logical programming. Logic-programming languages, such as Datalog [5], enable high-performance deductive inference in modern implementations (e.g., LogicBlox [1] or Soufflé [24]) by using efficient operations on database tables to perform the forward-chaining that operationalizes Datalog's declarative inference rules. For example, a business analytics system may use a traditional relational database to store customer relationships, perform computations on that data via a library of functional code, and also perform deductive inference over the database to infer derived properties such as per-customer profit margins, clusters of related customers, or invoice categories.

When integrating logic programming into conventional codebases, programmers grapple with an inherent tension—should they use a fast, dedicated logic programming language, or should they chose to construct a custom task-specific implementation? Dedicated languages and engines are often faster and more scalable (as mature stand-alone implementations), but are challenging to integrate with an existing codebase in another language as they require (de)serialization at the boundary between the deductive inference engine and the system per-se.

In this paper we introduce **Ascent**, a chain-forward logic programming language embedded in Rust via procedural macros. Ascent utilizes a novel compilation methodology (including state-of-the-art techniques such as index generation and semi-naïve evaluation) to translate an extension of Datalog to high-performance relational-algebra kernels implemented in Rust. The central problem tackled by Ascent is to offer all of the benefits of modern high-throughput chain-forward logic programming while cleanly integrating with arbitrary application-logic written in the Rust programming language. Beyond simply generating efficient code, Ascent also allows Datalog rules to call into existing Rust code and vice-versa, enabling code reuse and separation of concerns.

Ascent's semantics extends Datalog, which supports logical rules expressible as Horn-clauses. In practice, using Datalog often requires extensions to its core logic to scale to typical deductive inference tasks (e.g., the PageRank algorithm is not expressible in vanilla Datalog, but is expressible with the common addition of aggregators). While Datalog encompasses many important deductive inference tasks (e.g., the DOOP analysis engine uses the Soufflé Datalog engine to

achieve a high-performance implementation of Java points-to analysis [27]), a few recent languages (such as Datafun [2] and Flix [16]) further extend Datalog with support for fixed-point computations over non-powerset lattices. While these languages offer richer semantics compared to Datalog and enable high-level specification of logic programs over lattices, they do not focus on efficient compilation. Ascent's compilation strategy also includes support for computing fixed points over arbitrary lattices.

In the remainder of this paper, we detail our implementation of Ascent and describe our macro-based compilation methodology. Section 2 details the preliminaries of Datalog and Rust, upon which Ascent is built. Section 3 then introduces Ascent by example, detailing its syntax and semantics (we leave a formal specification of Ascent's semantics to future work); specifics of our implementation are presented in Section 4. We evaluated Ascent in a few ways. First, we reimplemented Polonius [18] (a Datalog-based implementation of the Rust borrow checker using a popular library, Datafrog [21]) in Ascent; our implementation achieves comparable performance (generally within $1-4\times$) to Polonius' Datafrog-based implementation while requiring only half as many lines of code. Next, we use Ascent's lattice features to compute shortest paths in several large graphs; in these experiments we observe our Ascent implementation to be between $56\times$ and $170\times$ faster than a comparable implementation in Flix.

Specifically, we offer the following contributions:

- Ascent, an open-source logic-programming language extending Datalog, embedded in Rust, implemented via Rust's procedural macros.
- More generally, a compilation methodology for lattice-oriented deductive inference to high-performance Rust code.
- An evaluation demonstrating (a) the ability of Ascent to scale to realistic deductive inference tasks by porting the implementation of an existing Rust borrow checker to Ascent, (b) that Ascent's compilation strategy in general and for lattice-based computations enables orders-of-magnitude performance improvements vs. Flix, and (c) that Ascent exhibits competitive performance compared to Soufflé.

2 Background

We now briefly introduce relevant background of Datalog, a chain-forward logic programming language, and Rust, a safety-oriented language intended for development of efficient native code.

2.1 Datalog

A Datalog program is comprised of two components: an extensional database (EDB) enumerating a set of ground predicates (called facts), and an intensional database (IDB) in the form of a collection of Horn clauses (called rules). Each rule has the form $h \leftarrow b_1 \wedge \cdots \wedge b_n$ consisting of a single head clause (h) and a set of body clauses (b_is). Each clause is a predicate name followed by a list of arguments that are either variables or literal constants: $c(a_1, ..., a_m)$. Variables appearing in body clauses are implicitly regarded as universally quantified. For example, the rule $\texttt{path}(x,\ z) \leftarrow \texttt{edge}(x,\ y) \wedge \texttt{path}(y,\ z)$ says that for all x, y, and z, there is a path from x to z if there exists a fact $\texttt{edge}(x,\ y)$ and a fact $\texttt{path}(y,\ z)$.

Datalog enables a specific form of chain-forward logic programming, restricting head clauses to be positive literals (rather than, e.g., negated literals or disjunctions) and body clauses to include only positive literals. These restrictions ensure Datalog programs are computable. Evaluating a Datalog program involves iteratively materializing all facts that must be true, starting from an initial EDB, for each of the rules in the program. Formally, this semantics is specified as the least fixed point of the immediate consequence operator over the set of program rules. The immediate consequence of a rule is the set of facts immediately derivable from the set of ground facts in the current input database: $\texttt{IC}_R(\texttt{DB}) = \texttt{DB} \cup \{\texttt{Head}(R)[\Theta] | \exists \Theta.\, \texttt{Body}(R)[\Theta] \subseteq \texttt{DB}\}$, where $\texttt{Head}(R)$ is the head clause of R and $\texttt{Body}(R)$ is the set of body clauses of R; Θ is a substitution scheme, a mapping from variables to values. Here, facts in the set DB have the same shape as a rule clause: they are predicate symbols followed by a list of arguments. The immediate consequence operator for programs lifts this definition to sets of rules.

Datalog forms the implementation strategy for many state-of-the-art program analyses (e.g., [4, 11, 27]), graph mining [25, 31], and business analytics. Datalog's declarative nature enables terse implementations of complex logic, allowing the user (e.g., a program analysis engineer) to focus on logic core to the task at hand rather than its implementation details (e.g., the DOOP analysis engine is an order of magnitude smaller than other points-to analyses for Java, such as WALA [27]).

