Noname manuscript No.
(will be inserted by the editor)

Fast Left Kan Extensions Using The Chase

David I. Spivak · Ryan Wisnesky

July 2, 2019

Abstract We present a new algorithm for computing left Kan extensions based on the venerable "chase" algorithm from relational database theory. We show how our algorithm performs a breadth-first construction of an initial term model for a particular finite-limit theory associated with each left Kan extension, and we provide experimental data demonstrating our algorithm's performance.

Keywords Computational category theory · Left Kan extensions · the Chase · Data migration · Data integration

Both authors at
Conexus AI

Contents

1 Introduction

Left Kan extensions [7] are used for many purposes in automated reasoning: to enumerate the elements of finitely-presented algebraic structures such as monoids; to construct semi-decision procedures for Thue systems; to compute the cosets of a group; to compute the orbits of a group action; to compute quotients of sets; and more.

Left Kan extensions are described category-theoretically, and we assume a knowledge of category theory [2] in this paper. Let C and D be categories and $F : C \to D$ a functor. Given a functor $J : D \to \mathsf{Set}$, where $D \to \mathsf{Set}$ (also written Set^D) is the category of functors from D to the category of sets, Set, we define $\Delta_F(J) : C \to \mathsf{Set} \;:=\; J \circ F$, and think of Δ_F as a functor from $D \to \mathsf{Set}$ to $C \to \mathsf{Set}$. Δ_F has a left adjoint, which we write as Σ_F, taking functors in $C \to \mathsf{Set}$ to functors in $D \to \mathsf{Set}$. Given a functor $I : C \to \mathsf{Set}$, the functor $\Sigma_F(I) : D \to \mathsf{Set}$ is called the *left Kan extension* [7] of I along F.

Left Kan extensions of set-valued functors always exist, up to unique isomorphism, but they need not be finite (i.e., $\Sigma_F(I)(c)$ may have infinite cardinality for some object $c \in C$). In this paper we describe how to compute finite left Kan extensions when C, D, and F are finitely presented and I is finite, a semi-computable problem originally solved in [7] and significantly improved upon in [6].

1.1 Kan Extensions for Data Migration

Our primary interest in left Kan extensions is motivated by their use in data migration [18,21,19], where C and D represent database schemas, F a "schema mapping" [12] defining a translation from C to D, and I an input C-database (often called an *instance*) that we wish to migrate to D. Our implementation of the fastest left Kan algorithm we knew of from existing literature [6] was impractical for large input instances, yet it bore a striking operational resemblance to an algorithm from relational database theory known as *the chase* [9], which is also used to solve data migration problems, and for which efficient implementations are known [4].

In this paper, we formalize the above observation and show how to efficiently compute left Kan extensions by using a chase algorithm implementation and experimentally demonstrate its time and space performance. Our algorithm is part of the open-source categorical query language CQL, available at `http://categorical.info`. Note that because Δ_F also has a right adjoint, Π_F, known as a right Kan extension, and related to database joins along F [18], the functor Σ_F is the unique "dual to join" in a precise sense.

1.2 Kan Extensions for Term Model Construction

Our secondary interest in left Kan extensions is motived by a desire to build term models of equational theories quickly. In previous work [17] we showed that a left Kan extension $\Sigma_F(I)$ can be computed using techniques from automated theorem proving. Given $F : C \to D$ and $I : C \to \mathsf{Set}$ as above, we will think of C and D not as categories, but as presentations of categories; more precisely, as multi-sorted equational theories, where each object is a sort and each generating morphism is a

function symbol of arity one. Similarly, we will think of I as an equational theory that extends C with a new constant symbol for each element of I, along with appropriate equations; and we will think of F as a *theory morphism* [15] assigning each sort in C to a sort in D and each symbol $c : c_1 \to c_2$ in C to an (open) term $F(c) : F(c_1) \to F(c_2)$ in D such that if $p = q$ is provable in C, then $F(p) = F(q)$ is provable in D.

From this perspective, the left Kan extension $\Sigma_F(I)$ is the initial model of the equational theory $F(I)$ obtained by applying F to I. To compute the initial model it is sufficient, but not necessary, to construct a decision procedure for equality in $F(I)$. Having such a decision procedure allows us to build the initial model by enumerating closed terms and adding new terms to the initial model when they are not provably equal to any terms already in the initial model.

Algorithmically, decision-procedure based approaches can use either a depth or breadth first enumeration of terms, much like how chase based algorithms can use either a sequential chase (operationally analogous to the algorithm in [6]) or a parallel chase (operationally analogous to the algorithm in this paper). However, chase-based approaches do not initially construct a decision procedure, and as a result they are often significantly faster in practice. As a result, in this paper we show how to build term models of equational (and finite-limit [22]) theories quickly, by exploiting bulk operations from database theory.

