Higher-Order Structured Deduction

***Abstract*—State-of-the-art Datalog engines include expressive features such as ADTs (structured heap values), stratified aggregation and negation, various primitive operations, and the opportunity for further extension using FFIs. Current parallelization approaches for state-of-art Datalogs target shared-memory locking data-structures using conventional multi-threading, or use the map-reduce model for distributed computing. Furthermore, current state-of-art approaches cannot scale to formal systems which pervasively manipulate structured data due to their lack of indexing for structured data stored in the heap.**

In this paper, we describe a new approach to data-parallel structured deduction that involves a key semantic extension of Datalog to permit first-class facts and higher-order relations via defunctionalization, an implementation approach that exposes data parallelism uniformly both across sets of disjoint facts and over individual facts with nested structure. We detail a core language, DL$_s$, whose key invariant (subfact closure) ensures that all values are first-class facts and that all facts are first-class as values and threads of execution. We extend DL$_s$ to SLOG, a fully-featured language whose forms facilitate leveraging subfact closure to rapidly implement expressive, high-performance formal systems. We demonstrate SLOG by building a family of control-flow analyses from abstract machines, systematically, along with several implementations of classical type systems. We performed experiments on EC2, Azure, and Argonne's Theta at up to 1000 threads, showing orders-of-magnitude scalability improvements versus competing state-of-art systems.

***Index Terms*—declarative programming, semantics engineering, data-parallel deduction**

I. Structured Declarative Reasoning

Effective programming languages permit their user to write high-performance code in a manner that is as close to the shape of her own thinking as possible. A long-standing dream of our field has been to develop especially high-level *declarative* languages that help bridge this gap between specification and implementation. Declarative programming permits a user to provide a set of high-level rules and declarations that offer the sought-after solution as a latent implication to be materialized automatically by the computer. The semantics of a declarative language does the heavy lifting in operationalizing this specification for a target computational substrate—one with its own low-level constraints and biases. Modern computers provide many threads of parallel computation, may be networked to further increase available parallelism, and are increasingly virtualized within "cloud" services. To enable scalable cloud-based reasoning, the future of high-performance declarative languages must refine their suitability on both sides of this gulf: becoming both more tailored to human-level reasoning and to modern, massively-parallel, multi-node machines.

Logic-programming languages that extend Datalog have seen repeated resurgences in interest since their inception, each coinciding with new advances in their design and implementation. For example, Bddbddb [1] suggested that binary decision diagrams (BDDs) could be used to compress relational data while permitting fast algebraic operations such as relational join, but required *a priori* knowledge of efficient BDD-variable orderings to enable its compression, which proved to be a significant constraint. LogicBlox and Soufflé [2], [3] have since turned research attention back to semi-naïve evaluation over extensional representations of relations, using compression techniques sparingly (i.e., compressed prefix trees) and focusing on the development of high-performance shared-memory data structures. Soufflé represents the current state of the art at a low thread count, but struggles to scale well due to internal locking and its coarse-grained approach to parallelism. RadLog (i.e., BigDatalog) [4] has proposed scaling deduction to many-thread machines and clusters using Hadoop and the map-reduce paradigm for distributed programming. Unfortunately, map-reduce algorithms suffer from a (hierarchical) many-to-one collective communication bottleneck and are increasingly understood to be insufficient for leading high-performance parallel-computing environments [5], [6].

Most modern Datalogs are Turing-equivalent extensions, not simply finite-domain first-order HornSAT, offering stratified negation, algebraic data-types (ADTs), ad hoc polymorphism, aggregation, and various operations on primitive values. Oracle's Soufflé has added flexible pattern matching for ADTs, and Formulog [7] shows how these capabilities can be used to perform deductive inference of formulas; it seems likely future Datalogs will be used to implement symbolic execution and formal verification in a scalable, parallel manner.

In this paper, we introduce a new approach to simultaneously improve the expressiveness and data-parallelism of such deductive logic-programming languages. Our approach has three main parts: (1) a key semantic extension to Datalog, *subfacts* and a *subfact-closure property*, that is (2) implemented uniformly via ubiquitous fact interning, supported within relational algebra operations that are (3) designed from the ground-up to automatically balance their workload across available threads, using MPI to address the available data-parallelism directly. We show how our extension to Datalog permits deduction of structured facts, defunctionalization and higher-order relations, and more direct implementations of abstract machines (CEK, Krivine's, CESK), rich program analyses (k-CFA, m-CFA), and type systems. We detail our implementation approach and evaluate it against the best current Datalog systems, showing improved scalability and performance.

We offer the following contributions to the literature:

1) An architecture for extending Datalog to structured recursive data and higher-order relations, uniform with

respect to parallelism, allowing inference of tree-shaped facts which are indexed and data-parallel both horizontally (across facts) and vertically (over subfacts).
2) A formalism of our core language, relationship to Datalog, and equivalence of its model theoretic and fixed-point semantics, mechanized in Isabelle/HOL.
3) A high-performance implementation of our system, SLOG, with a compiler, REPL, and runtime written in Racket (10.6kloc), Python (2.5kloc), and C++ (8.5kloc).
4) An exploration of SLOG's applications in the engineering of formal systems, including program analyses and type systems. We include a presentation of the systematic development of program analyses from corresponding abstract-machine interpreters—the abstracting abstract machines (AAM) methodology—where each intermediate step in the AAM process may also be written using SLOG.
5) An evaluation comparing SLOG's performance against Soufflé and RadLog on EC2 and Azure, along with a strong-scaling study on the ALCF's Theta supercomputer which shows promising strong scaling up to 800 threads. We observe improved scaling efficiency and performance at-scale, compared with both Soufflé and RadLog, and better single-thread performance vs. Soufflé when comparing SLOG subfacts to Soufflé ADTs.

II. SLOG: DECLARATIVE PARALLEL DEDUCTION OF STRUCTURED DATA

For Datalogs used in program analysis, manipulation of abstract syntax trees (ASTs) is among the most routine tasks. Normally, to provide such ASTs as input to a modern Datalog engine, one first requires an external flattening tool that walks the richly structured syntax tree and produces a stream of flat, first-order facts to be provided as an input database. For example, the Datalog-based Java-analysis framework DOOP [8], [9], ported to Soufflé [2] in 2017, has a substantial preprocessor (written in Java) to be run on a target JAR to produce an input database of AST facts for analysis.

A key observation that initially motivated our work into this subject was that although this preparatory transformation is required to provide an AST as a database of first-order facts, the same work could not be done from within these Datalogs because it required generating unique identities (i.e., pointers to intern values) for inductively defined terms. In fact, any work generating ASTs as facts can not be done within Datalog itself but must be an extension to the language. Consider the pair of nested expressions that form an identity function:

```
(lam "x" (ref "x"))  --flattens-->  (= lam-id (lam "x" ref-id))
                                    (= ref-id (ref "x"))
```

Supplying unique intern values `lam-id` and `ref-id` as an extra column for those relations, and thus permitting them to be linked together, is the substance of this preparatory transformation. Our language, SLOG, proposes this interning behavior for facts be ubiquitous, accounted for at every iteration of relational algebra used to implement the fixed point.

In Soufflé, the language has more recently provided algebraic data-type (ADT) declarations and struct/record types for heap-allocated values which can be built up into ASTs or other such structured data. These datatypes must be declared and can then be used as $ expressions within rules; e.g., `$lam("x",$ref("x"))`. The downside of these ADTs in Soufflé is that they are not treated as facts for the purposes of triggering rules and are not indexed as facts, which would permit more efficient access patterns.

Our language, SLOG, respects a *subfact closure property*: every subfact is itself a first-class fact in the language and every top-level fact (and subfact) is a first-class value and has a unique identity (as an automatic column-0 value added to the relation). A clause `(foo x y)` in SLOG, always has an implied identity column and is interpreted the same as `(= _(foo x y))` if it's missing (where underscore is a wildcard variable). A nested pair of linked facts such as `(foo x (bar y) z)` is desugared as `(= _(foo x id z))` and `(= id (bar y))`. Thus we can represent an identity function's AST in SLOG as the directly nested fact and subfact `(lam "x" (ref "x"))`; under the hood this will be equivalent to two flat facts with a 0-column id provided by an interning process that occurs at the discovery of each new fact. In SLOG:

- Each structurally-unique fact/subfact has a unique intern-id stored in its 0 column so it may be referenced as another fact's subfact and treated as a first-class value.
- All data is at once a first-class fact (able to trigger rule evaluation), a first-class value (able to be referenced by other facts), and a first-class thread of execution (treated uniformly by a data-parallel MPI backend that dynamically distributes the workload spatially within, and temporally across, fixed-point iterations).