Datalog is fairly restrictive, and most production Datalog engines include extensions to the core logic to enable more ergonomic implementation. For example, vanilla Datalog allows only constants to be used as base atoms, and does not support computing over structured data (such as abstract syntax trees representing programs). Thus, several efforts enhance Datalog's expressivity, including allowing defining algebraic data types and pure functions [3], allowing computing fixed points of non-powerset lattices [16], and altering the semantics of the Datalog rules (e.g., disjunctive Datalog [9]). Ascent follows these directions, offering a unique combination of expressivity, convenience, and performance to the user. Importantly, Ascent strives to be a practical language, offering acceptable performance, while integrating cleanly with existing Rust code and libraries.

⟨*rule*⟩ ::= ⟨*head-cl*⟩* **<-** ⟨*body-cl*⟩*;

⟨*head-cl*⟩ ::= ⟨*ident*⟩**(**⟨*expr*⟩***)**

⟨*body-cl*⟩ ::= ⟨*ident*⟩**(**⟨*body-arg*⟩***)**
| **if** ⟨*expr*⟩
| **if let** ⟨*pat*⟩ **=** ⟨*expr*⟩
| **let** ⟨*pat*⟩ **=** ⟨*expr*⟩
| **for** ⟨*pat*⟩ **in** ⟨*expr*⟩
| **!**⟨*ident*⟩**(**⟨*agg-arg*⟩***)**
| **agg** ⟨*pat*⟩ **=** ⟨*expr*⟩**(**⟨*ident*⟩***)** **in** ⟨*ident*⟩**(**⟨*agg-arg*⟩***)**
| **(**⟨*disjunction*⟩**)**

⟨*body-arg*⟩ ::= ⟨*ident*⟩ | ⟨*expr*⟩ | **?**⟨*pat*⟩

⟨*agg-arg*⟩ ::= ⟨*ident*⟩ | ⟨*expr*⟩

⟨*disjunction*⟩ ::= ⟨*body-cl*⟩*
| ⟨*body-cl*⟩* **||** ⟨*disjunction*⟩

Figure 1. Grammar of Ascent rules. A non-terminal followed by * means a comma-separated list of 0 or more occurrences of the non-terminal. ⟨*ident*⟩, ⟨*pat*⟩, and ⟨*expr*⟩ represent valid identifiers, patterns, and expressions respectively.

2.2 Rust

Rust is a modern, type-safe, systems programming language meant to produce efficient native code [19]. Rust supports algebraic data types (ADTs) and pattern matching similar to Haskell and Standard ML and has an expressive trait system comparable to Haskell's type classes, but (similar to C/C++) does not require a managed runtime or a garbage collector. Unlike C and C++, Rust focuses on providing strong type-safety guarantees. Rust's type system disallows spatial and temporal memory violations, including dangling pointers, buffer overflows, use after frees, and similar bugs. Rust's type safety is achieved via a linear type system, which tracks memory ownership and is extended by its borrow system.

Similar to Scheme and Racket [8], Rust has a powerful procedural macro system. Rust procedural macros are functions that take token streams as input, and produce token streams to replace the macro invocation. A procedural macro invocation can contain arbitrary syntax, and is not limited to valid Rust syntax. We use Rust's procedural macros to develop Ascent as a logic programming language embedded in Rust.

3 The Language

Ascent is not a complex language: its power is due to subtle extensions to Datalog, allowing a few more syntactic forms than Datalog, along with the fact that the full power of Rust is at the user's disposal for defining data types, functions, and, as we discuss in 3.1, `Lattice` types to be used in Ascent `lattice` definitions.

The syntax of Ascent rules is presented in Figure 1. Some notable differences from Datalog are allowing patterns and expressions as arguments, and the **for** form, which gives Ascent a higher-order flavor in that it allows relations or sets to be stored as first class values and used like body clauses, as we'll see later in this section.

A typical introductory example in Datalog is computing the transitive closure of an input relation. We'll first use Ascent syntax to present this example.

```
ascent!{
   relation edge(i32, i32);
   relation path(i32, i32);

   path(x, y) <-- edge(x, y);
   path(x, z) <-- edge(x, y), path(y, z);
}
```

These rules compute `path`, the transitive closure of the input relation `edge`. The `relation` definitions specify the arity and types of arguments of relations used in the rules.

The two rules in this program are the exact logical requirements for a path to exist between two nodes in a graph. A Datalog engine or the Ascent compiler is able to compute all the connected nodes in a graph without any further instructions from the user on how it should do so.

The advantages of Ascent become more apparent with more involved examples. Abstract CESK* [30] is an abstract interpreter for λ-calculus. Abstract abstract machines (AAMs) like this are used for static analysis of higher-order languages based on λ-calculus. Abstract CESK* is a tunable analysis, allowing the analysis designer to control the precision and polyvariance of the analysis through the *tick* and *alloc* functions used in the analysis [10, 30]. For instance, one instantiation of these functions yields the well-known k-cfa analysis [26].

$\varsigma \rightarrow_{CESK^*_t} \varsigma'$, where $k \in \sigma(a), b = alloc(\varsigma, k), u = tick(\varsigma, k)$

$\langle x, \rho, \sigma, a, t\rangle$	$\langle v, \rho_2, \sigma, a, u\rangle$ where $(v, \rho_2) \in \sigma(\rho(x))$
$\langle (e_0\ e_1), \rho, \sigma, a, t\rangle$	$\langle e_0, \rho, \sigma \sqcup [b \rightarrow \mathbf{ar}(e_1, \rho, a)], b, u\rangle$
$\langle v, \rho, \sigma, a, t\rangle$	
if $k = \mathbf{ar}(e, \rho_2, c)$	$\langle e, \rho_2, \sigma \sqcup [b \rightarrow \mathbf{fn}(v, \rho, c)], b, u\rangle$
if $k = \mathbf{fn}(\lambda x.e, \rho_2, c)$	$\langle e, \rho_2[x \rightarrow b], \sigma \sqcup [b \rightarrow (v, \rho)], c, u\rangle$

Figure 2. The abstract CESK* machine, taken from [30]

This analysis contains four rules, we present the first and last rules in Ascent as a means of motivating the advantages of Ascent compared to Datalog and a number of Datalog-derived languages. The other rules are omitted to save space, there is no obstruction to writing those in Ascent as well.

```
ascent!{
  relation σ(Addr, Storable);
  relation ς(LambdaCalcExpr, Env, Addr, Time);

  // first rule:
```

```
  ς(v, ρ2, a, tick(e, t, k)) <--
    ς(?e@Ref(x), ρ, a, t),
    σ(ρ[x], ?Value(v, ρ2)),
    σ(a, ?Kont(k));
  // ...
  // last rule:
  σ(b, Value(v.clone(), ρ.clone())),
  ς(e, upd(&ρ2, x, b), c, tick(v, t, k)) <--
    ς(?v@Lam(..), ρ, a, t),
    σ(a, ?Kont(k)),
    if let Fn(Lam(x, e), ρ2, c) = k,
    let b = alloc(v, t, k);
}
```

The relation σ is the store, associating abstract addresses with values (closures or continuations), the relation ς is the state of the abstract machine, and the rules dictate how the machine steps from one state to another.

