Data-Parallel Structured Deduction

THOMAS GILRAY, University of Alabama at Birmingham, USA
ARASH SAHEBOLAMRI, Syracuse University, USA
SIDHARTH KUMAR, University of Alabama at Birmingham, USA
KRISTOPHER MICINSKI, Syracuse University, USA

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 enables 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 each subfact is materialized as a top-class fact. 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 (such as STLC and LF). We performed experiments on EC2, Azure, and ALCF's Theta at up to 1000 threads, showing orders-of-magnitude scalability improvements versus competing state-of-art systems.

CCS Concepts: • **Software and its engineering → Compilers**; **Parallel programming languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: declarative programming, semantics engineering, data-parallel deduction

1 STRUCTURED DECLARATIVE REASONING

Effective programming languages permit their user to write high-performance code in a manner that is as close to their 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 [87] 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 [3, 44] 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) [74] 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 [2, 80].

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 [8] 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 several program analyses and type systems. We include an end-to-end presentation of the systematic development of program analyses from corresponding abstract-machine interpreters—the abstracting abstract machines (AAM) methodology—showing how each intermediate step in the AAM process my 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.

2 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 [11, 12], ported to Soufflé [3] 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 underlying HornSAT 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 like `(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 Slog 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.
- In Slog, 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/values), 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 3 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 3.1 for extensions and idioms). Using these straightforward syntactic extensions enables a wide range of deduction and reasoning systems (see Section 4 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 improvement and parallelism over current state-of-art systems in the implementations of analyses we generate (see Section 6 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.

3 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 4) 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.

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 $Body(R)$ and $Head(R)$ 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., $\langle Subcl \rangle_{DL} ::= \langle Var \rangle \mid \langle Lit \rangle$). This subset (and its semantics) corresponds to Datalog.

$$\langle Prog \rangle ::= \langle Rule \rangle^*$$
$$\langle Rule \rangle ::= \langle Clause \rangle \leftarrow \langle Clause \rangle^*$$
$$\langle Clause \rangle ::= (\text{tag } \langle Subcl \rangle^*)$$
$$\langle Subcl \rangle ::= (\text{tag } \langle Subcl \rangle^*) \mid \langle Var \rangle \mid \langle Lit \rangle$$
$$\langle Lit \rangle ::= \langle Number \rangle \mid \langle String \rangle \mid ...$$

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

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 $IC_P : DB \rightarrow DB$. 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 ($db \in DB = \mathcal{P}(Fact)$). A fact is a clause without variables:

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

In Datalog, *Val*s are restricted to a finite set of atoms ($Val_{DL} ::= Lit$). To define IC_P, we first define the immediate consequence of a rule $IC_R : DB \rightarrow DB$, which supplements the provided database with all the facts that can be derived directly from the rule given the available facts in the database:

$$IC_R(db) \triangleq db \cup \bigcup \Big\{ unroll(Head(R)\big[\overrightarrow{v_i \backslash x_i}\big]) \mid \{\overrightarrow{x_i \rightarrow v_i}\} \subseteq (Var \times Val) \ \wedge Body(R)\big[\overrightarrow{v_i \backslash x_i}\big] \subseteq db \Big\}$$

The *unroll* function has the following definition:

$$unroll\big((tag\ item_1\ ...\ item_n)\big) \triangleq \{(tag\ item_1\ ...\ item_n)\} \cup \bigcup_{i \in 1...n} unroll(item_i)$$
$$unroll(v)_{v \in Lit} \triangleq \{\}$$

The purpose of the unroll function is to ensure that all nested facts are included in the database as well, 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 fact in own right, and not merely a carrier of structured data. Later sections (starting in section 3.1) illustrate the importance of subfact closure by demonstrating how we utilize this behavior to construct idioms that make programming in Slog more convenient.

The immediate consequence of a program is the union of the immediate consequence of each of its constituent rules, $IC_P(db) \triangleq db \cup \bigcup_{R \in P} IC_R(db)$. Observe that IC_P is monotonic over the

the lattice of databases whose bottom element is the empty database. Therefore, if IC_P has any fixed points, it also has a least fixed point [77]. 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 3.1. *(Formalized in Isabelle.) The least fixed point of IC_P is subfact-closed.*

It is worth pointing out that the fixed point semantics of Datalog is similar, 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., [16]. 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 $I = \bigcup\{unroll(f) \mid f \in I\}$. For Datalog, the Herbrand Interpretation is defined similarly, with the difference that subfact-closure is not a requirement for Datalog, as Datalog facts do not contain subfacts.

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.

$$I \models R \text{ iff } \forall\{\overrightarrow{x_i \longrightarrow v_i}\} \;.\; Body(R)\left[\overrightarrow{v_i \backslash x_i}\right] \subseteq I \longrightarrow Head(R)\left[\overrightarrow{v_i \backslash x_i}\right] \in I$$

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 Herbrand models of P. 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 $D(P) \triangleq \bigcap_{I \in \mathbf{M}(P)} I$. It can be shown that such an intersection is a Herbrand model itself:

LEMMA 3.2. *The intersection of a set of Herbrand models is also a Herbrand model.*

Unlike Datalog, nontrivial DL_s programs have Herbrand universes that are infinite. Thus, a DL_s program may have only 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 3.3. *Herbrand models of a DL_s program are fixed points of the immediate consequence operator.*

Lemma 3.4. *Fixed points of the immediate consequence operator of a DL_s program that are subfact-closed are Herbrand models of the program.*

$\langle$*toplvl-rule*$\rangle$ ::= $\langle$*rule*$\rangle$ | $\langle$*hclause*$\rangle$

$\langle$*rule*$\rangle$ ::= [$\langle$*hd-item*$\rangle$* <- $\langle$*bd-item*$\rangle$*]
| [$\langle$*bd-item*$\rangle$* -> $\langle$*hd-item*$\rangle$*]

$\langle$*bd-item*$\rangle$ ::= $\langle$*rule*$\rangle$ | $\langle$*bclause*$\rangle$

$\langle$*hd-item*$\rangle$ ::= $\langle$*rule*$\rangle$ | $\langle$*hclause*$\rangle$

$\langle$*bclause*$\rangle$ ::= ($\langle$*tag*$\rangle$ $\langle$*ibclause*$\rangle$*)
| (= $\langle$*var*$\rangle$ ($\langle$*tag*$\rangle$ $\langle$*ibclause*$\rangle$*))

$\langle$*hclause*$\rangle$ ::= ($\langle$*tag*$\rangle$ $\langle$*ihclause*$\rangle$*)
| (= $\langle$*var*$\rangle$ ($\langle$*tag*$\rangle$ $\langle$*ihclause*$\rangle$*))

$\langle$*atom*$\rangle$ ::= $\langle$*var*$\rangle$ | $\langle$*lit*$\rangle$

$\langle$*lit*$\rangle$ ::= $\langle$*string*$\rangle$ | $\langle$*number*$\rangle$

$\langle$*ihclause*$\rangle$ ::= ($\langle$*tag*$\rangle$ $\langle$*ihclause*$\rangle$*)
| ?($\langle$*tag*$\rangle$ $\langle$*ibclause*$\rangle$*)
| {$\langle$*tag*$\rangle$ $\langle$*ibclause*$\rangle$*}
| [$\langle$*hlist-item*$\rangle$*]
| ?[$\langle$*blist-item*$\rangle$*]
| $\langle$*atom*$\rangle$

$\langle$*ibclause*$\rangle$ ::= ($\langle$*tag*$\rangle$ $\langle$*ibclause*$\rangle$*)
| !($\langle$*tag*$\rangle$ $\langle$*ihclause*$\rangle$*)
| {$\langle$*tag*$\rangle$ $\langle$*ibclause*$\rangle$*}
| [$\langle$*blist-item*$\rangle$*]
| ![$\langle$*hlist-item*$\rangle$*]
| $\langle$*atom*$\rangle$

$\langle$*hlist-item*$\rangle$ ::= $\langle$*ihclause*$\rangle$ | $\langle$*ihclause*$\rangle$ ...