With subfacts as first-class citizens of the language (see Section III for details), able to trigger rules, various useful idioms emerge in which a subfact triggers a response from another rule via an enclosing fact (see Section III-A for extensions and idioms). Using these straightforward syntactic extensions enables a wide range of deduction and reasoning systems (see Section IV for a discussion of applications in program analyses and type systems). Because subfacts are first-class in SLOG, rules that use them will naturally force the compiler to include appropriate indices enabling efficient access patterns, and represent thread joins in the natural data-parallelism SLOG exposes. As a result, we are able to show a deep algorithmic and parallelism improvement over current state-of-art systems in the implementations of analyses we generate (see Section VI for our evaluation with apples-to-apples comparisons against Soufflé and RadLog). In some experiments, SLOG finishes in 4–8 seconds with Soufflé taking 1–3 hours—attesting to the importance of subfact indices. In others, we observe efficient strong-scaling on up to hundreds of threads, showing the value of our data-parallel backend.

III. Structurally Recursive Datalog

The core semantic difference between SLOG and Datalog is to allow structurally recursive, first-class facts. This relatively minor semantic change enables both enhanced expressivity (naturally supporting a wide range of Turing-equivalent idioms, as we demonstrate in Section IV) and anticipates compilation to parallel relational algebra (which interns all facts and distributes facts via their intern key). In this section, we present the formal semantics of a language we call Structurally Recursive Datalog (henceforth DL_s), the core language extending Datalog to which SLOG programs compile. All of the definitions related to DL_s have been formalized, and all of the lemmas and theorems presented in this section have been formally proven in Isabelle/HOL.

$$
\begin{array}{lcl}
\langle \textit{Prog} \rangle & ::= & \langle \textit{Rule} \rangle^* \\
\langle \textit{Rule} \rangle & ::= & \langle \textit{Clause} \rangle \leftarrow \langle \textit{Clause} \rangle^* \\
\langle \textit{Clause} \rangle & ::= & (\text{tag } \langle \textit{Subcl} \rangle^*) \\
\langle \textit{Subcl} \rangle & ::= & (\text{tag } \langle \textit{Subcl} \rangle^*) \mid \langle \textit{Var} \rangle \mid \langle \textit{Lit} \rangle \\
\langle \textit{Lit} \rangle & ::= & \langle \textit{Number} \rangle \mid \langle \textit{String} \rangle \mid \ldots
\end{array}
$$

Fig. 1: Syntax of DL_s; tag is a relation name.

Syntax: The syntax of DL_s is shown in Figure 1. As in Datalog, a DL_s program is a collection of Horn clauses. Each rule R contains a set of body clauses and a head clause, denoted by $\textit{Body}(R)$ and $\textit{Head}(R)$ respectively. DL_s (and SLOG) programs must also be well-scoped: variables appearing in a head clause must also be contained in the body.

We define a strict syntactic subset of DL_s, *DL* as the restriction of DL_s to clauses whose arguments are literals (i.e., $\langle \textit{Subcl} \rangle_{DL} ::= \langle \textit{Var} \rangle \mid \langle \textit{Lit} \rangle$). This subset (and its semantics) corresponds to Datalog.

Fixed-Point Semantics: The fixed-point semantics of a DL_s program P is given via the least fixed point of an *immediate consequence* operator $\textit{IC}_P : \textit{DB} \to \textit{DB}$. Intuitively, this immediate consequence operator derives all of the immediate implications of the set of rules in P. A database *db* is a set of facts ($\textit{db} \in \textit{DB} = \mathcal{P}(\textit{Fact})$). A fact is a literal s-expression:

$$\textit{Fact} ::= (\textit{tag } \textit{Val}^*) \qquad \textit{Val} ::= (\textit{tag } \textit{Val}^*) \mid \textit{Lit}$$

In Datalog, *Val*s are restricted to a finite set of atoms ($\textit{Val}_{DL} ::= \textit{Lit}$). To define IC_P, we first define the immediate consequence of a rule $IC_R : DB \to DB$, which supplements the provided database with all the facts that can be derived directly from the rule given the available facts in the database; we also define an *unroll* metafunction, which produces the (reflexive) set of all subfacts of a fact. The semantics uses *unroll* to ensure all subfacts are included in the database, a property we call *subfact closure*. This property is crucial to the semantics of DL_s (and SLOG), because in DL_s, each nested fact is a top-level fact as well, and not merely a carrier of structured data. Later sections (starting in section III-A) illustrate the importance of subfact closure to enables idioms that make SLOG more expressive.

The immediate consequence of a program is the union of the immediate consequence of each of its constituent rules, $\textit{IC}_P(db) \triangleq db \cup \bigcup_{R \in P} \textit{IC}_R(db)$. Observe that $\textit{IC}_P$ is monotonic over the the lattice of databases whose bottom element is the empty database. Therefore, if $\textit{IC}_P$ has any fixed points, it also has a least fixed point [10]. Iterating to this least fixed point directly gives us a naïve, incomputable fixed-point semantics for DL_s programs. Unlike pure Datalog, existence of a finite fixed point is not guaranteed in DL_s. This is indeed a reflection of the fact that DL_s is Turing-complete. The DL_s programs whose immediate consequence operators have no finite fixed points are non-terminating.

As discussed earlier, all SLOG databases must be subfact-closed (i.e. all subfacts are first-class facts). We can show that the least fixed point of the immediate consequence operator has the property that it is subfact-closed.

Lemma III.1. *(Formalized in Isabelle.) The least fixed point of $\textit{IC}_P$ is subfact-closed.*

The fixed-point semantics of Datalog is similar to DL_s, the only difference being that the *unroll* function is not required, as Datalog clauses do not contain subclauses.

Model Theoretic Semantics: The model theoretic semantics of DL_s closely follows the model theoretic semantics of Datalog, as presented in, e.g., [11]. The *Herbrand universe* of a DL_s program is the set of all of the facts that can be constructed from the relation symbols appearing in the program. Because DL_s facts can be nested, the Herbrand universe of any nontrivial DL_s program is infinite. One could, for example, represent natural numbers in DL_s using the zero-arity relation Zero and the unary relation Succ. The Herbrand universe produced by just these two relations, one zero arity and one unary, is inductively infinite.

A *Herbrand Interpretation* of a DL_s program is any subset of its Herbrand universe that is subfact-closed. In other words, if I is a Herbrand Interpretation, then $I = \bigcup\{\textit{unroll}(f) \mid f \in I\}$. For Datalog, the Herbrand Interpretation is defined similarly, but subfact-closure is elided.

Given a Herbrand Interpretation I of a DL_s program P, and a rule R in P, we say that R is true in I ($I \models R$) iff for every substitution of variables in R with facts in I, if all the body clauses with those substitutions are in I, so is the head clause of R with the same substitutions of variables. If every rule in P is true in I, then I is a *Herbrand model* for P. The denotation of P is the intersection of all its Herbrand models. We define $\mathbf{M}(P)$ to be the set of all Herbrand models of P, and $D(P)$ to be the denotation of P. We then have $D(P)$ to be the intersection of $I \in \mathbf{M}(P)$; such intersection is also a Herbrand model:

Lemma III.2. *(Formalized in Isabelle.) The intersection of a set of Herbrand models is also a Herbrand model.*

Unlike Datalog, DL_s programs may have Herbrand universes that are infinite, and also infinite Herbrand models. If a DL_s program has no finite Herbrand models, its denotation is infinite and so no fixed-point may be finitely calculated using the fixed-point semantics. We now relate the operational semantics of DL_s to its model-theoretic semantics.