The first rule states that if the control expression in the current state is a variable reference x, the address for x is looked up in the current environment ρ, and its value (v, ρ_2) is looked up in the store. The machine then transitions to a state with v as the control expression and ρ_2 as the environment, with the continuation address component (a) of the state unchanged.

The last rule says that if the current control expression is a value v (a lambda in pure λ-calculus), and there is a function-application continuation associated with the current continuation address a, the machine transitions a state where the control expression is the body of the invoked function, e, the environment is updated to bind the parameter of the function, x, to an allocated binding address, b, which in turn is bound to the closure (v, ρ) in the abstract store.

Because the analysis computed is a finite over-approximation of program behaviors, the store is weakly updated to accumulate the new closure along with any other closures bound to the same address. The finite set of timestamps, each u, is used along with the abstract allocation function to finitize the set of abstract addresses, in turn bounding the complexity of the analysis.

In the Ascent rules, we seamlessly weave logical inference with the additional utilities provided by Ascent: we use ADTs to represent various components of the analysis directly, including the AST of the language under analysis, the `Storables` (values stored in the store), and the possible `Continuations`. We use pattern matching to operate over values of these ADTS. For example, in the clause `ς(?v@Lam(..), ρ, a, t)`, we check that the current control expression is a `Lam`, and bind it to variable `v`. The `?` in argument position signifies that what follows is a pattern and not an expression. We use functions written in Rust for definitions of `alloc` and `tick`.

Another interesting point in this example is handling of environments. They map variables to addresses in the abstract CESK* machine. In the Ascent analysis we are able to represent them as exactly that, a mapping from variables to addresses: `type Env = Rc<BTreeMap<Var, Addr>>`. Note that this analysis is not expressible in Datalog due to the use of first-class environments [4, 22, 29].

Furthermore, Ascent permits arbitrary expressions in argument position, allowing us to directly look up the address of the variable in the first rule in the clause `σ(ρ[x], ...)`.

Among the implementations of Datalog and languages supporting logical inference in the style of Datalog that we are aware of, we believe this combination of features is unique. Soufflé [24] is a popular high-performance Datalog engine. It has limited support for user-defined data types, but defining a data structure like `BTreeMap` that allows efficient lookup is not realistic in Soufflé. Moreover, user-defined functions, what Soufflé calls functors, must be defined in C++ and compiled separately. Flix [16] is a language that allows logical inference and supports ADTs and user-defined functions. However, Flix does not support expressions in argument position, nor does it support pattern matching inside rules. Formulog [3] is a language that combines Datalog with ADTS, first-order functions, and SMT solvers. Formulog allows expressions and patterns as predicate arguments, but as we'll see later, it lacks other features of Ascent, including generative clauses and lattices.

A beneficial aspect of Ascent is the fact that it affords the programmer precise control over the data structures and the memory layout of the values manipulated by Ascent rules. In the abstract CESK* example, we used `BTreeMap` for environments. The user might determine that `SmallVec` (or a similar data structure that avoids heap allocation if possible) is a better choice, since environments are bounded in size (by the number of variables appearing in the program under analysis), or in another scenario, the user might determine that since updating `BTreeMaps` requires cloning the B-tree[1], a better choice would be an immutable data structure that allows sharing of keys/values across updates, and use a hash array mapped trie (HAMT), a popular data structure for immutable maps.

A careful look at Figure 2 may reveal that there is a difference between the mathematical description of the abstract CESK* machine and the Ascent rules implementing it. The Ascent implementation utilizes a widened store [26], a technique for improving the time complexity of such analyses. Keeping the stores state-specific is also possible in Ascent, here is the first rule of the analysis, rewritten to accommodate state-specific stores:

```
ς(v, ρ2, σ, a, tick(v, t,k)) <--
  ς(?Ref(x), ρ, σ, a, t),
  for xv in σ[&ρ[x]].iter(), if let Value(v,ρ2) = xv,
  for av in σ[a].iter(), if let Kont(k) = av;
```

[1]We take this opportunity to stress that accidental (or otherwise) mutation of values stored in relations, which would break the semantics of Ascent, is prevented by Rust's type system: a variable appearing as a relation argument is an immutable reference to the stored value.

We use generative clauses in this rule to iterate over all the possible values stored for a single address. A generator's syntax is taken from the Rust `for` loop. This is intentional to signal to the Ascent user that she can take advantage of Rust's iterators to populate relations. To show the benefit of generators, we point to Soufflé generative functor `range`. In Soufflé, to populate a relation `foo` with numbers 0 to 4, one can write `foo(x) :- x = range(0,5)`. The same can be achieved in Ascent by using the syntax `foo(x) <-- for x in 0..5`. The important difference is that any `Iterator`-valued expression can be used in place of `0..5`, including values stored in relations as shown in the rule above. This is another way in which Ascent takes advantage of the existing Rust ecosystem.

3.1 Going Beyond the Powerset Lattice

A Datalog implementation discovers additional facts from the set of currently known facts until a fixed point is reached. In other words, it ascends the powerset lattice to a fixed point for a function encoding the Datalog rules (the immediate consequence operator). Every point in the lattice is a set of facts, and these points are ordered by set inclusion.

Accumulating more and more facts however is not always what is required. For example, if we needed to compute the shortest path lengths in a weighted graph, all the standard Datalog semantics would be able to do is to compute all the possible path lengths between nodes, not just the shortest path lengths. Worse yet, if the graph contained cycles, the Datalog program would diverge, as there are an infinite number of paths between nodes in a cycle.