$\langle$*blist-item*$\rangle$ ::= $\langle$*ibclause*$\rangle$ | $\langle$*ibclause*$\rangle$ ...

Fig. 2. The syntax of Sloc. $\langle$*var*$\rangle$ is the set of variables, and $\langle$*tag*$\rangle$ is the set of relation names. A few syntactic forms, including disjunction, have been elided.

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 3.5. *The model theoretic semantics and fixed point semantics of DL_s are equivalent.*

Proof sketch: Form the lemma that all Herbrand models are fixed points of IC_P, we conclude that $D(P)$ is a superset of the intersection of all the fixed points. We know that the least fixed point of IC_P (which we'll call LFP_P) is a subset of the intersection of all the fixed points. We therefore have $LFP_P \subseteq D(P)$. From the fact the LFP_P is a Herbrand model, we conclude that $D(P) \subseteq LFP_P$. Putting these facts together, we conclude that $LFP_P = D(P)$.

3.1 Key extensions to the core language

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.

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

which desugars to the rule

```
[(= 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 to show direction.

Two more rules are needed to define a free-variables analysis. 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

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

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 [4]. 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, which they are compiled into internally.

Consider the example in Figure 3, which lets us prove an arithmetic statement like `(plus (plus (nat 1) (nat 2)) (nat 1)) ⇓ 4`. We can construe this rule in a few ways, as written. It could be that both the expression and value should be provided and are proved according to these rules, or it could be treated as a calculator, with the expression provided as input.

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

$$\frac{}{(\texttt{nat}\ n) \Downarrow n}\,[\text{nat}]$$

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

$$\frac{e_0 \Downarrow v_0 \qquad e_1 \Downarrow v_1 \qquad v = v_0 + v_1}{(\texttt{plus}\ e_0\ e_1) \Downarrow v}\,[\text{plus}]$$

Fig. 3. Natural-deduction-style reasoning with ! clauses in Slog.

Subclauses, written with parentheses, are treated as top-level clauses whose id column value is unified at the position of the subclause. 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 3 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 like `+`.

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

Putting this all together and adding Slog's built-in list syntax—currently implemented as linked-lists of Slog facts in the natural way—we can implement rules for appending lists, a naturally direct-recursive task due to a linked list naturally having its first element at its front, so a second list can only be appended to the back of the first list, and the front element onto the front of that.

```
(append ?(do-append [] ls) ls)

[(append !(do-append ls0 ls1) ls')
 -->
 (append ?(do-append [x ls0 ...] ls1)
         [x ls' ...])]

; or ind. case could even be written:
(append ?(do-append [x ls0 ...] ls1)
        [x
         {append !(do-append ls0 ls1)}
         ...])
```

4 APPLICATIONS

In this section, we will examine several related applications of Slog: implementing reduction systems, natural deduction systems, AAM-based program analyses, and natural-deduction-style type systems.

We start with a λ-calculus interpreter. Let's observe how β-reduction can be defined via capture-avoiding substitution. If a **do-subst** fact is emitted where a reference to variable `x` is being substituted with expression `E`, associate it in the **subst** relation with `E`:

```
(subst ?(do-subst (ref x) x E) E)
```

However, if `x` and `y` are distinct variables, the substitution yields expression `(ref x)` unchanged:

```
[(=/= x y) --> (subst ?(do-subst (ref x) y E) (ref x))]
```

Recall that ?-clauses are body clauses, so these rules could also have been written more verbosely:

```
[(= d (do-subst (ref x) x E)) --> (subst d E)]
[(=/= x y) (= d (do-subst (ref x) y E)) --> (subst d (ref x))]
```

At a lambda, where the formal parameter shadows the variable being substituted, its scope ends and substitution stops:

```
(subst ?(do-subst (lam x Ebody) x E) (lam x Ebody))
```

If the variable does not match and is not free in the (argument) expression `E`, substitution may continue under the lambda, triggered by a ! clause:

```
[(=/= x y)  ~(free E x)
 --> (subst ?(do-subst (lam x Ebody) y E)
            (lam x {subst !(do-subst Ebody y E)}))]
```

Three further syntactic extensions are being used in this rule. First off, the negated `(free E x)` clause in the body requires that the compiler stratify computation of `free`, as normal when adding otherwise nonmonotonic rule dependance to Datalog.

Second, the process of rewriting the lambda body is triggered by the establishment of a **`do-subst`** fact via a ! clause. These ! clauses generate facts on-the-fly during rule evaluation, allowing other rules to hook-in by generating a fact to trigger them (using a ! clause) and expecting a response in the surrounding body clauses. These ! clauses are implemented by generating a subrule whose head clause is the intermediate ! clause and whose body contains all body clauses the ! clause depends upon, along with and any ? clauses in the rule (which are always required to trigger any rule). In this case, term-substitution rules respond to the **`do-subst`** request via the **`subst`** relation, as queried here by the {} expression in `{subst !(do-subst Ebody y E)}`.

Third, this {} syntax allows for looking up the final column of a relation by providing all but the final-column value. `(foo x {bar y})` desugars into `(and (foo x z) (bar y z))`, allowing for the looked-up value to be unified with the position of the {} expression in a natural way. Do note that the relation need not actually be functional and could just as easily associate multiple values with any input. {} expressions and ! clauses are especially expressive when used together in this way for direct recursion.

If we were to desugar the {} syntax, ? clause, and ! clause in this rule, we would obtain two rules. The rule below on the left emits a **`do-subst`** fact for the body of the lambda, if it qualifies for rewriting, and a **`ruleXX-midpoint`** fact saving pertinant details of the rule needed in its second half. Below on the right, the second half of the rule requires that the first half of the rule triggered and that the **`subst`** relation has responded with a rewritten lambda body for the **`do-subst`** fact `do'`.

```
[(=/= x y)  ~(free E x)
 (= do (do-subst (lam x Ebody) y E))
 -->
 (= do' (do-subst Ebody y E))
 (ruleXX-midpoint do do' x)]
```

```
[(ruleXX-midpoint do do' x)
 (subst do' Ebody')
 -->
 (subst do (lam x Ebody'))]]
```

Finally, in the case of an application, substitution is performed down both subexpressions. If one ! clause were nested under the other, they would need to be ordered. In this case, the compiler will detect that both !-clause facts can be emitted

```
(subst ?(do-subst (app Ef Ea) x E)
       (app {subst !(do-subst Ef x E)}
            {subst !(do-subst Ea x E)}))
```

in parallel, so this rule will also split into two rules as in the rule above. The first rule will generate both !-clause facts and the second rule will await a response for both.

With a substitution function defined, we can define evaluation. A lambda is already fully reduced. An application reduces its left-hand subexpression to a lambda, substitutes the argument for the formal parameter, and reduces the body.