Equivalence of Model-Theoretic and Fixed-Point Semantics: To show that the model-theoretic and fixed-point semantics of DL_s compute the same Herbrand model, we need to show that the least fixed point of the immediate consequence operator is equal to the intersection of all the Herbrand models for any program. We start by proving the following lemmas (proved in Isabelle; proofs elided for space).

Lemma III.3. *(Formalized in Isabelle.) Herbrand models of a DL_s program are fixed points of IC_P.*

Lemma III.4. *(Formalized in Isabelle.) Fixed points of the immediate consequence operator of a DL_s program that are subfact-closed are Herbrand models of the program.*

By proving that the Herbrand models and subfact-closed fixed points of the immediate consequence operator are the same, we conclude that the least fixed point of the immediate consequence operator IC_P (a subfact-closed database) is equal to the intersection of all its Herbrand models.

Theorem III.5. *(Formalized in Isabelle.) The model theoretic semantics and fixed point semantics of DL_s are equivalent.*

A. Key extensions to the core language

Subclauses, written with parentheses, are treated as top-level clauses whose id column value is unified at the position of the subclause. With subfacts, a common idiom becomes for a subfact to appear in the body of a rule, while its surrounding fact and any associated values are meant to appear in the head. For these cases, we use a ? clause, an s-expression marked with a "?" at the front to indicate that although it may appear to be a head clause, it is actually a body clause and the rule does not fire without this fact present to trigger it. The following rule says that if a `(ref x)` AST exists, then x is a free variable with respect to it. The rule

```
(free ?(ref x) x)
```

⇓ desugars to

```
[(= e-id (ref x)) --> (free e-id x)]
```

exposing that the ? clause is an implicit body clause. But if there are no body clauses apart from the ? clauses, the rule may be written without square braces and an arrow.

Two more rules are needed to define **free**:

```
[(=/= x y) (free Eb y)
   --> (free ?(lam x Eb) y)]
[(or (free Ef x) (free Ea x))
   --> (free ?(app Ef Ea) x)
```

The second of these shows another extension: disjunction in the body of a rule is pulled to the top level and splits the rule into multiple rules. In this case, there is both a rule saying that a free variable in `Ef` is free in the application and a rule saying that a free variable in `Ea` is free in the application.

Another core mechanism in SLOG is to put head clauses in position where a body clause is expected. Especially because an inner clause can be *responded to* by a fact surrounding it, or by rules producing that fact, being able to emit a fact on-the-way to computing a larger rule is what permits natural-deduction-style rules through a kind of rule splitting, closely related to continuation-passing-style (CPS) conversion [12]. A ! clause, under a ? clause or otherwise in the position of a body clause, is a clause that will be deduced as the surrounding rule is evaluated, so long as any ? clauses are satisfied and any subexpressions are ground (any clauses it depends on have been matched already). These ! clauses are intermediate head clauses; technically the head clauses of subrules, into which they are compiled internally.

```
(interp ?(do-interp (nat n)) n)
```

[NAT] $\dfrac{}{\texttt{(nat } n\texttt{)} \Downarrow n}$

```
[(interp !(do-interp e0) v0)
 (interp !(do-interp e1) v1)
 (+ v0 v1 v)
 --> ;-------- [plus]
 (interp ?(do-interp (plus e0 e1)) v)]
```

[PLUS] $\dfrac{e_0 \Downarrow v_0 \quad e_1 \Downarrow v_1 \quad v = v_0 + v_1}{\texttt{(plus } e_0\ e_1\texttt{)} \Downarrow v}$

Fig. 2: Natural-deduction-style reasoning in SLOG.

For example, in Figure 2, we write the natural deduction rules which let us prove statements such as `(plus (plus (nat 1) (nat 2)) (nat 1)) ⇓ 4`. We may understand these rules in a several equivalent ways; either (a) both the expression and value may be provided, to be "proved" according to these rules, or (b) the expression is provided as input, the answer (alongside a certificate) may be generated. Take note that our only creative decision in transliterating these inference rules into SLOG was to explicitly mark inputs and outputs using **interp** and **do-interp**, ? and !.

Another common use for a relation is as a function, or with a designated output column, deterministic or not, so SLOG also supports this type of access via { } inner clauses, which have their final-column value unified at the position of the curly-brace subclause. For example, the rule in Figure 2 could also have been written as below, with the clause `{+ v0 v1}` in place of variable `v`. This example illustrates that this syntax can also be used for built-in relations such as `+`.

```
[(interp !(do-interp e0) v0)
 (interp !(do-interp e1) v1)
 --> ;-------- [plus]
 (interp ?(do-interp (plus e0 e1))
         {+ v0 v1})]
```

IV. APPLICATIONS

The *abstracting abstract machines* (AAM) methodology [13], [14] proscribes a particular systematic application of abstract interpretation [15], [16] on abstract-machine operational semantics. AAM proposes key preparatory refactorings of an abstract machine, to remove direct sources of unboundedness through recursion, before more straightforward structural abstraction can be applied. A full development is elided for space, but every step may be written in SLOG, in a way that mirrors the natural inference rules one would write on paper. A CEK machine, such as Krivine's machine [17] or a call-by-value variant of it, can be developed by adding an explicit environment and continuation/stack. There are then two main sources of unboundedness in the CEK machine that make evaluation itself unbounded: environments and continuations. Environments contain closures which themselves contain environments; continuations are a stack of closures formed inductively (in practice using SLOG's list feature or by using nested

subfacts). AAM proposes threading each such fundamental source of unboundedness through a store, added in a normal store-passing transformation of the interpreter that might be used to add direct mutation or other effects to the language. Environments will map variables to addresses in the store, not to closures directly, and the stack will be store allocated at least once per function application so the stack may not grow indefinitely without the store's domain (addresses) likewise growing without bound. These two changes will permit us to place a bound on the addresses allocated, and thereby finitize the machine's state space as a whole.

Figure 3 shows a CEK interpreter that has been factored into **eval**, **apply**, and **ret** states, and subjected to a store-passing transformation. In SLOG we can instrument this with a timestamp or abstract contour, tracking the previous k callsites, by using a ! clause to request an updated timestamp in the **tick** function. We then allocate addresses specific to the variable name and current abstract contour, which at last gives us a finite address set and analysis, in this case tuned to implement k-call sensitivity or a classic k-CFA analysis.

```
;; Eval states
[(eval (ref x) env k _)
 -->
 (ret {store {env-map env x}} k)]
[(eval (lam x body) env k _)
 -->
 (ret (clo (lam x body) env) k)]
[(eval (app ef ea) env k c)
 -->
 (eval ef env
       (ar-k ea env (app ef ea) c k)
       c)]
;; Ret states
[(ret vf (ar-k ea env call c k))
 -->
 (eval ea env (fn-k vf call c k) c)]
[(ret va (fn-k vf call c k))
 -->
 (apply call vf va k c)]
[(ret v (kaddr e env))
 (store (kaddr e env) k)
  -->
 (ret v k)]
;; Apply states
[(apply call (clo (lam x Eb) env) va k c)
 -->
 (eval Eb env' (kaddr Eb env') c')
 (store (kaddr Eb env') k)
 (store (addr x c') va)
 (= env' (ext-env env x (addr x c')))
 (= c' {tick !(do-tick call c)})]

;; tick (tuning for 3-k-CFA)
(tick ?(do-tick call [h0 h1 _]) [call h0 h1])
```

Fig. 3: An AAM for global-store k-CFA in SLOG. This and a related m-CFA (not shown) are evaluated in Section VI.

To instantiate a monovariant control-flow analysis from this abstract interpreter, it would be enough to use the variable name itself as the address or to generate an address `(addr x)`. When the environment and store become finite, so does the number of possible states. Consider what happens, as the naturally relational **store** relation encoding the global store conflates multiple values at a single address for the same variable. Conflation in the store would lead naturally to nondeterminism in the model of control-flow.