The shortest paths problem can be framed as follows. We are not interested in the powerset lattice: given the current database of facts $\mathtt{DB}_0 \cup \{\mathtt{path}(a, b, l_1)\}$ and a newly discovered fact $\mathtt{path}(a, b, l_2)$, we are not interested in $\mathtt{DB}_0 \cup \{\mathtt{path}(a,\ b,\ l_1),\ \mathtt{path}(a,\ b,\ l_2)\}$ as the new database. Instead, we want to only store $min(l_1, l_2)$ for nodes a and b: $\mathtt{DB}_0 \cup \{\mathtt{path}(a,\ b,\ min(l_1, l_2))\}$.

Flix [16] was one of the first languages to tackle this problem by allowing the user to define arbitrary lattices in addition to normal relations. Ascent takes inspiration from Flix and provides similar capabilities.

The Ascent user can use the `lattice` keyword in place of `relation` to define a lattice. Here, we take advantage of Rust's type system in enforcing the requirements for a lattice. The last column of a defined `lattice` must implement the `Lattice` trait (defined in the `ascent` library).

A lattice is a partial order where every pair of elements has a least upper bound and a greatest lower bound defined. The definition of the `Lattice` trait then is not surprising:

```
pub trait Lattice: PartialOrd + Sized{
  fn meet(self, other: Self) -> Self;
  fn join(self, other: Self) -> Self;
}
```

This trait is implemented for most of the types of the standard library where an implementation is appropriate, the types include integers, `bool`, `Option<T>`, and tuple types, among others. The `ascent` library also provides a number of type wrappers for redefining lattice operations. One notable example is `Dual<T>`: if `T` is a `Lattice`, so is `Dual<T>`, flipping the pairs of operations (`<=`, `>=`) and (`meet`, `join`).

We can now come back to our motivating example and write the Ascent program for computing the shortest path lengths in a weighted graph:

```
ascent!{
  lattice shortest_path(i32, i32, Dual<u32>);
  relation edge(i32, i32, u32);

  shortest_path(x, y, Dual(*w)) <-- edge(x, y, w);
  shortest_path(x, z, Dual(w + l)) <--
    edge(x, y, w),
    shortest_path(y, z, ?Dual(l));
}
```

Since we are interested in shortest paths, we use the type `Dual<u32>` to ensure that for any pair of nodes, from the path lengths computed so far, only the shortest one is stored.

Semantically, an Ascent `lattice` works differently from a `relation`. While a relation can be denoted as a set that an Ascent evaluation iterates to a fixed point by adding more facts to, a `lattice` defined in Ascent is denoted as a partial map from the non-lattice columns (every column except the last one) to the lattice column. For our shortest paths example this would be $(\mathtt{i32}, \mathtt{i32}) \rightharpoonup \mathtt{Dual<i32>}$. A fact of an Ascent `lattice` then constitutes a partial map defined on a single point: the non-lattice columns of the tuple.

The least upper bound of two `lattice`s is defined as follows: when a point is in the domain of both partial maps, join their respective values, and when a point exists only in one of the maps' domains, just take its value.

$$\begin{aligned}\mathtt{DB}_1 \sqcup \mathtt{DB}_2 = \{(x,\ y \sqcup z) \mid (x,\ y) \in \mathtt{DB}_1, (x,\ z) \in \mathtt{DB}_2\} \cup \\ \{(x,\ y) \mid (x,\ y) \in \mathtt{DB}_1 \wedge \nexists\, z.\ (x,\ z) \in \mathtt{DB}_2\} \cup \\ \{(x,\ z) \mid (x,\ z) \in \mathtt{DB}_2 \wedge \nexists\, y.\ (x,\ y) \in \mathtt{DB}_1\}\end{aligned}$$

The Ascent compiler emits code that uses the normal set union semantics for `relation`s, and respects this semantics for `lattice`s.

Having lattices enables more static analyses in Ascent. Going back to the abstract CESK* example, we may want to augment the language under analysis with numbers and arithmetic operations, and keep track of variables bound to numbers in addition to closures.

Doing so in pure Datalog poses a problem (beyond Datalog's inability to express environments). If we assume arithmetic operations are available in Datalog, we can choose to have a store for numbers, and have rules in the analysis to represent the arithmetic operations in the language being analyzed. The problem with this approach is that unlike regular abstract CESK* analysis, which, with a suitable choice

of the *alloc* and *tick* functions, guarantees termination by abstracting the store addresses and in turn finitizing the set of `Storables`, the set of integers is not finite, entailing that the analysis may diverge (e.g., the term $let\, f(x) = f(x{+}1)\ in\, f(0)$ will cause the analysis to diverge).

The issue again stems from the analysis being confined to the powerset lattice and not being able to abstract numbers. Ascent provides a solution: define a lattice of finite height to abstract numbers and use it for storing numbers. One candidate for such a lattice is a flat lattice, often used for constant propagation. Using this lattice ensures that the analysis will not diverge, as once the analysis tries to store two different numbers at the same address, this lattice ensures they get replaced by `Top`. Here is a snippet of the modified analysis capable of handling numbers:

```
// ...
lattice σnum(Addr, ConstPropagation<i64>);

σnum(b.clone(), lit),
ς(e, upd(&ρ2, x, b), c, tick(v, t, k)) <--
  ς(?v@Lit(lit), ρ, a, t),
  σ(a, ?Kont(k)),
  if let Fn(Lam(x, e), ρ2, c) = k,
  let b = alloc(v, t, k);
```

This rule is similar to the last rule of Figure 2, except that it handles numbers rather than lambdas: if the current control expression is a (abstract) number, the analysis stores it in the `σnum` `lattice`, ensuring the number of `σnum` facts stored is finite, which in turns guarantees termination of the analysis.

3.2 User-Extensible Aggregation

Stratified aggregation and negation are useful additions to Datalog implemented by many Datalog variants. They can be simulated in Ascent through use of appropriate lattices. Negation, sum, and count are achievable by utilizing a `Set` lattice; and min and max are less expensive, by respectively using the `Dual` type or integers directly. However the ergonomics and performance of such solutions are not ideal.