```
; values
(interp ?(do-interp (lam x body))
        (lam x body))
; application
[(interp !(do-interp fun) (lam x body))
 (subst !(do-subst body x arg) body')
 -->
 (interp ?(do-interp (app fun arg))
         {interp !(do-interp body')})]
```

The compiler will detect in this case that the `do-subst` !-clause fact depends on variable `body`, and the `do-interp` ! clause in the head depends on `body'`, but that the first `do-interp` !-clause fact only depends on the variable fun from the original ?-clause fact kicking off the rule. These three sequential ! clauses split the rule into four parts during compilation, just as a CPS tranformation [4] would explicitly break a traditional functional implementation of this recursive, substitution-based interpreter into one function entry point and three continuation entry points. Unlike traditional CPS translation of functional programs however, ! clauses in Slog will naturally emit multiple facts for parallel processing in a nonblocking manner when variable dependence allows for parallelism.

4.1 Abstract Machines

Next, instead of using terms alone to represent intermediate points in evaluation, we may wish to explicitly represent facets of evaluation such as the environment, the stack, and the heap. Instead of representing environments through substitution, we may want to represent them explicitly in a higher-order way. As shown in Figure 4, with first-class facts and ad hoc polymorphic rules, we can use defunctionalization to implement first-class relations, providing a global `env-map` relation, we can read with an `{env-map env x}` expression (assuming ground variables `env` and `x`), along with a `(ext-env env x val)` facility for deriving an extended environment.

```
; environments (defunctionalized)
(env-map ?(ext-env env x val) x val)
[(=/= x y) --> (env-map ?(ext-env env x _) y {env-map env y})]
```

Fig. 4. Defunctionalized environments; extension via `(ext-env env x v)`, lookup via `{env-map env x}`.

On the left of Figure 5 shows an abstract machine for CBN evaluation, and on the right, an abstract machine for CBV evaluation. At the top, the rules for reference use `{env-map env x}` to access the value from the defunctionalized `env-map` relation. In the CBN version, we cannot count on the stored closure to be a lambda closure, so we continue interpretation, using another {} expression to drop-in the transitive reduction of the stored argument closure. Lambda closures are the base case which `interp` as themselves. Finally, application closures trigger a closure to evaluate `Ef` via a ! clause, `!(clo Ef env)`, and the lambda closure that finally results has its body evaluated under its environment, extended with parameter mapped to argument. In the CBN interpreter, `(ext-env env' x (clo Ea env))` puts the argument expression `Ea` in the environment, closed with the current environment. In the CBV interpreter, `(ext-env env' x Eav)` puts the argument value `Eav` in the environment (after first evaluating it). In both these interpreters, the `app`-handling rules use ! clauses to implicitly create handling rules and a chain of continuation facts so `interp` maybe be utilized in a direct-recursive manner. The ! syntax introduces a CPS-style transformation that provides a stack in the interpretation of Slog rules for these CE interpreters to map their stack onto.

```
; ref
(interp ?(clo (ref x) env)
        {interp {env-map env x}})
; lam
(interp ?(clo (lam x Eb) env)
        (clo (lam x Eb) env))
; app
[(interp !(clo Ef env)
         (clo (lam x Eb) env'))
 (= env'' (ext-env env' x (clo Ea env)))
 (interp !(clo Eb env'') v)
 -->
 (interp ?(clo (app Ef Ea) env) v)]
```

```
; ref
(interp ?(clo (ref x) env)
        {env-map env x})
; lam
(interp ?(clo (lam x Eb) env)
        (clo (lam x Eb) env))
; app
[(interp !(clo Ef env)
         (clo (lam x Eb) env'))
 (interp !(clo Ea env) Eav)
 (interp !(clo Eb (ext-env env' x Eav)) v)
 -->
 (interp ?(clo (app Ef Ea) env) v)]
```

Fig. 5. Two CE (closure-creating) interpreters in Slog; for CBN eval. (left) and CBV eval. (right).

```
; eval ref
(interp ?(cek (clo (ref x) env) k)
        {interp !(cek env-map env x k)})

; eval lam (apply)
(interp ?(cek (clo (lam x Eb) env)
              [aclo k ...])
        {interp !(cek (clo Eb
                           (ext-env env x aclo))
                      k)})

; eval app
(interp ?(cek (clo (app Ef Ea) env) k)
        {interp !(cek (clo Ef env)
                      [(clo Ea env) k ...])})
; return / halt
(interp ?(cek (clo (lam x Eb) env) [])
        (clo (lam x Eb) env))
```

```
; eval ref
(interp ?(cek (clo (ref x) env) k)
        {interp !(cek env-map env x k)})
; eval lam (ret to ar-k)
(interp ?(cek (clo (lam x Eb) env)
              (ar-k aclo k))
        {interp !(cek aclo
                      (fn-k (clo (lam x Eb) env)
                            k))})
; eval lam (ret to fn-k)
(interp ?(cek (= aclo (clo (lam _ _) _))
              (fn-k (clo (lam x Eb) env) k))
        {interp !(cek (clo Eb
                           (ext-env env x aclo))
                      k)})
; eval app
(interp ?(cek (clo (app Ef Ea) env) k)
        {interp !(cek (clo Ef env)
                      (ar-k (clo Ea env) k))})
; return to (halt-k)
(interp ?(cek (clo (lam x Eb) env) (halt-k))
        (clo (lam x Eb) env))
```

Fig. 6. Two CEK (stack-passing) interpreters in Slog; for CBN eval. (left) and CBV eval. (right).

We can also implement the stack ourselves within our interpreter, thereby eliminating its need for our interpreter itself, by applying a stack-passing transformation. On the left of Figure 6 shows Krivine's machine [47], a tail-recursive abstract machine for CBN evaluation, and on the right, a tail-recursive abstract machine for CBV evaluation. Each of these machines incrementally constructs and passes a stack. In the CBN stack-passing interpreter, each application reached pushes a closure for the argument expression onto the stack. When a lambda is reached, this continuation is handled by popping its latest closure, the argument value. In the CBV stack-passing interpreter, each application reached pushes an **`ar-k`** continuation frame on the stack to save the argument value and environment. Whan a lambda is reached, this continuation is handled by swapping it for a **`fn-k`** continuation that saves the function value while the argument expression is evaluated (before application). Finally, when a lambda is reached, the **`fn-k`** continuation is handled by applying the saved closure. Now that the stack is entirely maintained by the interpter itself, you may note that all recursive uses of `{interp (cek ...)}` are in tail position for the result column of relation **`interp`**.

4.2 Abstracting Abstract Machines

The *abstracting abstract machines* (AAM) methodology [60, 85] proscribes a particular systematic application of abstract interpretation [17–19] on abstract-machine operational semantics like those we've just built in Slog. 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. In particular, there are two main sources of unboundedness in the CEK machines: environments and continuations. Environments contain closures which themselves contain environments; continuations are a stack of closures formed inductively in the CBV CEK machine and formed using Slog's list syntax in the CBN CEK machine to more closely follow the usual presentation of Krivine's machine [47]. 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 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.