A (potentially) more precise, though (potentially) more costly analysis specializes all control-flow and store points to a finite history of recent or enclosing calls, as in k-CFA. Such a *k-call-sensitive* analysis can be instantiated using a specific instrumentation and allocation policy, as can many others [18]. It requires an instrumentation to track a history of k enclosing calls, and then an *abstract allocation policy* that specializes variables by this call history at binding time. Such context-sensitive techniques are a gambit that the distinction drawn between variable `x` when bound at one call-site vs another will prove meaningful—in that it may correlate with its distinct values. Increasing the polyvariance allows for greater precision while also increasing the upper-bound on analysis cost. In a well-known paradox of program analyses, greater precision sometimes goes hand-in-hand with lower cost in practice because values that are simpler and fewer are simpler to represent [19]. At the same time, we use the polyvariant entry point of each function, its body and abstract contour—`(kaddr Eb c')`—to store allocate continuations as suggested by previous literature on selecting this address [20] so as to adapt to the value polyvariance chosen.

T-APP $\dfrac{\Gamma \vdash e_0 : T_0 \to T_1 \quad e_1 : T_0}{\Gamma \vdash (e_0\ e_1) : T_1}$

```
[(: !(ck Γ e0) (-> T0 T1))
 (: !(ck Γ e1) T0)
 -->;-------- T-App
 (: ?(ck Γ (app e0 e1)) T1)]
```

Fig. 4: STLC Application: natural deduction and SLOG

The original k-CFA uses higher-order environments, unlike analyses written for object oriented languages which implicitly use flat environments (objects) [21]. The corresponding CFA for functional languages is called m-CFA and is elided for space, but is also used in Section VI to evaluate SLOG. m-CFA has only the latest call history as a flat context. Instead of a per-variable address with a per-variable history tracked by a per-state environment, m-CFA stores a variable `x` at abstract contour `c` (i.e., abstract timestamp, instrumentation, 3-limited call-history) in the store at the address `(addr x c)`. This means at every update to the current flat context `c`, now taking the place of the environment, all free variables must be propagated into an address `(addr x c)`.

Taking a constructive logic interpretation allows us to understand SLOG's subfacts as proofs, naturally enabling structural type systems. While the rules of many traditional type systems readily transliterate into SLOG, we have focused on *algorithmic* type checking due to its simple decidability characteristics: synthesizing proofs is an area of future work. We have written an implementation of the Simply-Typed λ-calculus (STLC) and fragments of substructural and dependent type systems [22]. Figure 4 shows the apply rule from STLC, the **ck** relation forces the typechecking of subexpressions and—if they result in a valid derivation—allows building the derivation of the larger expression.

V. IMPLEMENTATION

We have implemented SLOG in a combination of Racket (the compiler, roughly 10,600 lines), C++ (runtime system and parallel RA backend; roughly 8,500 lines), Python (REPL and daemon; roughly 2,500 lines) and SLOG (60 lines of utilities).

A. Compiler

Our Racket-based compiler translates SLOG source code to C++ code that links against our backend. We follow the nanopass style, building a chain of passes, each producing increasingly-low-level intermediate representations (IRs) [23]. After parsing, *organize-pass* performs superficial simplifications and handles list splicing. SLOG's distribution paradigm is built upon binary joins, and thus *partitioning-pass* next breaks down bodies with multiple clauses into sequences of binary joins. While we elide a detailed discussion, partitioning into binary joins represents an algorithmic challenge, as there may be many ways to partition a set of clauses into a sequence of binary joins. We use a combination of heuristics and manual partitioning (we allow a $--$ syntax to indicate ordering) to avoid overmaterialization. Next, *split-selections-pass* identifies a set of indices for each relation determined by an anlysis of its access patterns, and adds administrative rules to replicate indices of the same relation. The last two passes are calculation of *strongly-connected components (SCCs)* and *incrementalization* (anticipating semi-naïve evaluation), which further divides representation of each relation-index into **new**, **delta**, and **total** versions to support incrementalized evaluation; the backend merges **delta** into **total** after each iteration—**new** becomes **delta**. What is novel in our compiler is that it supports nested clauses, !, ?, $--$, and $\{\}$, via first-class facts, which are then supported all the way down to the implementation as relational algebra and within those operations as well, described next. Besides these novelties, the compiler follows a well-established implementation approach used by other modern Datalogs. The other key difference from Soufflé's compiler is that SLOG does not currently support shared indicies and heterogenous k-ary joins as Soufflé does due to its approach to data-parallelism via MPI. If we approached parallelism as Soufflé does, via shared-memory data structures on a single node, then both k-ary joins and prefix-based index sharing would be possible for SLOG as well.

B. Backend

Our parallel relational-algebra backend is designed to flexibly scale up to large-scale multi-node HPC clusters, exploiting as many threads of parallelism as are available using the Message-Passing Interface (MPI). Based on the bulk-synchronous-processing protocol and built using the `MPI-everywhere` approach [24], [25], our parallel RA framework addresses the problem of partitioning and balancing workload across processes by using a two-layered distributed hash-table based on Balanced Parallel Relational Algebra (BPRA) [26]. In order to materialize newly generated facts within each iteration, and thus facilitate *iterated* RA (in a fixed-point loop), an all-to-all data exchange phase is used at every iteration. Figure 5 shows a schematic diagram of all the phases (including the added interning phase) in the context of an incrementalized TC computation. There are three primary phases in the backend: (1) RA kernel computation, (2) all-to-all communication and (3) local insertion.

RA kernel computation: The two-layered distributed approach, with local hash-based joins and hash-based distribution of relations, is a foundational method to distribute RA-kernel (primarily join) operations over many nodes in a networked cluster computer. This algorithm involves partitioning relations by their join-column values so that they can be efficiently distributed to participating processes [27]. The main insight behind this approach is that for each tuple in the outer relation, all relevant tuples in the inner relation must be hashed to the same MPI process or node, permitting joins to be performed locally on each process.

The challenge with parallel workload partitioning is to ensure every process is responsible for similar-sized workloads. The key issue in enforcing this load balance is to deal with *inherently imbalanced* data coming from key-skewed relations. To ensure uniform load across processes, we have built on previous approaches [26], [28] that use dynamic mitigation of load-imbalance. The approach [26] uses a two-layered distributed hash-table to partition tuples over a fixed set of *buckets*, and, within each bucket, to a dynamic set of *subbuckets* which may vary across buckets. Each tuple is assigned to a bucket based on a hash of its key-column values, but within each bucket tuples are hashed on non-join-column values, assigning them to a local subbucket, then mapped to an MPI process. Within subbuckets, tuples are stored in B-trees, organized by key-column values. The first step in a join operation is an *intra-bucket communication* phase within each bucket so that every subbucket receives all tuples for the outer relation across all subbuckets (while the inner relation only needs tuples belonging to the local subbucket). Following this, a *local join* operation (with any necessary projection and renaming) is performed in every subbucket.

All-to-all communication: To enable iterated parallel RA (in a fixed-point loop), processes must engage in a non-uniform all-to-all inter-process shuffle of generated tuples to the process managing that tuple in the output index. This data exchange is performed to *materialize* the output tuples generated from the local compute phase (where RA kernels are executed) to their appropriate processes (based on their bucket-subbucket assignment). Materializing a tuple in an output relation (resulting from an RA operation) involves hashing on its join and non-join columns to find its bucket and sub-bucket (respectively), and then transmitting it to the process that maintains that bucket/sub-bucket.

The overall scalability of the RA backend relies on the scalability of the all-to-all inter-process data exchange phase. However, all-to-all is notoriously difficult to scale [29]–[31]—largely because of the *quadratic* nature of its workload. We address this scaling issue by adopting recent advancements [32] that optimize non-uniform all-to-all communication by extending the $\log$-time Bruck algorithm [31], [33], [34] for non-uniform all-to-all workloads.