To address this, Ascent provides aggregation clauses. Unlike typical implementations of Datalog, aggregation in Ascent is not confined to a set of built-in aggregators. The `ascent` library provides common aggregators expected in Datalog implementations (`min`, `max`, `sum`, `count`, and `mean`); however, these aggregators are provided as library functions. The user is free to define her own aggregators to be used inside Ascent rules. One can define a `median` aggregator for example, or even a `percentile` aggregator: an aggregator parameterized over k that returns the k-th percentile of the desired column of the aggregated relation:

```
relation population(i64);
relation population_75p(i64);
// ...
population_75p(p75) <--
  agg p75 = (percentile(75))(x) in population(x);
```

4 Implementation

Ascent is embedded in Rust via procedural macros. Thus, the Ascent compiler is invoked through the appropriate macro invocation `ascent! {...}`. When invoked, the compiler processes the input token stream and produces Rust code for efficient evaluation of the input program rules. The Ascent implementation is available online[2]. The compiler is comprised of several key stages, which we describe in this section.

Parsing and desugaring. We represent Ascent's abstract syntax tree (AST) in Rust via conventional structural types. Ascent first parses the token stream into an AST. During this process, Ascent also desugars some superficial language constructs such as `?` patterns and disjunctive clauses (e.g., `(foo(x) || bar(x))`).

Index selection. Modern Datalog engines achieve performance by having a variety of indices for each relation according to its usage in the program. Ascent computes necessary indices by traversing the AST and computing—for each relation—subsets of columns relevant to joins. For example, the inclusion of a rule `foo(x,z) <-- bar(x,y), baz(x,y,z)` would require `bar` and `baz` to be indexed on their first two columns to allow efficient iteration that only looks for potential matches on the joined columns. This pass first computes relevant indices, then particularizes body clauses within rules to use the appropriate index for a given relation. This stage also ensures each `lattice` is indexed on its non-lattice-columns, required for efficient updating of its facts.

Computing SCCs and Incrementalization. Rules of Datalog programs, and by extension Ascent, are stratified into strata of strongly connected components before execution. Ascent computes a dependence graph of rules (rule A depends on rule B iff there is a relation R mentioned in rule A's body and rule B's head) and then computes SCCs of this dependence graph. For each SCC, we explicate static (unchanged) and dynamic (inductively updated) relations to subsequently enable semi-naïve evaluation.

Semi-naïve evaluation is an efficient evaluation strategy for Datalog which refines the naïve (chaotic iteration) strategy to avoid redundant work by using a worklist. This is accomplished by partitioning each relation into three sets: **new**, **delta**, and **total**. **total** is the subset of facts that have been discovered in iterations prior to last. **delta** is the facts discovered in the last iteration, and the newly discovered facts for a relation are added to the **new** set. This partitioning guarantees that the **total** versions of dynamic relations do not need to be joined together, as they won't yield any new facts. After determining which relations within an SCC are

[2] https://github.com/s-arash/ascent

dynamic, rules in the SCC are duplicated to contain the necessary combinations of **delta** and **total** versions of relations used within their bodies.

Code generation. The incrementalization stage is the last before code generation. To perform code generation, Ascent emits a `struct` for the input Ascent program containing one field for each relation (or lattice) in the program, and one field for each index of a relation/lattice. Next, we generate a single `run` method which acts as the entrypoint to perform evaluation of the program. This method walks over the SCC graph in topologically-sorted order and, for each SCC evaluates the rules within that SCC using the semi-naïve evaluation strategy.

Our compilation strategy for rules is similar to that of Soufflé's: rules are compiled to a series of nested loops iterating over values of respective tuples in relations appearing within the rule to perform joins [24]. While the order of clauses in Ascent is significant, the compiler is capable of reordering clauses in cases where such reordering does not introduce ungrounded variables, which is crucial to efficient evaluation of rules.

For example, consider the following rule for computing the transitive closure of a relation: `tc(x, z) <-- r(x, y), tc(y, z)`. This rule is translated to the following Rust code (simplified for clarity).

```
if r_ind_1_total.len() <= tc_ind_0_delta.len(){
  for (y, r_tuples) in r_ind_1_total.iter(){
    if let Some(tc_tuples)=tc_ind_0_delta.get(y){
      for &r_ind in r_tuples.iter() {
        let x = &self.r[r_ind].0;
        for &tc_ind in tc_tuples.iter() {
          let z = &self.tc[tc_ind].1;
          let new_row: (i32, i32) = (*x, *z);
          // insert new_row into tc and update
          // its indices
        }
      }
    }
  }
} else {
  // iterate over tc first ...
}
```

Here `r_ind_1_total` and `tc_ind_0_delta` are the indices required for the join. The emitted code first picks the smaller one (`r` shown here), then iterates over its key-value pairs, finding matches in `tc`. Once a match is found, the matching tuples of `r` and `tc` are iterated over, yielding new facts to be added to the **new** version of `tc`.

In addition to the `ascent` macro, we provide the `ascent_run` macro. The output of invocations of this macro are expressions, as opposed to items for `ascent` invocations. The output of `ascent_run` is a value containing fields for relations defined in the Ascent program, computed to a fixed point. The main advantage of `ascent_run` is that the rules in the Ascent program will have access to local variables in scope.

Additionally, Ascent programs can be generic. Ascent allows a `struct` declaration at the top of an Ascent program, which can include type parameters.

We combine these features to define a generic version of the TC program that computes the (optionally reflexive) transitive closure of the input relation:

```
fn compute_tc<N>(r: Vec<(N, N)>, reflexive: bool)
 -> Vec<(N, N)> where N: Clone + Hash + Eq {
  ascent_run!{struct TC<N: Clone + Hash + Eq>;
    relation r(N, N) = r;
    relation tc(N, N);
    tc(x, y) <-- r(x, y);
    tc(x, z) <-- r(x, y), tc(y, z);
    tc(x, x), tc(y, y) <-- if reflexive, r(x, y);
  }.tc
}
```

`Clone`, `Hash`, and `Eq` are constraints that every relation column type must satisfy.

5 Evaluation

In this section, we describe our evaluation of Ascent via three sets of experiments. The first is the reimplementation of Polonius, a Datalog-based implementation of the Rust borrow checker. The second evaluates our compilation of lattices by comparing the implementation of a shortest-paths algorithm in Ascent with an equivalent implementation in Flix. The third uses a simple graph-mining algorithm to compare the performance of Ascent with Soufflé. All the experiments were performed on a machine with a Core i7-8650U Intel CPU and 16GB of RAM.