```
; eval ref -> ret
[(eval (ref x) env sto k c)
 -->
 (ret {sto-map sto {env-map env x}} sto k c)]
; eval lam -> ret
[(eval (lam x Eb) env sto k c)
 -->
 (ret (clo (lam x Eb) env) sto k c)]
; eval app -> eval
[(eval (app Ef Ea) env sto k c)
 -->
 (eval Ef env sto (ar-k Ea env k) c)]
; ret to kaddr -> ret
[(ret vf sto (kaddr c') c)
 -->
 (ret vf sto {sto-map sto (kaddr c')} c)]
; ret to ar-k -> eval
[(ret vf sto (ar-k Ea env k) c)
 -->
 (eval Ea env sto (fn-k vf k) c)]
; ret to fn-k -> apply
[(ret va sto (fn-k vf k) c)
 -->
 (apply vf va sto k c)]
; apply -> eval
[(apply (clo (lam x Eb) env) va sto k c)
 -->
 (eval Eb
       (ext-env env x (addr c))
       (ext-sto (ext-sto sto (kaddr c) k)
                (addr c) va)
       k
       {+ 1 c})]
```

Fig. 7. A CESKT (control, environment, store, kontinuation, timestamp) interpreter in Slog.

Figure 7 shows the CBV CEK machine of Figure 6 modified in a few key ways, yielding a CESKT machine with control expression, environment, store, continuation, and timestamp/contour components:

- *abstract-machine states have been factored* into **`eval`**, **`apply`**, and **`ret`** configurations; an **`eval`** state has a control expression, environment (mapping variables to addresses), store (mapping addresses to closures and continuations), current continuation, and timestamp (tracking the size of the store, and thus the next address); an `apply` state has a closure being applied, argument value, store, continuation, and timestamp; and a `ret` state has a value being returned, a store, a continuation, and a timestamp;
- *state transitions have been written as small-step rules* that always terminate; previously, our CEK machines were written to take a big-step from **`cek`**-state to the final, denoted value as logged in the `(interp e v)` relation, but in tail-recursive fashion, using ! clauses); Figure 7 has no explicit small-step relation, but simply says, for example, that the existence of a **`ret`** state permits us to deduce to existence of an **`apply`** state; if we were to want an explicit **`step`** relation, we could again give this rule a presentation with an implied body via a ? clause; for example:

```
(step ?(ret va sto (fn-k vf k) c)
      (apply vf va sto k c))
```

- *states have been subjected to a store-passing transformation* which has added a store `sto` and timestamp (stored-value count) `c` to each state; environments now bind variables to addresses and the current store binds those addresses to values; we perform a variable lookup with `{sto-map sto {env-map env x}}`; at an `apply` state, we use the store count `c` to generate a fresh address `(addr c)` for the parameter `x`; we also store-allocate the current continuation at a continuation address `(kaddr c)`, in preparation for modeling the stack finitely as well; when returning to a `(kaddr c)`, the continuation is simply fetched from the store as in the fourth rule down (ret to kaddr).

From here it sufficies to pick a finite set from which to draw addresses. To instantiate a monovariant control-flow analysis from this CESKT 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 `sto-map` relation encoding stores conflates multiple values at a single address for the same variable. Conflation in the store would lead naturally to nondeterminism in any `step` relation. When looking up a variable, two distinct `ret` states could result, leading to two distinct `apply` states after some futher steps.

A (potentially) more precise, though (potentially) more costly analysis would be to specialize all control-flow points and store points by a finite history of recent or enclosing calls. Such a *k-call-sensitive* analysis can be instantiated using a specific instrumentation and allocation policy, as can many others [32]. 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 programming analyses, greater precision sometimes goes hand-in-hand with lower cost in practice because values that are simpler and fewer are simpler to represent [89]. 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 [35] so as to adapt to the value polyvariance chosen.

The per-state store by itself is a source of exponential blowup for any polyvariance control-flow analysis [59]. Instead, it is standard to use a global store and compute the least-upper-bound of all per-state stores. In Slog this is as simple as using a single global `(store addr val)` relation instead of a defunctionalized `(sto-map sto addr val)` relation that represents all per-state stores in one. The left side of Figure 8 shows a version of the CESKT machine with a global store and a tunable instrumentation that can be varied by changing the `tick` function rule. Currently, `tick` instantiates this to a 3-k-CFA: at each function application, the current call site (now saved in the `ar-k` and `fn-k` continuation frames to provide to the apply state) is saved in front of the current call history and the fourth-oldest call is dropped.

This is the classic k-CFA, except perhaps that the original k-CFA, formulated for CPS as it was, also tracked returns positively instead of reverting the timestamp as functions return like we do here. The original k-CFA used true higher-order environments, unlike equivalent analyses written for object oriented languages which implicitly had flat environments (objects) [61]. The corresponding C FA for functional languages is called m-CFA and is shown on the right side of Figure 8. m-CFA has only the latest call history as a flat context. Instead of having 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

```
;; 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])
```

```
;; Eval states
[(eval (ref x) k c)
 -->
 (ret {store (addr x c)} k)]
[(eval (lam x body) k c)
 -->
 (ret (clo (lam x body) c) k)]
[(eval (app ef ea) k c)
 -->
 (eval ef (ar-k ea (app ef ea) c k) c)]


;; Ret states
[(ret vf (ar-k ea call c k))
 -->
 (eval ea (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 c))
 (store (kaddr e c) k)
 -->
 (ret v k)]
;; Apply states
[(apply call (clo (lam x Eb) _) va k c)
 -->
 (eval Eb (kaddr Eb c') c')
 (store (kaddr Eb c') k)
 (store (addr x c') va)
 (= c' {tick !(do-tick call c)})]
; Propagate free vars
[(free y (lam x body))
 (apply call (clo (lam x body) clam) _ _ c)
 -->
 (store (addr y {tick !(do-tick call c)})
        {store (addr y clam)})]
```

Fig. 8. An AAM for global-store k-CFA (left) and m-CFA (right) in Slog. These are evaluated in Section 6.

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)`.

4.3 Type Systems

Along with operational semantics and analysis, Slog's focus on exposing subfacts—and thus computation structure—as a first-class object naturally extends to the implementation of structural type systems [64, 65]. In this section we will develop several type systems in Slog, starting with the simply-typed λ-calculus and building up to first-order dependent types. We first detail a transliteration of a textbook development of STLC and remark upon our use of Slog's constructs to systematically map each rule in the typing judgement $\Gamma \vdash e : \tau$ to a corresponding rule in Slog. We next discuss explicit materialization of proof objects and intuitionistic propositional logic. We conclude by implementing first-order dependent types (the $\forall$, $\Longrightarrow$ fragment of intuitionistic predicate logic) in Slog by developing a 150-line rendition of λLF.

In this section, we focus on the transliteration of *algorithmic* (i.e., syntax-directed) type checking algorithms in Slog. While type checking (or more broadly, type synthesis) is not syntax-directed in general, we focus here on type systems and logics which have simple and well-understood

$$
\begin{array}{llll}
\textit{STLC Terms} & e & ::= & (\lambda\,(x : \tau)\,e) \\
 & & \mid & (e_0\ e_1) \\
 & & \mid & x \\
\lambda\textit{LF Contexts} & \Gamma & ::= & \varnothing \\
 & & \mid & \Gamma, x : T \\
 & & \mid & \Gamma, x :: K \\
\lambda\textit{LF Kinds} & K & ::= & \Pi x : T.K \mid * \\
\textit{STLC Types} & \tau \in T & ::= & \tau \to \tau \\
 & & \mid & \ldots \\
\lambda\textit{LF Types} & \tau \in T & ::= & X \\
 & & \mid & \Pi x : T.\ T \\
 & & \mid & (T\ e)
\end{array}
$$

Fig. 9. Syntax for the formal systems we present.

$$\text{T-Var}\quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