Local inserts: After all-to-all data exchange, every process receives a set of possibly-new facts that must be materialized to be used as input in the subsequent iteration of the fixed-point loop. Local insert is a two-step process involving

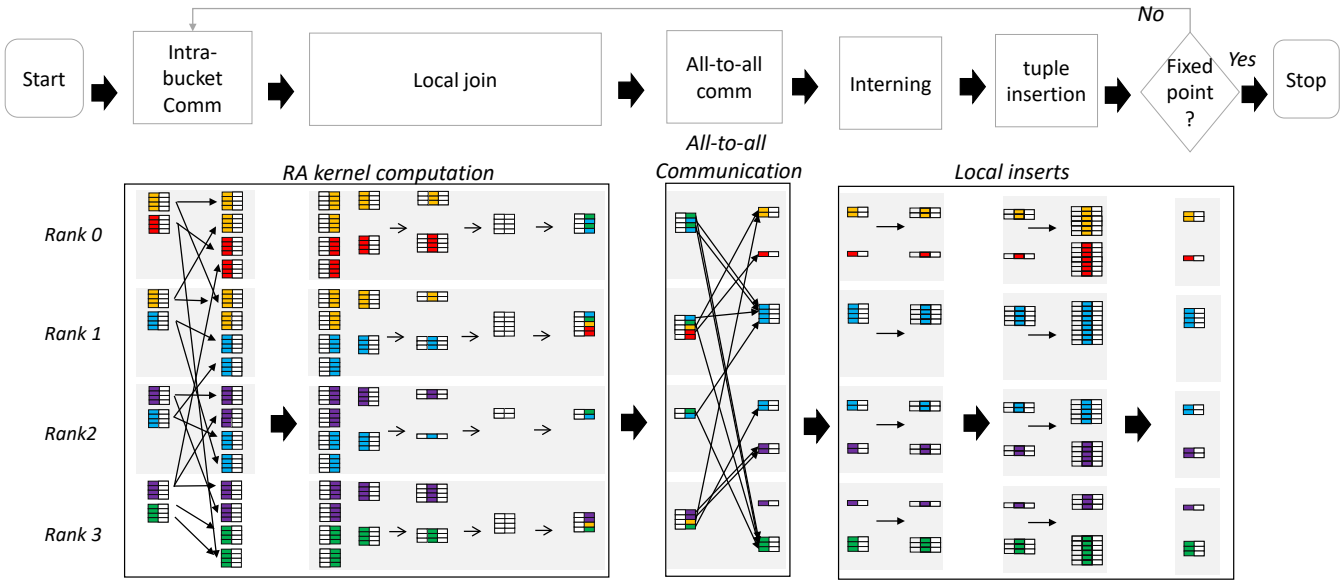

Fig. 5: An illustration of the main phases of our parallel RA backend.

interning and inserting newly generated facts in the appropriate version of a relation (delta and total). Interning assigns a unique 64-bit key to every fact, and in order to scale this process, it must be is performed in an embarrassingly parallel manner without the need for any synchronization among processes. This is done by reserving the first 16 bits of the key for unique sub-buckets ids, and the remaining 48-bits for facts. Since, a sub-bucket is never split across a process, reserving 16 bits for sub-bucket ids ensures that globally unique intern keys can be created concurrently across processes. The fact-id component of the intern key is created by a bump pointer, which ensures that locally all facts receive a unique key. A check is performed by a global operation that checks the size of all relations across all processes, and if all sizes remains unchanged across a subset of iterations, this indicates that a fixed-point has been attained and the program can terminate.

VI. Evaluation

We aimed to measure and evaluate SLOG's improved indexing and data parallelism, using three sets of performance benchmarks (PBs):

- **PB1** (Section VI-A) How does SLOG compare against other systems designed for performance and parallelism on traditional Datalog workloads (without ADTs): Soufflé and RadLog?
- **PB2** (Section VI-B) How do SLOG subfacts perform against Soufflé ADTs in the context of the m-CFA and k-CFA benchmarks developed in Section 4.
- **PB3** (Section VI-C) How well can SLOG scale to many threads on a supercomputer?

We evaluated **PB1** and **PB2** by running a set of experiment on large cloud machines from Amazon AWS and Microsoft Azure. For **PB1**, we ran a set of strong scaling experiments of transitive closure on large graphs, picking transitive closure as an exemplary problem to measure end-to-end throughput of deductive inference at scale. For **PB2**, we measure the performance of the implementation of our k and m-CFA analyses from Section IV compared to an equivalent implementation in Soufflé using abstract datatypes (ADTs). We answer **PB3** by running experiments on the Theta supercomputer at Argonne National Supercomputing Lab, scaling a control-flow analysis for the λ-calculus to 1000 threads on Argonne's Theta.

A. Transitive Closure

We sought to compare SLOG's full-system throughput on vanilla Datalog against two comparable production systems: Soufflé and Radlog. Soufflé is engineered to achieve the best-known performance on unified-memory architectures, and supports parallelism via OpenMP. Radlog is a Hadoop-based successor to the BigDatalog deductive inference system, which uses Apache Spark to perform distributed joins at scale [35]. We originally sought to compare SLOG directly against BigDatalog, but found it does not support recent versions of either Spark or Java (being built to target Java 1.5). Under direction of BigDatalog's authors, we instead used Radlog, which is currently under active development and runs on current versions of Apache Spark.

We performed comparisons on an Standard_M128s instance rented from Microsoft Azure [36]. The node used in our experiments has 64 physical cores (128 threads) running Intel Xeon processors with a base clock speed of 2.7GHz and 2,048GB of RAM. To directly compare SLOG, Soufflé, and Radlog, we ran each on the same machine using 15, 30, 60, and 120 threads. We ran SLOG using OpenMPI version 4.1.1 and controlled core counts via `mpirun`. We compiled Soufflé from its Git repository, using Soufflé's compiled mode to compile each benchmark separately to use the requisite number of threads before execution. Radlog runs natively on Apache

TABLE I: Single-node TC Experiments

Graph Properties				Time (s) at Process Count			
Name	Edges	|TC|	System	15	30	60	120
FB-MEDIA	206k	96M	SLOG	62	40	21	**18**
			Soufflé	35	33	34	37
			Radlog	254	295	340	164
RING10000	10k	100B	SLOG	363	218	177	**115**
			Soufflé	149	143	140	141
			Radlog	464	646	852	1292
SUITESPARSE	412k	3.35T	SLOG	–	1,593	908	**671**
			Soufflé	1,417	1,349	1,306	1,282
			Radlog	–	–	–	–

Spark, which subsequently runs on Hadoop. To achieve a fair comparison against Soufflé and SLOG, we ran Radlog using Apache Spark configured in local mode; Spark's local mode circumvents the network stack and runs the application directly in the JVM. We used three large graphs shown the first column of Table I: FB-MEDIA is media-related pages on Facebook, RING10000 is ring graph of 10,000 nodes, and SUITESPARSE is from the UF Sparse Matrix Collection [37]. We configured Radlog according the directions on its website, experimenting with a variety of partitions (used for shuffling data between phases) to achieve the best performance we could. Ultimately, we used three times as many partitions as available threads, except for RING10000, for which we found higher partition counts caused significantly lower performance.

Table I details the results of our single-node performance comparisons in seconds for each thread count, where each datapoint represents the best of three runs (lower is better). Experiments were cut off after 30 minutes. In every case, we found that SLOG produced the best performance overall at 120 threads, even compared to Soufflé's best time. However, as expected, our results indicate that Soufflé outperforms SLOG at lower core counts (below 60). Soufflé implements joins with tight loops in C++, and (coupled with its superior single-node datastructures) this allows Soufflé to achieve better performance than either SLOG or Radlog at lower core counts. We found that Radlog did not scale nearly as well as either Soufflé or SLOG. We expected this would be the case: both SLOG and Soufflé compile to C++. By comparison, Radlog's Spark-based architecture incurs significant sequential overhead due to the fact that it is implemented on top of the JVM and pays a per-iteration penalty by using Hadoop's aggregation and shuffling phase. SLOG also incurs sequential overhead compared to Soufflé due to its distributing results after every iteration, though results indicate that our MPI-based implementation helps ameliorate this compared to Radlog.

B. AAMs and CFAs

Next, we sought to benchmark the analyses described in Section IV at scale versus an equivalent implementation using ADTs in Soufflé (we ignore Radlog in this comparison due to its lack of support for ADTs). We developed a SLOG analysis for each of six different polyvariance choices: three k-CFA (k=3,4,5) and three m-CFA (m=10,12,15) implementations. We then systematically derived six different Soufflé-based variants. We tested each of these on six different term sizes, drawn from a family of worst-case terms identified in David Van Horn's thesis [38]. We then benchmark both SLOG and Soufflé on each of these instances and report upon their results, scalability, and broad trends which we observed. Critically, our Soufflé code is an exact port of the SLOG code we used, except that $-ADT values are used in place of subfacts and the analysis was designed in the first place to avoid the need for these subfacts to trigger rule evaluation as they can in SLOG.