1.3 Outline

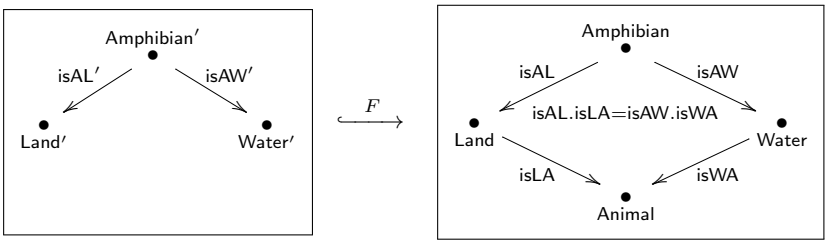
In section 2 we describe left Kan extensions and prove lemmas relating them to the chase algorithm. In section 3 we describe our particular chase algorithm implementation, compare it to the algorithm in [6], and provide experimental results. We conclude in section 4 by discussing the differences between the chase as used in relational database theory and as used in this paper.

2 Theory

2.1 Running Example

Our running example of a left Kan extension is a data integration problem that cannot be solved (for all input instances) with a single relational algebra query of fixed size: *quotienting a set by an equivalence relation*. In this example, the input data consists of amphibians, land animals, and water animals, such that every amphibian is exactly one land animal and exactly one water animal. We wish to compute all of the animals without double-counting the amphibians, which we can do by taking the disjoint union of the land animals and the water animals and then equating the two instances of each amphibian.

Our source schema C is the span $\mathsf{Land'} \leftarrow \mathsf{Amphibian'} \rightarrow \mathsf{Water'}$, our target schema D extends C into a commutative square with new sort / terminal object Animal and no ' marks, and the functor F is the inclusion:



Our input functor $I : C \to \mathsf{Set}$, displayed with one table per object, is:

Amphibian'	isAL'	isAW'
gecko	lizard	salamander
frog	toad	newt

Land'
lizard
toad
human
cow
horse

Water'
fish
salamander
newt
dolphin

Frogs are double counted as both toads and newts, and the left Kan extension equates them as animals. Similarly, geckos are both lizards and salamanders. We thus expect $5 + 4 - 2 = 7$ animals.

There are infinitely many left Kan extensions of I along F; each will be naturally isomorphic to the one below in a unique way; in other words the following tables are unique up to choice of names. The amphibians table of $\Sigma_F(I)$ is identical to that of I and is omitted:

Land	isLA
lizard	gecko
toad	frog
human	human
cow	cow
horse	horse

Water	isWA
fish	fish
salamander	gecko
newt	frog
dolphin	dolphin

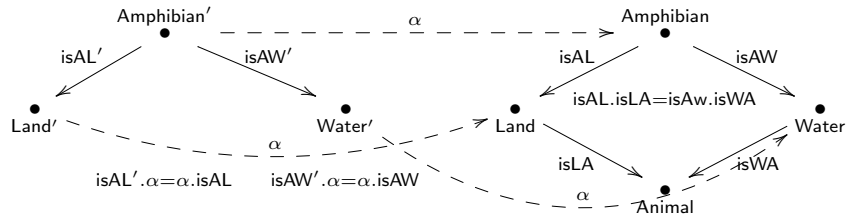
Animal
fish
frog
dolphin
human
cow
horse
gecko

Because in this example F is fully faithful, the natural transformation $\eta_I : I \to \Delta_F(\Sigma_F(I))$, i.e. the unit of $\Delta_F \dashv \Sigma_F$ adjunction, is an identity of C-instances; it associates each source Land′ animal to the same-named target Land animal, etc.

2.2 The Collage of a Functor

The *collage* [11] of a functor $F : C \to D$, written $col(F)$, is a canonical presentation of the category that "displays" F and which helps to axiomatize the natural transformation $\eta_I : I \to \Delta_F(\Sigma_F(I))$ associated with the left Kan extension of any instance I along F.

To construct $col(F)$ we first take the disjoint union of C and D. We then add a generating morphism $\alpha_c : c \to F(c)$ for each object $c \in C$, and finally we add an equation $F(f) \circ \alpha_c = \alpha_{c'} \circ f$ for each generating morphism $f : c \to c' \in C$:



The evident inclusion functors $i_C : C \to col(F)$ and $i_D : D \to col(F)$ of C and D into $col(F)$ will be used several times throughout the paper. We will also make use of the following easy propositions:

Proposition 1 *Let $F\colon C \to D$ be a functor. For objects $c \in C$ and $d \in D$, there is a bijection between hom-sets,*

$$D(F(c), d) \cong col(F)(i_C(c), i_D(d)).$$

Proposition 2 *Let $F\colon C \to D$ be a functor. The following are equivalent:*