```
[-->;-------- T-Var
 (: ?(ck Γ (ref x)) {env-ext Γ x})]
```

$$\text{T-Abs}\quad \frac{\Gamma, x : T_1 \vdash e : T_2}{(\lambda\,(x : T_1)\,e) : T_1 \to T_2}$$

```
[(: !(ck (env-ext Γ x T1) e) T2)
 -->;-------- T-Abs
 (: ?(ck Γ (λ x T1 e)) (-> T1 T2))]
```

$$\text{T-App}\quad \frac{\Gamma \vdash e_0 : T_0 \to T_1 \qquad e_1 : T_0}{\Gamma \vdash (e_0\ e_1) : T_1}$$

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

Fig. 10. Simply-Typed Lambda Calculus (TAPL Fig. 9.1) and Slog Equivalent

decidability and completeness characteristics. For example, the type systems of STLC and *λLF* rule out problematic unbounded terms (such as Ω) which would otherwise lead to intractable materialization. We anticipate Slog's declarative style will also be a good fit for other more general formalizations of type systems, including substructural typing, bidirectional typing [25, 66], refinement typing [22, 24, 30], liquid types [69, 86], and so on; however, we leave these to future work.

Simply-typed λ-calculus. We begin with the simply-typed λ-calculus [7, 64]. The syntax of STLC terms and types is shown at the top of Figure 9. Our syntax roughly follows Chapter 9 of Benjamin Pierce's *Types and Programming Languages* [64]. STLC extends the untyped λ-calculus: λ-abstractions are annotated with types, and variable typing defers to a typing environment which assigns types to type variables and is subsequently extended at callsites. STLC defines a notion of "simple" types including (always) arrow types between simple types and (sometimes) base types (e.g., nat), depending on the presentation.

The typing judgement is presented in Figure 10 as a set of natural-deduction style rules (left), alongside the corresponding Slog rules (right). Each typing rule is predicated upon a message `?(ck Γ e)`, which triggers type synthesis for `e` under the typing environment Γ. The inference rules (left) are schematic in the term being checked, but are intended to be instantiated in a demand-driven way by a proof author (however automated)—`ck` formalizes the demand-driven nature of the rules, though we must build some infrastructure to demand the type of the top-level expression:

```
[(success Γ e τ) <-- (typecheck Γ e τ) (: !(ck Γ e) τ)]
```

To execute the type checking algorithm, the user supplies as input a fact `(typecheck Γ e τ)` with the desired environment, term, and suspected type. While Slog will not materialize any outputs in `success` for ill-typed terms, the relatively-straightforward use of stratified negation (which we

$$\text{TA-Abs} \quad \frac{\Gamma \vdash S :: * \qquad \Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda (x : S) t) : \Pi x : S . T}$$

$$\text{TA-App} \quad \frac{\Gamma \vdash t_1 : \Pi x{:}S_1 . T \qquad \Gamma \vdash t_2 : S_2 \qquad \Gamma \vdash S_1 \equiv S_2}{\Gamma \vdash (t_1\ t_2) : [x \mapsto t_2]\, T}$$

$$\text{KA-App} \quad \frac{\Gamma \vdash S :: \Pi x{:}T_1 . K \qquad \Gamma \vdash t : T_2 \qquad \Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash (S\ t) : [x \mapsto t]\, K}$$

```
[(->wh !(do->wh t1) t1')
 -->;------
 (->wh ?(do->wh (app t1 t2)) (app t1' t2))]
```

```
[(:: !(ck-k Γ S) (star))
 (: !(ck-t (ext-env Γ x S) t) T)
 -->;------
 (: ?(ck-t Γ (λ x S t)) (Π x S T))]

[(: !(ck-t Γ t1) (Π x S1 T))
 (: !(ck-t Γ t2) S2)
 --
 (yes !(=== Γ S1 S2))
 -->;------
 (: ?(ck-t Γ (app t1 t2))
   {subst !(do-subst T x t2)})]

[(:: !(ch-k Γ S) (Π x T1 K))
 (: !(ck-t Γ t) T2)
 --
 (yes !(=== Γ T1 T2))
 -->;------
 (: ?(ch-t Γ (type-app S t))
   {subst !(do-subst K x t)})]