Experimental Setup: The experiments described in this subsection were run on a c6a.metal instance rented from Amazon Web Services (AWS), including 192 hardware threads (when run using the .metal instance types) and 384 GiB of RAM. Because both SLOG and Soufflé are designed to enable parallelism, we ran each experiment at two distinct scales: 8 and 64 processes (threads). SLOG was invoked using mpirun, and Soufflé's compiled mode was used to produce a binary which was subsequently run and timed using GNU time. We did not systematically measure memory usage; recent microbenchmarks for TC report 3-5x memory blowup versus Soufflé. We record and report the best of three runs for each experiment (imposing a four hour cutoff). To avoid an unfair comparison to Soufflé with respect to on-disc ADT materialization (which may explode due to linearization of linked data), our Soufflé implementation does not output control-flow points or store directly—instead we measure and report their size using the sum aggregate (built in to Soufflé).

Results: We report our results in Table II. Each of six distinct analysis choices is shown along the left side. Along rows of the table, we show experiments for a specific combination of analysis, precision, and term size. We detail the total number of iterations taken by the SLOG backend, along with control-flow points, store size, and runtime at both eight and 64 processes for SLOG and Soufflé. Times are reported in minutes / seconds form; several runs of Soufflé took under 1 second (which we mark with <0:01), and ☃ indicates that the run timed out after four hours.

Inspecting our results, we observed several broad trends. First, as problem size increases, SLOG's runtime grows less-rapidly than Soufflé's. This point may be observed by inspecting runtimes for a specific set of experiments. For example, 10-m-CFA with term size 200 took SLOG 26 seconds, while Soufflé's run took 56 seconds. Doubling the term size to 400 takes 104 seconds in SLOG, but 398 seconds in Soufflé—a slowdown of $4\times$ in SLOG, compared to a slowdown of $7\times$ in Soufflé. A similar trend happens in many other experiments, e.g., 15 minutes to over three hours for Soufflé ($13\times$ slowdown) vs. 2 to 4 seconds ($2\times$ slowdown) in SLOG's runtime on 5-k-CFA. Inspecting the output of Soufflé's compiled C++ code for each experiment helped us identify the source of the slowdown. For example, the following rule for return joins on the subfact `$KAddr(e, env)`—because Soufflé does not index subfacts, a scan-then-filter approach is used.

```
ret(v,sto,k) :- ret(v,sto,$KAddr(e, env)),
```

TABLE II: Control-Flow Analysis Experimental Results: Slog vs. Soufflé

	Size	Iters	Cf. Pts	Sto. Sz.	8 Processes		64 Processes	
					Slog	Soufflé	Slog	Soufflé
3-k-CFA	8	1,193	98.1k	23.4k	00:01	01:07	0:02	00:15
	9	1,312	371k	79.8k	00:02	14:47	0:03	02:56
	10	1,431	1.44M	291k	00:06	☃	0:05	45:49
	11	1,550	5.68M	1.11M	00:27	☃	0:16	☃
	12	1,669	22.5M	4.32M	02:14	☃	1:07	☃
	13	1,788	89.8M	17.0M	12:17	☃	5:08	☃
4-k-CFA	9	1,363	312k	65k	00:01	14:38	00:03	02:08
	10	1,482	1.2M	229k	00:05	☃	00:05	40:30
	11	1,601	4.69M	853k	00:20	☃	00:13	☃
	12	1,720	18.5M	3.28M	01:40	☃	00:53	☃
	13	1,839	73.8M	12.9M	08:44	☃	03:58	☃
	14	1,958	294M	50.8M	60:53	☃	35:46	☃
5-k-CFA	9	1,429	203k	50.7k	00:02	05:30	0:03	01:15
	10	1,548	757k	167k	00:04	65:20	0:04	015:08
	11	1,667	2.91M	597k	00:13	☃	0:08	196:06
	12	1,786	11.4M	2.24M	00:56	☃	0:27	☃
	13	1,905	45.2M	8.69M	04:38	☃	2:00	☃
	14	2,024	180M	34.2M	25:14	☃	9:58	☃

	Size	Iters	Cf. Pts	Sto. Sz.	8 Processes		64 Processes	
					Slog	Soufflé	Slog	Soufflé
10-m-CFA	50	6,120	21k	656k	00:02	00:02	00:10	00:01
	100	11,670	42k	2.78M	00:07	00:09	00:20	00:04
	200	22,770	86.5k	11.4M	00:26	00:56	00:42	00:23
	400	44,970	173k	46.4M	01:44	06:26	01:38	01:56
	800	89,370	348k	187M	07:35	45:22	04:21	09:33
	1600	178,170	698k	750M	32:56	☃	14:36	1:02:35
12-m-CFA	25	3,559	17k	385k	00:01	<:01	0:06	<:01
	50	6,434	36k	1.89M	00:04	00:03	0:11	00:03
	100	12,184	74k	8.82M	00:16	00:24	0:23	00:10
	200	23,684	151k	34.6M	01:10	02:37	0:53	00:55
	400	46,684	305k	142M	05:04	18:39	2:23	04:12
	800	92,684	611k	574M	22:46	2:38:22	7:28	24:58
15-m-CFA	12	2,211	14.4k	137k	00:01	<:01	0:04	<:01
	24	3,591	35.9k	1.44M	00:03	00:02	0:06	00:01
	48	6,351	78.6k	8.29M	00:14	00:15	0:14	00:07
	96	11,871	164k	38.9M	01:08	01:41	0:36	00:36
	192	22,911	335k	168M	05:15	12:10	1:49	02:51
	384	44,991	678k	697M	24:10	1:32:35	6:45	16:30

```
kont_map($KAddr(e, env),k).
```

For a fixed problem size, we found that Soufflé and SLOG both scaled fairly well. Soufflé consistently performed well on small input sizes; additional processes did not incur slowdowns, and Soufflé's efficiency was generally reasonable (roughly 50%) when algorithmic scalability did not incur slowdowns. For example, in 3-k-CFA (n=8), Soufflé took 67 seconds at 8 processes, and 15 seconds at 64 processes. SLOG's parallelism doesn't outweigh communication overhead on smaller problems, particularly on problems with high iteration count and low per-iteration work. As problem size increases, our SLOG implementations show healthy scalability; efficiency grows as problem size grows (e.g., 24:10 to 6:45 on 15-m-CFA/384, 22:46 to 7:28 on 12-m-CFA/800).

C. Multi-node Scaling Experiments

In recent years, several cloud providers have launched MPI-capabable HPC nodes for their cloud services and significantly upgraded their network interconnects [39], [40]. Alongside related advances in cloud architectures and MPI implementations, this move signals the ability for massively parallel analytics to be deployed as a service in the near future. In this spirit, we also conduct some preliminary strong-scaling experiments on the Theta supercomputer. For this we used a fully monomorphized m-CFA (distinct from the m-CFA used in the previous subsection)—experiments we had ready to go when our allocation on Theta became possible. The Theta Supercomputer [41] at the Argonne Leadership Computing Facility (ALCF) of the Argonne National Laboratory is a Cray machine with a peak performance of 11.69 petaflops. It is based on the second-generation Intel Xeon Phi processor and is made up of 281,088 compute cores. It has an aggregate of 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM and 10 PiB of online disk storage. The supercomputer has a Dragonfly network topology uses the Lustre filesystem.

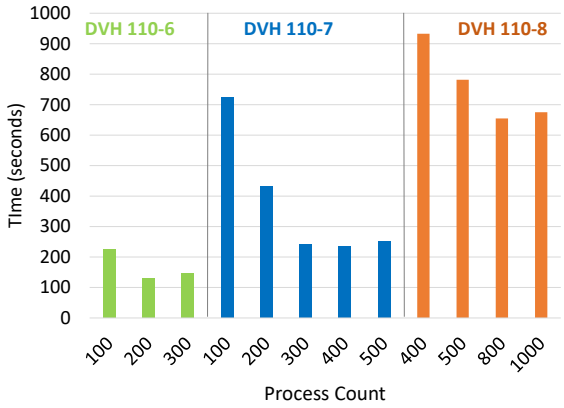

Fig. 6: Strong-scaling experiments on Theta.