5.1 Rust Borrow Checker

Rust's type system includes a sophisticated borrow checker to check memory usage invariants. For example, the borrow checker ensures that references point to live memory at the time of each access, a crucial property for type safety of Rust programs. The Rust borrow checker can be expressed as a series of Datalog rules [17]. This formulation of the Rust borrow checker has inspired a formalization of the Rust language, along with safety proofs [33].

To provide intuition for how the Datalog-based borrow checker works, we use the following example:

```
1  let mut v = vec![1, 2];
2  let r: &i32 = &v[0]; // L1
3  v.push(3);
4  println!("r is {}", r);
```

This innocent-looking piece of code has an illegal memory access, and is rejected by the Rust compiler (specifically by the borrow checker)—`r` takes a reference to an element of vector `v` (it *borrows* `v`), then `v` is mutated, possibly deallocating the memory referenced by `r`, meaning that the final line is potentially reading deallocated memory.

Each reference type (e.g., `&i32`) is annotated with a unique *origin* parameter, along with each reference (or loan). The

Table 1. The Polonius benchmarks. Times are in seconds. subset size refers to the # of facts in the relation `subset`, the biggest relation in all test cases. clap-rs is the benchmark in the Polonius repo. serde-fmt is the biggest function (in # of input facts) in the serde library [7]. ascent-codegen is the code generation function of the Ascent compiler. ascent_naïve_compute is Rust code generated by the naïve implementation of borrow checker in Ascent. chess-search is the search function of a chess engine.

Benchmark		Naïve Analysis				Optimized Analysis			
name	LOC	subset size	Datafrog time	Ascent time	slow-down	subset size	Datafrog time	Ascent time	slow-down
clap-rs	2100	6.7M	12.03	11.96	.99x	2.5M	4.49	4.54	.99x
serde-fmt	170	4.5M	1.83	4.45	2.4x	2.1M	0.43	0.72	1.7x
ascent-codegen	800	1.61M	0.80	2.09	2.6x	0.75M	0.28	0.40	1.4x
ascent_naïve_compute	1000	14.2M	10.8	44.0	4.1x	6.0M	2.65	5.20	1.96x
chess-search	600	25.1M	17.0	55.7	3.3x	14.6M	4.64	10.2	2.2x

usual subtyping relation must hold between variables and their values, and thus in line 2 of the above example, the type of `r` (which we here call `&'a i32` by introducing the origin parameter) must be a supertype of the type of expression `&v[0]` (which we'll call `&'b i32`); this in turn requires that `'a` subsumes `'b` (`'b:'a`). Intuitively, this may be read as "`'b` outlives `'a`". These relations must hold wherever the control flow graph leads. In our example, this means that the relation must hold on lines 3 and 4 as well. We can summarize this requirement as a logical rule:

```
relation cfg_edge(Point, Point);
relation subset(Origin, Origin, Point);
subset(origin1, origin2, point2) <--
  subset(origin1, origin2, point1),
  cfg_edge(point1, point2);
```

We also need to track the kinds of loans. There are two kinds of loans in Rust, shared (immutable), and unique (mutable). The problem with the above code is that a second loan exists for `v` while a unique loan is active, which is disallowed by Rust's borrow checker. This fact is encoded as an input fact of the form `loan_invalidated_at(L1, line_3)`. The analysis also includes rules for tracking what loans are alive at what points of the execution (`loan_live_at`):

```
loan_live_at(loan, point) <--
  origin_contains_loan_on_entry(origin, loan, point),
  origin_live_on_entry(origin, point);
```

`origin_live_on_entry` is an input to the analysis, and `origin_contains_loan_on_entry` propagates another input to the analysis, `loan_issued_at` through the `subset` and `cfg_edge` relations. This means that `origin_contains_loan_on_entry('a, L1, line_3)` is derivable. From our example, `'a` is alive at line 3 because it is accessed on line 4; combined with the aforementioned derived fact, we can derive that the loan L1 is alive at line 3. On the other hand, L1 is invalidated at line 3. These facts combine to produce the error:

```
error(loan, point) <--
  loan_invalidated_at(loan, point),
  loan_live_at(loan, point);
```

The Polonius [18] project implements the borrow checker using Datafrog [21], a library for writing Datalog rules in Rust. This borrow checker is experimentally integrated into the Rust compiler, although it has not yet replaced the existing hand-written borrow checker.

We implemented both the naïve and optimized versions of the borrow checkers in Polonius using Ascent. These both compute the same potential errors, with the optimized version using domain-specific knowledge to reduce the number of intermediate facts generated, decreasing execution time.

We compared our results with the existing Datafrog-based implementation, as summarized in Table 1. The performance of the Ascent implementation is comparable to the Datafrog implementation for the only benchmark in the Polonius project: clap-rs, though we also include four more benchmarks not present in Polonius' implementation. We used mostly macro-generated code for the other benchmarks, as they tend to be big enough in size to meaningfully stress the implementations. We ensured the correctness of our implementation by comparing the output of each analysis and verifying that their results match exactly.

The difference in performance ranges from roughly equal in some benchmarks, to Datafrog being about 4× faster in the worst benchmarks for Ascent. This difference is partially attributed to the fact that in Datafrog, relation tuples must be total orders, allowing Datafrog to utilize optimal multi-way join algorithms [23]. Ascent can be updated to use optimal join algorithms when the user instructs the compiler that relations have total order columns, utilizing optimal joins when possible while retaining its current flexibility.

With respect to Polonius, Ascent's significant advantage is its concision. In the naïve analysis, the Datafrog implementation is 101 lines of code, versus 46 LOC for Ascent; in the optimized analysis, the difference is 210 LOC for Datafrog vs 100 LOC for Ascent. We also observe an ergonomic benefit to Ascent, which avoids unnecessary low-level details that must be specified in the Datafrog implementation. For example, here is a Datafrog rule from Polonius:

Table 2. Shortest paths benchmarks. Times are in seconds. Loop1000 is a loop with 1000 nodes, Complete100 and Complete1000 are complete graphs with 100 and 1000 nodes. Watts-Strogatz is a random small-word graph [32] with random weights. Facebook survey is a Facebook friendship graph [20]. HE PhysTH is a coauthorship graph of High Energy Physics theory papers [14]. DNF indicates that the analysis failed with an out of memory error after running for several hours.