(->wh ?(do->wh (app (λ x T1 t1) t2))
  {subst !(do-subst t1 x t2)})
```

Fig. 11. *λLF*: Selected rules

have implemented in Slog) allows using our typechecking algorithm as a decision procedure, due to the completeness of STLC.

Proof Theory and Intuitionistic Propositional Logic. The well-known Curry-Howard Isomorphism relates terms in pure functional languages to proofs in constructive logics [20, 21, 42]. For example, the existence of an isomorphism between terms in STLC and proofs in intuitionistic propositional logic (IPL) imply that our Slog rules (above) may also be read as proofs of tautologies in IPL. However, one difference between our Slog implementation and the on-paper semantics for STLC is the exact materialization of proof tree. For example, consider the T-App rule; when each of the premises are deduced (demanded via `ck`), the type of the application as a whole is deduced. Our implementation does not explicitly materialize a proof tree for each antecedent in T-App; instead, we rely upon the semantics of Horn clauses and the behavior of other rules' responses to `ck`. A transformation to explicitly bind each subfact allows materializing an explicit proof.

First-Order Dependent Types: λLF. The Edinburgh Logical Framework (*λLF*) is a dependently-typed λ-calculus [40]. It is a first-order dependent type system, in the sense that it stratifies objects into kinds, types (families), and terms (objects)—kinds may quantify over types, but not over other kinds. The syntax of *λLF* is detailed at the bottom of Figure 9—it extends STLC with kinds, which are either $*$ or (type families) $\Pi x : T.K$, where T is a simple type (of kind $*$). *λLF* generalizes the arrow type to the dependent product: $\Pi x : T.T$. System *λLF* enjoys several decidability properties which make it particularly amenable to implementation in Slog. The first is strong normalization, which implies that reduction sequences for well-typed terms in our implementation will be finite. The second is *λLF*'s focus on canonical forms and hereditary substitution [41]. In *λLF*, terms are canonicalized to weak-head normal form (WHNF); this choice enables inductive reasoning on these canonical forms, and this methodology forms the basis for Twelf [63].

The *judgments-as-types* principle interprets type checking for *λLF* as proving theorems in intuitionstic predicate logic; Using this principle, we may define traditional constructive connectives (such as $\wedge$, $\vee$, and $\exists$) via type families and their associated rules. For example, including in Γ a binding $\wedge \mapsto \Pi P : \mathit{prop}.\, \Pi Q : \mathit{prop}.\, *$ allows using the constructor $\wedge$, though $\wedge$ must be instantiated with a suitable P and Q, which must necessarily be of some sort (e.g., *prop*) also bound in Γ.

We have performed a transliteration of *λLF* as formalized in Chapter 2 of *Advanced Topics in Types and Programming Languages (ATAPL)* [65]. Our transliteration (from pages 57–58) consists of roughly 150 lines of Slog code. Figure 11 details several of the key rules. TA-Abs introduces a Π type, generalizing the T-Abs rule in Figure 10. The TA-App applies a term t_1, of a dependent product type $\Pi x : S_1 \,.\, T$, whenever the input t_2 shares an equivalent type, S_2. The notion of equality here is worth mentioning: ≡ demands reduction of its arguments to WHNF—*λLF* is constructed to identify terms under WHNF, thus ensuring ≡ will terminate as long as the term is typeable. Reduction to WHNF is readily implemented in Slog; two exemplary rules detailed at the bottom of Figure 11 outline the key invariant in WHNF: reduce down the leftmost spine, eliminating β-redexes via application. Equality checks are demanded by the TA-App and KA-App rules, and force normalization of their arguments to WHNF before comparison of canonical forms, generating a witness in the `yes` table before triggering the head of the rule.

5 IMPLEMENTATION

We have implemented Slog in a combination of Racket (the compiler, roughly 10,600 lines), C++ (the runtime system and parallel RA backend; roughly 8,500 lines), Python (a REPL and daemon; roughly 2,500 lines) and Slog (60 lines for list-splicing support). In this section, we describe relevant particulars.

5.1 Compiler

Our Racket-based compiler translates Slog source code to C++ code that links against our parallel RA backend. Our compiler is structured in the nanopass style, composed of multiple passes which consume and produce various forms of increasingly-low-level intermediate representations (IRs) [45]. After parsing, *organize-pass* performs various simplifications, such as canonicalizing the direction of `-->` and splitting disjunctive body clauses and conjunctive head clauses into multiple rules. This pass also eliminates various syntactic niceties of the language, including ! and ? clauses, nested rules, list syntax, and splicing variables. When a program includes splicing variables, the compiler concatenates a small splicing support library. Finally, this pass performs static unification, syntactically identifying clauses and variables that are statically constrained to be equal.

Slog's distribution paradigm is built upon binary joins. Thus, after organization, *partitioning-pass* breaks down bodies with multiple clauses into sequences of binary joins. Partitioning represents an algorithmic challenge, as there may be many ways to partition a set of clauses into sequences of binary joins. For example, a rule such as [H <-- B_1 B_2 ... B_n] may be converted into an $(n-1)$-length sequence of binary joins (first joining B_1 and B_2 to form an intermediate relation which is subsequently joined with B_3) or a tree of binary joins (joining each B_i and B_{i+1} into intermediate relations which are then joined). Unfortunately, optimal partitioning is undecidable in general, and our compiler relies upon a set of heuristics and practical optimizations we've found to work well in practice, along with enabling the user to manually suggest a partitioning by using a `--` syntax operator. Our early implementations preferred to form trees of joins—based on the intuition that this would extract more parallel work—however, we found this often resulted in materializing large numbers of unnecessary facts (essentially forming large Cartesian products to later be filtered in subsequent joins). We have come to see effective partitioning as a fundamental part of writing high-performance code in Slog, similar to Soufflé's Sideways Information Passing Strategy [78].

After partitioning, *split-selections-pass* performs index selection by inspecting each rule in the program and calculating a set of indices. There are two important differences between Slog and a typical shared-memory Datalog implementation. First, because of our distribution methodology, it is impossible to organize indices in a trie-shaped representation that allows overlap-based index compression [76]. Second, fact interning is slightly tricky in the presence of multiple indices: we

must be careful to avoid a fact being assigned two distinct intern keys during the same iteration in separate indices. Our solution is to designate a special *canonical index* which is used for intern key origination, along with a set of special administrive rules which replicate, for each relation, the intern key to every non-canonical index.

The last two passes are strongly-connected component (SCC) construction and incrementalization. Datalog programs are typically stratified into a plan of SCCs for evaluation. This aids efficiency (a single-node implementation may ignore considering rules unncessary to the current SCC, our distributed implementation evaluates SCCs using task-level parallelism) and is also semantically relevant in the presence of stratified negation (to ensure all negated relations are computed strictly-before negation runs). Finally, incrementalization (i.e., semi-naïve evaluation) transforms the program to use a worklist-based evaluation strategy, every relation appearing in a rule body is split into two versions—**delta** and **total**. Rules are rewritten to add facts to **delta**, while bodies are triggered by new entries in **delta**; our backend merges **delta** into **total** at the end of each iteration.

5.2 Backend

Our parallel relational-algebra backend supports fixed-point iterations and is designed for large-scale multi-node HPC clusters. Based on the bulk-synchronous-processing protocol and built using the `MPI-everywhere` model [29, 91], the parallel RA framework addresses the problem of partitioning and balancing workload across processes by using a two-layered distributed hash-table [50]. 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 12 shows a schematic diagram of all the phases (including the 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 [84]. 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.

A major challenge with parallel workload partitioning is to ensure that every process gets to work on similar sized workloads that gives a load balanced system. A major challenge to enforce 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 [50, 51] by developing strategies that mitigate load-imbalance in a dynamic manner. The approach [50] 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. Our scheme permits buckets that have more tuples to be split across multiple processes, but requires some additional communication among subbuckets for any particular bucket. We have developed and evaluated a static refinement strategy, used before fixed-point iteration, to decide how many subbuckets to allocate per-bucket. To distribute subbuckets to managing processes, we use a round-robin mapping scheme which we found to be significantly more effective than hashing.

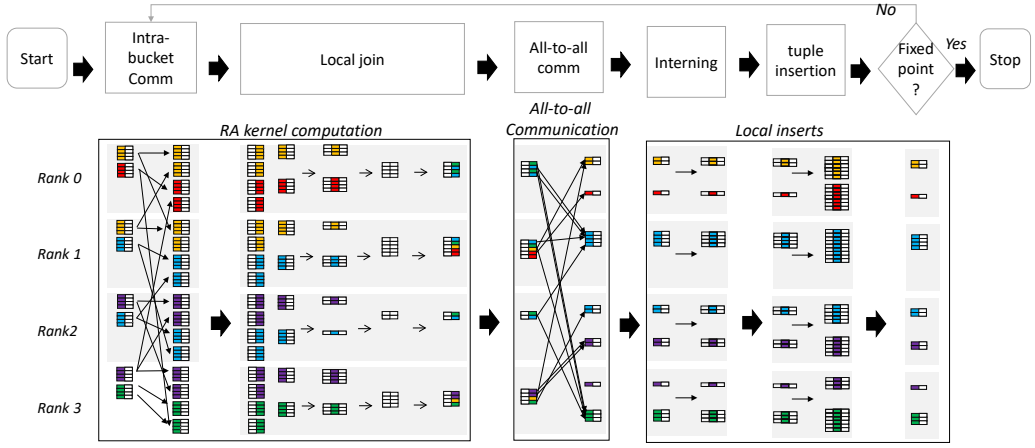

Fig. 12. An illustration of the main phases of our parallel RA backend.