We ran three sets of m-CFA worst-case experiments with 110 terms and m = 6, 7 and 8. The results of the three sets of experiments, referred to as `dvh-110-6`, `dvh-110-7` and `dvh-110-8` are plotted in Figure 6. For all three set of experiments we observe improvement in performance with increase in process counts, until maximum efficiency is attained, after which performance degrades with increasing process counts, due to communication overhead and workload starvation. In general, for a given workload (problem size), we observe a range of processes that exhibit healthy scalability. `dvh-110-6` shows a near 100% scaling efficiency (2× speedup while increasing the process count from 100 to 200), performance however drops when the number of processes is increased to 300. Similarly, `dvh-110-7` shows a 75% scaling efficiency (3× speedup when the process count is increased from 100 to 400), and `dvh-110-8` shows a 71% scaling efficiency (1.42× speedup going 400 to 800).

VII. Related and Future Work

Distributed Datalog: There have been significant implementation efforts to scale Datalog-like languages to large clusters of machines. For example, BigDatalog [4], Distributed

SociaLite [42], Myria [43], and Radlog [44] all run on Apache Spark clusters (servers networked together via commodity switches within a datacenter). Extending Spark's architecture with recursive queries (and aggregates), these frameworks scale to large datasets typical of Spark queries. SLOG differs from these systems in two primary ways. First, compared to SLOG's MPI-based implementation, Apache Spark's framework-imposed overhead is increasingly understood to be a bottleneck in scalable data analytics applications, with several authors noting order-of-magnitude improvements when switching from Spark to MPI [5], [6], [45]. Second, none of the aforementioned systems support first-class subfacts; for example, while Radlog can compute the length of the shortest path from a specific point, it cannot materialize the path per-se. Recently, Radlog's authors have created DCDatalog, a parallel Datalog which targets shared-memory SMP architectures and demonstrate a $10\times$ runtime speedup compared to Soufflé on a machine with four eight-core processors and 256GB of RAM. Unfortunately, DCDatalog is not open-source, and we have not been able to obtain a copy for evaluation; we believe it is difficult to interpret DCDatalog's results compared to SLOG and Soufflé, as their paper notes "Soufflé does not allow aggregates in recursion, and thus it must use a stratified query that results in very poor performance" for several evaluation queries. Last, Nexus (also closed-source) has claimed a significant performance boost (up to $4\times$) compared to BigDatalog by using Apache Flink, a data-flow processing language [46].

Datalog Extensions: Noting the first-order nature of vanilla Datalog—and often inspired by Datalog's efficient semi-naïve evaluation strategy—there has been extensive work in extending Datalog with additional expressive power [7], [47]–[50]. Flix augments Datalog with lattices [47], [48], but is not specifically focused on efficient compilation; recently, Ascent is a macro-based implementation of Datalog in Rust which includes lattices and shows orders-of-magnitude runtime improvements versus Flix [51]. Similarly, Datafun is a pure functional language which computes fixed points of monotone maps on semilattices [49], [50]. Compared to SLOG, Datafun's evaluation strategy is top-down and based on the λ-calculus; the authors have recently studied semi-na ive evaluation of Datafun upon work on the incremental λ-calculus [52], [53]. SLOG's primary difference from this work is that it is based on DL_s rather than the λ-calculus; because of this, semi-naïve evaluation for functions in SLOG (using defunctionalization) requires no extra logic.

Datalog + Constraints: An increasingly-popular semantic extension to Datalog is adding first-class constraints [7], [54]–[56]. These constraints typically allow interfacing with an SMT solver, potentially in a loop with subsequent analysis [7]. Formulog includes ADTs and first-order functions over ADTs, allowing Turing-equivalent to build formulas of arbitrary size to be checked by Z3 [7]; we anticipate SLOG will perform well compared to Formulog when subfact-indexing is of concern, though by Amdahl's law this effect will be smaller in code whose computation is dominated by calls to Z3. Similarly, Rosette efficiently compiles solver-aided queries to efficient implementations using host language constructs and a symbolic virtual machine (SVM) [55], [56]. SLOG is largely orthogonal to these systems, which focus on shared-memory implementations and are not primarily concerned with parallelization. We have transliterated proof-of-concept examples from both of these projects into SLOG, but it is currently impossible to call Z3 from SLOG as doing so would require all facts be resident on a single node. Semantically, SLOG is more directly comparable to constrained HornSAT or existential fixed-point logic, which have attracted recent interest for their application to program verification [57]–[60]. DL_s can express constrained HornSAT problems as long as a decision procedure for the background logic is available; we plan to study usage of SLOG for CHCs in subsequent work.

Parallel Program Analyses: Given the algorithmic complexity intrinstic to large-scale program analyses, there has been significant interest in its parallelization [2], [8], [61]–[64] or implementation using special-purpose datastructures [65]–[69]. There are a variety of fundamental approaches to scalability; for example, summarization-based analyses (such as Saturn [62], [63]) are attractive due to the task-level parallelism they expose. Much work in scaling program analysis has focused on context-insensitive analyses—wherein task-level parallelism is more directly exploitable. The goals of SLOG are most closely related to current efforts on scaling rich, whole-program context-sensitive analyses using deductive inference [2], [3], [70].

Parallel Relational Algebra: SLOG's backend builds upon recent successes in balanced, parallel relational algebra (BPRA) and follow-up work on compilation of vanilla Datalog to parallel relational algebra kernels [71]–[73]. However, that work focuses mainly on the low-level implementation of relational algebra kernels rather than a unified programming language, compiler, and runtime.

VIII. Conclusion

In this work we extended Datalog with subfacts, explicating a core language, DL_s, which supports subfacts as first-class facts, values, and threads. This straightforward semantic extension enables several key innovations, including the expression of higher-order functions via defunctionalization, subfact indexing, and a rich connection to constructive logic via the proofs-as-programs interpretation. We have implemented these ideas in SLOG, a fully featured language for data-parallel structured deduction, using a new MPI-based relational algebra backend, and we demonstrate its application to operational semantics, abstract interpreters, and formal systems broadly. Our experiments show that SLOG is competitive with, or (when programming with algebraic data) orders-of-magnitude faster than state-of-the-art systems, showing improved data-parallelism and scaling up to hundreds of cores on large unified machines and high-performance clusters.

References

[1] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005, pp. 97–118.

[2] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, "Porting DOOP to soufflé: a tale of inter-engine portability for datalog-based analyses," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 2017, pp. 25–30.

[3] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 422–430.

[4] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1135–1149. [Online]. Available: https://doi.org/10.1145/2882903.2915229

[5] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proc. VLDB Endow.*, vol. 10, no. 8, p. 901–912, apr 2017. [Online]. Available: https://doi.org/10.14778/3090163.3090168

[6] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, vol. 53, pp. 121–130, 2015, iNNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915017895

[7] A. Bembenek, M. Greenberg, and S. Chong, "Formulog: Datalog for smt-based static analysis," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–31, 2020.

[8] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 243–262. [Online]. Available: http://doi.acm.org/10.1145/1640089.1640108

[9] ——, "Strictly declarative specification of sophisticated points-to analyses," *SIGPLAN Not.*, vol. 44, no. 10, p. 243–262, Oct. 2009. [Online]. Available: https://doi.org/10.1145/1639949.1640108

[10] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications." *Pacific journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[11] S. Ceri, G. Gottlob, L. Tanca *et al.*, "What you always wanted to know about datalog(and never dared to ask)," *IEEE transactions on knowledge and data engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[12] A. W. Appel, *Compiling with continuations*. Cambridge university press, 2007.

[13] M. Might, "Abstract interpreters for free," in *International Static Analysis Symposium*, ser. SAS '10. Springer, 2010, pp. 407–421.

[14] D. Van Horn and M. Might, "Abstracting abstract machines," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10. New York, NY, USA: ACM, 2010, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/1863543.1863553

[15] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252. [Online]. Available: http://doi.acm.org/10.1145/512950.512973

[16] ——, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1979, pp. 269–282.

[17] J.-L. Krivine, "A call-by-name lambda-calculus machine," *Higher-order and symbolic computation*, vol. 20, no. 3, pp. 199–207, 2007.