Benchmark			SP Lengths			Shortest Paths		
name	nodes	edges	Flix time	Ascent time	speedup	Flix time	Ascent time	speedup
Loop1000	1000	1000	59.7	0.62	96x	60.6	0.69	87x
Complete100	100	9900	4.3	0.025	172x	5.9	0.057	103x
Complete1000	1000	999000	6328	42.8	148x	6779	121	56x
Watts-Strogatz	1000	1999	150	1.34	112x	161	1.60	100x
Facebook survey	4039	88233	468	4.87	96x	485	7.0	69x
HE PhysTH	9877	51970	DNF	315	—	—	378	—

```
subset_errors.from_leapjoin(
 &subset,(
  placeholder_origin.extend_with(|&(o1, _o2, _p)|o1),
  placeholder_origin.extend_with(|&(_o1, o2, _p)|o2),
  known_placeholder_subset
   .filter_anti(|&(o1, o2, _p)| (o1, o2)),
  datafrog::ValueFilter::from(|&(o1, o2, _p), _| {
   o1 != o2
  }),
 ),
 |&(o1, o2, p), _| (o1, o2, p),
);
```

By contrast, the corresponding rule in Ascent is much terser:

```
subset_error(origin1, origin2, point) <--
  subset(origin1, origin2, point),
  placeholder_origin(origin1),
  placeholder_origin(origin2),
  !known_placeholder_subset(origin1, origin2),
  if origin1 != origin2;
```

In Datafrog, indexing must be performed manually: the user is responsible for introducing different versions of the same relation with different indices and writing rules for populating these versions. There are also restrictions on multi-way joins: such a join can include no more than one dynamic relation (one that is read from and updated in the same iteration). Additionally, Datafrog cannot compute the interdependencies of rules and break down a collection of rules into a series of SCCs for optimal computation. All such optimizations must be done manually by the programmer.

5.2 Shortest Paths

We used the shortest paths program, introduced in 3.1 as a motivating example for lattices, to compare the performance of Ascent with Flix. We also wrote a version of the program that computes the actual shortest paths in addition to shortest path lengths. Modifying the shortest path lengths program to include the actual paths is straightforward in Ascent:

```
lattice shortest_path(i32,i32,(Dual<u32>,List<i32>));
shortest_path(x, z, (Dual(w + len), cons(x, p))) <--
  edge(x, y, w),
  shortest_path(y, z, ?(Dual(len), p));
// ...
```

Here, `List<T>`, used to store the paths between nodes, is a singly-linked list, and the type `(Dual<u32>, List<i32>)` is a total order with lexicographic ordering, making it a `Lattice` with the correct semantics for this problem.

We wrote both programs in Flix, and compared their performance on a collection of test graphs. The graphs include both synthetic and real-world graphs. We used Flix version 0.25, the latest version available at the time of performing our experiments. The results are presented in Table 2.

We saw significant speedups of around two orders of magnitude in our Ascent implementations, with the smallest speedup being 56×. This again highlights the benefit of our compilation strategy: we compile the source Ascent program directly to high-performance Rust code. The generated Rust code, when compiled, avoids any dynamic dispatch (which Flix relies on for invoking joins, among other operations), and avoids unnecessary indirections, including boxing of simple values (as required by the Java runtime): the `Dual<u32>` type has the exact same footprint as `u32`, but the same cannot be said of the wrapper type we had to use for integers in Flix to implement `leastUpperBound`.

5.3 Graph Mining

As a means of comparing Ascent's performance with Soufflé, we implemented identical programs for discovering cliques of up to size 5 in Ascent and Soufflé's Datalog. We ran the programs on a number of real-world graphs and verified that the results match exactly. We used Soufflé version 2.1 in compiler mode. The results are presented in Table 3. We note that while Ascent is competitive with Soufflé's single-threaded performance, Soufflé is capable of generating multi-threaded code, a feature currently absent in Ascent.

Table 3. 5-clique benchmarks. Times are in seconds and represent best of five runs. Astro Phys is a coauthorship graph of Astrophysics papers [14]. Brightkite is a location-based social network [6]. Enron emails is a graph of email communications [12]. Slashdot Zoo is a social network [13].

Benchmark			Times		
name	nodes	edges	Soufflé	Ascent	speedup
HE PhysTH	10K	52K	0.11	0.07	1.6x
Astro Phys	19K	396K	15.6	19.6	0.8x
Brightkite	58K	428K	5.73	4.76	1.2x
Enron emails	37K	368K	2.92	2.15	1.4x
Slashdot Zoo	77K	905K	4.47	3.45	1.3x

6 Related Work

Compilation of Datalog. Soufflé [24] is a high-performance Datalog engine that uses a compilation strategy similar to Ascent's: both utilize a systems programming language as the compilation target. For Soufflé the target language is C++. Soufflé utilizes efficient index selection schemes [28] and various other techniques for generating efficient code. While Ascent does not yet utilize all these techniques, they can in principle be implemented in Ascent.

Extensions to Datalog. Formulog[3] is an extension to Datalog which aims to integrate an SMT solver with Datalog. Formulog has support for defining algebraic data types and first order functions operating over the ADTs in addition to primitive data types. In Formulog the defined ADTs can appear inside SMT formulas. Additionally, uninterpreted functions and sorts can be defined in Formulog for use inside SMT formulas; however, first-order functions defined in Formulog are not accessible in SMT formulas. This may force the programmer to effectively define functions that need to be used inside and outside SMT formulas twice, once as a Formulog function, and once as a formula.

It is theoretically possible to interface with SMT solvers in Ascent, so long as there are Rust libraries for doing so. However, Ascent would not not have the tight integration with SMT solvers that Formulog provides, as Formulog is designed specifically with that goal in mind.

Computing fixed points on non-powerset lattices. As discussed in Section 3, Flix combines Datalog-style rules with ADTS, functions, and the ability to compute non-powerset-based fixed points by defining lattices, which was the inspiration for implementing lattices in Ascent.