A join operation can only be performed for two *co-located relations*: two relations each keyed on their respective join columns that share a bucket decomposition (but not necessarily a subbucket decomposition for each bucket). This ensures that the join operation may be performed separately on each bucket as all matching tuples will share a logical bucket; it does not, however, ensure that all two matching tuples will share the same subbucket as tuples are assigned to subbuckets (within a bucket) based on the values of non-join columns, separately for each relation. The first step in a join operation is therefore an *intra-bucket communication* (Figure **??**) 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 (Figure **??**).

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 their position in an 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. As tuples generated from the local compute phase may each belong to an arbitrary bucket/sub-bucket in the output relation, an all-to-all communication phase is required to shuffle the output tuples to their managing processes. Given variability in the number of tuples generated across processes, and in their destination processes (due to inherent imbalance), the communication phase in our framework is *non-uniform* in nature. The output tuples may each belong to an arbitrary bucket in the output relation, an MPI *all-to-all* communication phase shuffles the output of all joins to their managing processes (preparing them for any subsequent iteration).

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 [49, 72, 79]—largely because of the *quadratic* nature of its workload. We address this scaling issue by adopting recent advancements [26] that optimizes non-uniform all-to-all communication by extending the log-time Bruck algorithm [13, 79, 83] for non-uniform all-to-all workloads. Traditional algorithms to implement non-uniform all-to-all communication takes linear iterations as every process must send and receive from every

other process. Bruck algorithm however, sends more amount of data in logarithmic steps, and therefore significantly improves overall performance of all-to-all data exchange. Using the Bruck implementation of non-uniform all-to-all algorithm was instrumental in the overall scalibility of the backend.

Local inserts. After all-to-all data exchange, every process receives new set of 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 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. After interning, facts are added to their appropriate versions of the relations (delta or total), and a check is performed to see if fixed-point is reached. This check is performed by a global operation that checks the size of all relations across all processes, and if all sizes remains unchanged across two iterations, then this indicates that fixed-point has been attained and the program terminates, otherwise a new iteration is initiated.

6 EVALUATION

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

- **PB1** (Section 6.1) How does Slog compare against other systems designed for performance and parallelism on traditional Datalog workloads (without ADTs): Soufflé and RadLog?
- **PB2** (Section 6.2) 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 6.3) How well can Slog scale on multiple nodes 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 4.2 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 800 cores on Theta.

6.1 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 [36]. 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.

Table 1. 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	–	–	–	–

We performed comparisons on an Standard_M128s instance rented from Microsoft Azure [58]. 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 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 frist column of Table 1: 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 [23]. 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 1 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.

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

6.2 AAMs and CFAs

Next, we sought to benchmark the analyses described in Section **??** 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 PhD thesis. 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 2. Each of the 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× in Slog, compared to a slowdown of 7× in Soufflé. A similar trend happens in many other experiments, e.g., 15 minutes to over three hours for Soufflé (13× slowdown) vs. 2 to 4 seconds (2× 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,$KAddr(e, env)),
                kont_map($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 Soufllé'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).

6.3 Multi-node Scaling Experiments

In the past couple years, both Microsoft Azure and Amazon Web Services have launched MPI-capabable HPC nodes for their cloud services and significantly upgraded their network interconnects. 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. What is possible today only on leadership-class supercomputers will become possible in cloud-based clusters within just a few years. 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 [62] 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 Dragonfly network topology and a Lustre filesystem.

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 13. All three experiments took 3693 iterations to complete, respectively generating 90,668,904, 338,619,102 and 1,286,335,724 facts. For all three set of experiments we observe improvement in performance with increase in process counts, until maximum efficiency is attained,

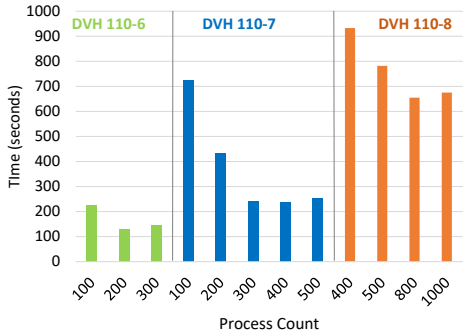

Fig. 13. Strong-scaling experiments on Theta.

after which performance degrades with increasing process counts, due to communication overhead and workload starvation. We find the scaling trends of our system to be encouraging. In general, for a given workload (problem size), there will be a range of processes that exhibits good scaling characteristics. `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 when process count is increased from 400 to 800).

7 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 [74], Distributed SociaLite [73], Myria [39], and Radlog [37] 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 [2, 48, 68, 80]. 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× 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×) compared to BigDatalog by using Apache Flink, a data-flow processing language [43].

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 [5, 6, 8, 54, 56]. Flix augments Datalog with lattices [54, 56], 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 [70]. Similarly, Datafun is a pure functional language which computes

fixed points of monotone maps on semilattices [5, 6]. 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 [15, 31]. 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 [8, 55, 81, 82]. These constraints typically allow interfacing with an SMT solver, potentially in a loop with subsequent analysis [8]. Formulog includes ADTs and first-order functions over ADTs, allowing Turing-equivalent to build formulas of arbitrary size to be checked by Z3 [8]; 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) [81, 82]. 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 [9, 10, 27, 38]. 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 [1, 3, 11, 14, 75, 90] or implementation using special-purpose datastructures [46, 53, 57, 67, 88]. There are a variety of fundamental approaches to scalability; for example, summarization-based analyses (such as Saturn [1, 90]) 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—and such applications have had surprising success (e.g., ddisasm [28]), but the goals of Slog are most closely related to current efforts on scaling rich, whole-program context-sensitive analyses using deductive inference [3, 44, 71].

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 [33, 34, 52]. Our backend is based upon BPRA but adds an interning phase to facilitate ubiqutuous fact interning, which necessitated significant extensions to BPRA's source code. Second, BPRA focuses mainly on the low-level implementation of relational algebra kernels rather than a unified programming language, compiler, and runtime system as we have presented. In our experimentation, we found that effectively scaling parallel relational algebra required significant algorithmic intuition that was extremely challenging to achieve using the parallel relational algebra API alone. Slog's features and semantics has enabled us to rapidly experiment with a wide variety of ideas in programming languages which, as our experiments in Section 6.2 show, may be hard to express in Soufflé due to unanticipated explosion in algorithmic complexity due to subfact indexing.

REFERENCES

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, page 43–48, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. Bridging the gap between hpc and big data frameworks. *Proc. VLDB Endow.*, 10(8):901–912, apr 2017.

[3] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis 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*, pages 25–30. ACM, 2017.

[4] Andrew W Appel. *Compiling with continuations*. Cambridge university press, 2007.

[5] Michael Arntzenius and Neel Krishnaswami. Seminaïve evaluation for a higher-order functional language. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[6] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 214–227, New York, NY, USA, 2016. Association for Computing Machinery.

[7] H. Barendregt, W. Dekkers, R. Statman, and Association for Symbolic Logic. *Lambda Calculus with Types*. Lambda Calculus with Types. Cambridge University Press, 2013.

[8] Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for smt-based static analysis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[9] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. *Horn Clause Solvers for Program Verification*, volume Fields of Logic and Computation II (pp.24-51), pages 24–51. Springer LNCS, 09 2015.