[18] T. Gilray, M. D. Adams, and M. Might, "Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '16. New York, NY, USA: ACM, 2016, pp. 407–420. [Online]. Available: http://doi.acm.org/10.1145/2951913.2951936

[19] A. K. Wright and S. Jagannathan, "Polymorphic splitting: An effective polyvariant flow analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 1, pp. 166–207, 1998.

[20] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. Van Horn, "Pushdown control-flow analysis for free," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 691–704. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837631

[21] M. Might, Y. Smaragdakis, and D. Van Horn, "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*, 2010, pp. 305–315.

[22] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[23] A. W. Keep and R. K. Dybvig, "A nanopass framework for commercial compiler development," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 343–350. [Online]. Available: https://doi.org/10.1145/2500365.2500618

[24] M. P. Forum, "Mpi: A message-passing interface standard," 1994.

[25] R. Zambre, D. Sahasrabudhe, H. Zhou, M. Berzins, A. Chandramowlishwaran, and P. Balaji, "Logically parallel communication for fast mpi+ threads applications," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[26] S. Kumar and T. Gilray, "Distributed relational algebra at scale," in *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019.

[27] P. Valduriez and S. Khoshafian, "Parallel evaluation of the transitive closure of a database relation," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 19–42, Feb. 1988.

[28] S. Kumar and T. Gilray, "Load-balancing parallel relational algebra," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 288–308.

[29] R. Kumar, A. Mamidala, and D. K. Panda, "Scaling alltoall collective on multi-core systems," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[30] D. S. Scott, "Efficient all-to-all communication patterns in hypercube and mesh topologies," in *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*. IEEE Computer Society, 1991, pp. 398–399.

[31] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[32] K. Fan, T. Gilray, V. Pascucci, X. Huang, K. Micinski, and S. Kumar, "Optimizing the bruck algorithm for non-uniform all-to-all communication," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 172–184.

[33] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[34] J. L. Träff, A. Rougier, and S. Hunold, "Implementing a classic: Zero-copy all-to-all communication with mpi datatypes," in *Proceedings of the 28th ACM international conference on Supercomputing*, 2014, pp. 135–144.

[35] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo, "Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 467–484. [Online]. Available: https://doi.org/10.1145/3299869.3324959

[36] Microsoft, "Azure — m-series," https://docs.microsoft.com/en-us/azure/virtual-machines/m-series, accessed: 2021-11-19.

[37] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[38] D. V. Horn, "The complexity of flow analysis in higher-order languages," Ph.D. dissertation, Brandeis University, 2009.

[39] E. Burness, "Introducing the new hbv2 azure virtual machines for high-performance computing," 2019, https://azure.microsoft.com/en-us/blog/introducing-the-new-hbv2-azure-virtual-machines-for-high-performance-computing/; accessed August 11, 2019.

[40] "AWS ParallelCluster," https://aws.amazon.com/hpc/parallelcluster/, accessed: 2022-06-01.

[41] S. Parker, V. Morozov, S. Chunduri, K. Harms, C. Knight, and K. Kumaran, "Early evaluation of the cray xc40 xeon phi system 'theta'at

argonne," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[42] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *Proc. VLDB Endow.*, vol. 6, no. 14, p. 1906–1917, sep 2013. [Online]. Available: https://doi.org/10.14778/2556549.2556572

[43] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu, "Demonstration of the myria big data management service," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 881–884. [Online]. Available: https://doi.org/10.1145/2588555.2594530

[44] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo, "Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 467–484. [Online]. Available: https://doi.org/10.1145/3299869.3324959

[45] D. S. Kumar and M. A. Rahman, "Performance evaluation of apache spark vs mpi: A practical case study on twitter sentiment analysis," *Journal of Computer Science*, vol. 13, no. 12, pp. 781–794, Dec 2017. [Online]. Available: https://thescipub.com/abstract/jcssp.2017.781.794

[46] M. Imran, G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl, "Fast datalog evaluation for batch and stream graph processing," *World Wide Web*, vol. 25, no. 2, p. 971–1003, mar 2022. [Online]. Available: https://doi.org/10.1007/s11280-021-00960-w

[47] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: A declarative language for fixed points on lattices," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 194–208, 2016.

[48] M. Madsen and O. Lhoták, "Safe and sound program analysis with flix," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 38–48. [Online]. Available: https://doi.org/10.1145/3213846.3213847

[49] M. Arntzenius and N. R. Krishnaswami, "Datafun: A functional datalog," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 214–227. [Online]. Available: https://doi.org/10.1145/2951913.2951948

[50] M. Arntzenius and N. Krishnaswami, "Seminaïve evaluation for a higher-order functional language," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, dec 2019. [Online]. Available: https://doi.org/10.1145/3371090

[51] A. Sahebolamri, T. Gilray, and K. Micinski, "Seamless deductive inference via macros," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 77–88. [Online]. Available: https://doi.org/10.1145/3497776.3517779

[52] P. G. Giarrusso, Y. Régis-Gianas, and P. Schuster, "Incremental λ-calculus in cache-transfer style," in *Programming Languages and Systems*, L. Caires, Ed. Cham: Springer International Publishing, 2019, pp. 553–580.

[53] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann, "A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 145–155. [Online]. Available: https://doi.org/10.1145/2594291.2594304

[54] M. Madsen and O. Lhoták, "Fixpoints for the masses: Programming with first-class datalog constraints," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428193

[55] E. Torlak and R. Bodik, "Growing solver-aided languages with rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–152. [Online]. Available: https://doi.org/10.1145/2509578.2509586

[56] ——, "A lightweight symbolic virtual machine for solver-aided host languages," *SIGPLAN Not.*, vol. 49, no. 6, p. 530–541, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594340

[57] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Solving constrained horn clauses using syntax and data," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. S. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–9. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8603011

[58] A. Blass and Y. Gurevich, *Existential fixed-point logic*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 20–36. [Online]. Available: https://doi.org/10.1007/3-540-18170-9_151

[59] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, *Horn Clause Solvers for Program Verification*. Springer LNCS, 09 2015, vol. Fields of Logic and Computation II (pp.24-51), pp. 24–51.

[60] A. Gurfinkel, "Program verification with constrained horn clauses (invited paper)," in *Computer Aided Verification*, S. Shoham and Y. Vizel, Eds. Cham: Springer International Publishing, 2022, pp. 19–29.

[61] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 183–198. [Online]. Available: https://doi.org/10.1145/1966445.1966463

[62] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 43–48. [Online]. Available: https://doi.org/10.1145/1251535.1251543

[63] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, p. 16–es, may 2007. [Online]. Available: https://doi.org/10.1145/1232420.1232423

[64] J. H. Siddiqui and S. Khurshid, "Parsym: Parallel symbolic execution," in *2010 2nd International Conference on Software Technology and Engineering*, vol. 1, 2010, pp. V1–405–V1–409.

[65] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, ser. APLAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 97–118. [Online]. Available: https://doi.org/10.1007/11575467_8

[66] T. Prabhu, S. Ramalingam, M. Might, and M. Hall, "Eigencfa: Accelerating flow analysis with gpus," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 511–522. [Online]. Available: https://doi.org/10.1145/1926385.1926445

[67] R. Kramer, R. Gupta, and M. Soffa, "The combining dag: a technique for parallel data flow analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 805–813, 1994.

[68] Y.-f. Lee, T. J. Marlowe, and B. G. Ryder, "Performing data flow analysis in parallel," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '90. Washington, DC, USA: IEEE Computer Society Press, 1990, p. 942–951.

[69] M. Méndez-Lojo, A. Mathew, and K. Pingali, "Parallel inclusion-based points-to analysis," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 428–443. [Online]. Available: https://doi.org/10.1145/1869459.1869495

[70] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 196–206.

[71] S. Kumar and T. Gilray, "Load-balancing parallel relational algebra," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 288–308.

[72] T. Gilray and S. Kumar, "Distributed relational algebra at scale," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 12–22.

[73] T. Gilray, S. Kumar, and K. Micinski, "Compiling data-parallel datalog," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 23–35. [Online]. Available: https://doi.org/10.1145/3446804.3446855