Later iterations of Flix have evolved in an orthogonal direction [15], embedding logic programs in a functional programming language, where the logic programs are first-class values and can be combined to yield new logic programs, a capability that Ascent lacks. In addition to allowing more expressivity in defining rules, as discussed in Section 3, Ascent is more performant than Flix, as we saw in Table 2.

Datafun [2] is a foundational attempt at combining higher order functional programming with computing fixed points on lattices. Datafun is capable of guaranteeing termination of fixed point computations by tracking the monotonicity of functions and the finiteness of the height of lattices on which the monotone functions operate, capabilities that are absent in Ascent (or Flix). We regard Ascent as a practical attempt at allowing computing fixed points of non-powerset lattices, with syntax already familiar to Datalog users; compared to the foundational nature of Datafun.

Logic programming in Rust. Datafrog [21] is a library for writing Datalog rules in Rust. Since Datafrog is not a language, writing Datalog rules in Datafrog tends to be verbose: the user is responsible for explicitly defining all the indices of a relation that are required, and ensuring those indices remain consistent by adding rules for transferring facts between them. The forms the rules take are also somewhat verbose as we discussed in 5.1.

Datafrog requires relation tuples to be total orders, allowing it to perform efficient multi-way joins [23] (although it imposes limitations on them: only one dynamic relation is allowed in a multi-way join). Ascent does not have that requirement, and does not take advantage of these efficient join algorithms currently. Having a best-of-both-worlds scenario, where relation tuples are not required to be total orders, but can be marked as so by the Ascent user to unlock better join algorithms is a future possibility for Ascent.

7 Future Work

There are a few areas of improvement that we'd like to work on to evolve Ascent.

Performance. We plan to investigate enabling more optimal join algorithms (e.g., [23]), and more optimal index selection schemes [28]. As was discussed earlier, currently Ascent relies on hashing for indexing relations, which allows more types to be used as relation columns, but sacrifices some performance.

Parallelization. The code generated by Ascent is currently single-threaded. Updating the compiler to generate parallel code is another avenue of improvement for Ascent. The advantage of our approach is that such a change will be transparent to the Ascent user as long as thread safety is not a concern. When types used as relation columns are not thread safe, the Rust type system will be able to flag those, ensuring future parallelized Ascent programs will be thread safe. `Rc`, the reference counted pointer type that we used in our examples, is an instance of a non-thread-safe type. The fix in this case is simple: using `Arc`, the thread-safe version of `Rc` instead.

8 Conclusion

We presented Ascent, a logic programming language in the style of Datalog embedded in Rust. Embedding Ascent in Rust allows the Ascent user to immediately take advantage of Rust's language features and ecosystem.

Throughout the paper, we demonstrated Ascent's expressivity, practicality, and performance. The abstract CESK* example was designed to show the expressivity of Ascent, including its ability to handle abstract domains through its support for lattices. The Polonius comparison was done to showcase the convenience and practicality of using Ascent: we were able to write the analysis exactly as it would've been written in Datalog from within Rust, where it is needed. We used the shortest paths program to demonstrate Ascent's performance, and the advantages of our compilation strategy. Finally, we showed that Ascent is competitive with Soufflé using the graph mining benchmarks. We hope to have demonstrated that Ascent presents an innovative approach to implementing compilers for logic programming, one that yields a powerful and practical language.

References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[2] Michael Arntzenius and Neelakantan R Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 214–227. https://doi.org/10.1145/2951913.2951948

[3] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based static analysis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31. https://doi.org/10.1145/3428209

[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[5] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

[6] Eunjoon Cho, Seth A Myers, and Jure Leskovec. 2011. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1082–1090. https://doi.org/10.1145/2020408.2020579

[7] The Serde Developers. 2021. *serde-rs/serde: Serialization framework for Rust.* Retrieved November 2, 2021 from https://github.com/serde-rs/serde

[8] R. Kent Dybvig. 1992. Writing Hygienic Macros in Scheme with Syntax-Case.

[9] Thomas Eiter, Georg Gottlob, and Heikki Mannila. 1997. Disjunctive Datalog. *ACM Trans. Database Syst.* 22, 3 (sep 1997), 364–418. https://doi.org/10.1145/261124.261126

[10] Thomas Gilray, Michael D. Adams, and Matthew Might. 2016. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-flow Analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP '16)*. ACM, New York, NY, USA, 407–420. https://doi.org/10.1145/2951913.2951936

[11] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[12] Bryan Klimt and Yiming Yang. 2004. Introducing the Enron corpus.. In *CEAS*.

[13] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1361–1370. https://doi.org/10.1145/1753326.1753532

[14] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2–es. https://doi.org/10.1145/1217299.1217301

[15] Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the masses: programming with first-class Datalog constraints. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428193

[16] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices* 51, 6 (2016), 194–208. https://doi.org/10.1145/2908080.2908096

[17] Nicholas Matsakis. 2018. *An alias-based formulation of the borrow checker.* Retrieved November 15, 2021 from http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker

[18] Nicholas Matsakis and The Rust Developers. 2021. *Rust-Lang/polonius: Defines the Rust borrow checker.* Retrieved November 2, 2021 from https://github.com/rust-lang/polonius

[19] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (oct 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[20] Julian J McAuley and Jure Leskovec. 2012. Learning to discover social circles in ego networks.. In *NIPS*, Vol. 2012. Citeseer, 548–56.

[21] Frank McSherry and The Rust Developers. 2021. *Rust-Lang/datafrog: A Lightweight Datalog engine in rust.* Retrieved November 15, 2021 from https://github.com/rust-lang/datafrog

[22] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 305–315. https://doi.org/10.1145/1806596.1806631

[23] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40. https://doi.org/10.1145/3180143

[24] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 196–206. https://doi.org/10.1145/2892208.2892226

[25] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

[26] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages.* Ph. D. Dissertation. Carnegie-Mellon University, Pittsburgh, PA.

[27] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International*

Conference on Datalog Reloaded (Oxford, UK) *(Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

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

[29] David Van Horn and Harry G Mairson. 2008. Deciding k CFA is complete for EXPTIME. *ACM Sigplan Notices* 43, 9 (2008), 275–282. https://doi.org/10.1145/1411203.1411243

[30] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

[31] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 763–782.

[32] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442. https://doi.org/10.1038/30918

[33] Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. 2019. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982* (2019). https://doi.org/10.48550/arXiv.1903.00982