[10] Andreas Blass and Yuri Gurevich. *Existential fixed-point logic*, pages 20–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[11] Martin Bravenboer and Yannis 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*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[12] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, October 2009.

[13] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems*, 8(11):1143–1156, 1997.

[14] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 183–198, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus 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*, PLDI '14, page 145–155, New York, NY, USA, 2014. Association for Computing Machinery.

[16] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. What you always wanted to know about datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[17] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[18] Patrick Cousot and Radhia 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*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[19] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, 1979.

[20] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934.

[21] Haskell B. Curry, Robert Feys, and William Craig. Combinatory logic, volume i. *Philosophical Review*, 68(4):548–550, 1959.

[22] Rowan Davies and Frank Pfenning. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, USA, 2005. AAI3168521.

[23] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[24] E. Denney. Refinement types for specification. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98), Shelter Island*, pages 148–166. Chapman and Hall, 1998.

[25] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), may 2021.

[26] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth 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*, pages 172–184, 2022.

[27] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Solving constrained horn clauses using syntax and data. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[28] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.

[29] Message P Forum. Mpi: A message-passing interface standard, 1994.

[30] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 268–277, New York, NY, USA, 1991. Association for Computing Machinery.

[31] Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental λ-calculus in cache-transfer style. In Luís Caires, editor, *Programming Languages and Systems*, pages 553–580, Cham, 2019. Springer International Publishing.

[32] Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, pages 407–420, New York, NY, USA, 2016. ACM.

[33] Thomas Gilray and Sidharth Kumar. Distributed relational algebra at scale. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 12–22, 2019.

[34] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 691–704, New York, NY, USA, 2016. ACM.

[36] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo 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*, SIGMOD '19, page 467–484, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo 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*, SIGMOD '19, page 467–484, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Arie Gurfinkel. Program verification with constrained horn clauses (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 19–29, Cham, 2022. Springer International Publishing.

[39] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 881–884, New York, NY, USA, 2014. Association for Computing Machinery.

[40] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, jan 1993.

[41] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

[42] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[43] Muhammad Imran, Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Fast datalog evaluation for batch and stream graph processing. *World Wide Web*, 25(2):971–1003, mar 2022.

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

[45] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 343–350, New York, NY, USA, 2013. Association for Computing Machinery.

[46] R. Kramer, R. Gupta, and M.L. Soffa. The combining dag: a technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):805–813, 1994.

[47] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation*, 20(3):199–207, 2007.

[48] Deepa S Kumar and M Abdul Rahman. Performance evaluation of apache spark vs mpi: A practical case study on twitter sentiment analysis. *Journal of Computer Science*, 13(12):781–794, Dec 2017.

[49] 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*, pages 1–8, April 2008.

[50] Sidharth Kumar and Thomas Gilray. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019.
[51] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 288–308, Cham, 2020. Springer International Publishing.
[52] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 288–308, Cham, 2020. Springer International Publishing.
[53] Yong-fong Lee, Thomas J. Marlowe, and Barbara G. Ryder. Performing data flow analysis in parallel. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, page 942–951, Washington, DC, USA, 1990. IEEE Computer Society Press.
[54] Magnus Madsen and Ondřej Lhoták. Safe and sound program analysis with flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 38–48, New York, NY, USA, 2018. Association for Computing Machinery.
[55] Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: Programming with first-class datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.
[56] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices*, 51(6):194–208, 2016.
[57] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 428–443, New York, NY, USA, 2010. Association for Computing Machinery.
[58] Microsoft. Azure — m-series. https://docs.microsoft.com/en-us/azure/virtual-machines/m-series. Accessed: 2021-11-19.
[59] Jan Midtgaard. Control-flow analysis of functional programs. *ACM computing surveys (CSUR)*, 44(3):1–33, 2012.
[60] Matthew Might. Abstract interpreters for free. In *International Static Analysis Symposium*, SAS '10, pages 407–421. Springer, 2010.
[61] Matthew Might, Yannis Smaragdakis, and David 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*, pages 305–315, 2010.
[62] Scott Parker, Vitali Morozov, Sudheer Chunduri, Kevin Harms, Chris Knight, and Kalyan Kumaran. Early evaluation of the cray xc40 xeon phi system 'theta'at argonne. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.
[63] Frank Pfenning and Carsten Schuermann. Twelf user's guide. Technical report, version 1.2. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998.
[64] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
[65] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.
[66] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, jan 2000.
[67] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. Eigencfa: Accelerating flow analysis with gpus. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 511–522, New York, NY, USA, 2011. Association for Computing Machinery.
[68] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer Science*, 53:121–130, 2015. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.
[69] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery.
[70] Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 77–88, New York, NY, USA, 2022. Association for Computing Machinery.
[71] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. ACM.
[72] David S Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *The Sixth Distributed Memory Computing Conference, 1991. Proceedings*, pages 398–399. IEEE Computer Society, 1991.
[73] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, sep 2013.
[74] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD

'16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery.

[75] J. H. Siddiqui and S. Khurshid. Parsym: Parallel symbolic execution. In *2010 2nd International Conference on Software Technology and Engineering*, volume 1, pages V1–405–V1–409, 2010.

[76] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, October 2018.

[77] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[78] Souffle Team. Soufflé — tuning (sideways information-passing strategy). https://souffle-lang.github.io/tuning#sideways-information-passing-strategy. Accessed: 2021-11-19.

[79] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[80] George K Thiruvathukal, Cameron Christensen, Xiaoyong Jin, François Tessier, and Venkatram Vishwanath. A benchmarking study to evaluate apache spark on large-scale supercomputers. *arXiv preprint arXiv:1904.11812*, 2019.

[81] Emina Torlak and Rastislav 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*, Onward! 2013, page 135–152, New York, NY, USA, 2013. Association for Computing Machinery.

[82] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *SIGPLAN Not.*, 49(6):530–541, jun 2014.

[83] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. Implementing a classic: Zero-copy all-to-all communication with mpi datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 135–144, 2014.

[84] Patrick Valduriez and Setrag Khoshafian. Parallel evaluation of the transitive closure of a database relation. *Int. J. Parallel Program.*, 17(1):19–42, February 1988.

[85] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.

[86] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in liquid haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 132–144, New York, NY, USA, 2018. Association for Computing Machinery.

[87] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer, 2005.

[88] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, page 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[89] Andrew K Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):166–207, 1998.

[90] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16–es, may 2007.

[91] Rohit Zambre, Damodar Sahasrabudhe, Hui Zhou, Martin Berzins, Aparna Chandramowlishwaran, and Pavan Balaji. Logically parallel communication for fast mpi+ threads applications. *IEEE Transactions on Parallel and Distributed Systems*, 2021.