- *the category of triples (I, J, f), with $I : C \to \mathsf{Set}$, $J : D \to \mathsf{Set}$, and $f\colon \Sigma_F(I) \to J$,*
- *the category of triples (I, J, f), with $I : C \to \mathsf{Set}$, $J : D \to \mathsf{Set}$, and $f\colon I \to \Delta_F(J)$,*
- *the category of functors $col(F) \to \mathsf{Set}$.*

2.3 Finite Limit Theories

A *finite limit theory* [22] consists of a set $s_1, \ldots, s_j$ of sorts and a set $p_1, \ldots, p_k$ of relation symbols—together these form a *signature*—as well as a set $\mathfrak{A}$ of formulae, which we call *axioms*, each having the following form:

$$\forall (x_0 : s_0) \cdots (x_n : s_n).\, \phi(x_0, \cdots, x_n) \Rightarrow \exists! (x_{n+1} : s_{n+1}) \cdots\ (x_m : s_m).\, \psi(x_0, \ldots, x_m)$$

where ϕ and ψ are (possibly empty) conjunctions of:

- assertions $x = x'$, for some variables of the same sort, or
- assertions $p(x, \ldots, x')$, for some variables of appropriate sort.

A *pre-model* I consists of a set $I(s)$ for every sort $s \in S$ and a subset $I(p) \subseteq I(s_{i_1}) \times \cdots \times I(s_{i_k})$ for every relation symbol p of arity $s_{i_1}, \ldots, s_{i_k}$. A $\mathfrak{A}$-model is a pre-model that additionally satisfies every axiom of $\mathfrak{A}$ in the obvious way.

Finite limit theories can be described using partial functions instead of relations, in which case they are often called *essentially algebraic theories* [22].

2.4 The Finite Limit Theory of a Category

We now describe how to convert a category presentation—including that for the collage $col(F)$ of any functor F—into a finite limit theory. To do so, consider the objects of C as sorts of the theory; convert each generating morphism of C to

a binary relation symbol; and add combined totality-functionality conditions for each generating morphism, for example:

$$\forall(x : \mathsf{Amphibian}).\ \exists!(y : \mathsf{Land}).\, \mathsf{IsAL}(x, y)$$

as well as the equations from C's presentation.[1] Here are the three equations from $col(F)$, one for the commutative square in D and two associated with F; see Section 2.2.

$$\mathsf{IsAL}(x, y) \wedge \mathsf{IsLA}(y, z) \wedge \mathsf{IsAW}(x, y') \wedge \mathsf{IsWA}(y', z') \Rightarrow z = z'$$
$$\mathsf{isAL}'(x, y) \wedge \alpha_{\mathsf{Land}'}(y, z) \wedge \alpha_{\mathsf{Amphibian}'}(x, y') \wedge \mathsf{IsAL}(y', z') \Rightarrow z = z'$$
$$\mathsf{isAW}'(x, y) \wedge \alpha_{\mathsf{Water}'}(y, z) \wedge \alpha_{\mathsf{Amphibian}'}(x, y') \wedge \mathsf{IsAW}(y', z') \Rightarrow z = z'$$

2.5 Chasing Embedded Dependencies

By weakening the finite-limit formula requirement that every existential quantifier be modally unique, we obtain what and what logicians call an *existential Horn clause* [14], what category theorists call a *regular formula* [22], and what database theorists call an *embedded dependency (ED)* [9], where a formula asserting equality is called *equality generating* (an EGD) and formulae asserting membership in a relation is called *tuple generating* (a TGD).

Given a regular theory (set of formulae) $\mathfrak{A}$ and a pre-model κ, to *chase* κ by $\mathfrak{A}$ is to construct a pre-model $chase_{\mathfrak{A}}(\kappa)$ and morphism $h : \kappa \to chase_{\mathfrak{A}}(\kappa)$ such that:

1. $chase_{\mathfrak{A}}(\kappa)$ satisfies $\mathfrak{A}$ (i.e., is an $\mathfrak{A}$-model).
2. for any model κ' satisfying $\mathfrak{A}$ and any morphism $h' : \kappa \to \kappa'$, there is a possibly non-unique morphism $g : chase_{\mathfrak{A}}(\kappa) \to \kappa'$ such that $g \circ h = h'$.

In the database theory literature, $chase_{\mathfrak{A}}(\kappa)$ is called a "universal solution"[9]. Note that two such universal solutions to the same problem may have different cardinalities; database theorists often identify instances κ and κ' for which there exists morphisms $\kappa \to \kappa'$ and $\kappa' \to \kappa$ even if the morphisms do not compose to the identity, a notion called "homomorphic equivalence". This deviation from traditional model-theoretic semantics is motivated by a need to distinguish input data from "null" or "missing" data constructed during data migration/integration, and is discussed further in the conclusion of this paper.

2.6 Chasing Finite Limit Theories

Universal solutions in the sense of database theory are not a tight enough solution concept to describe left Kan extensions. To obtain such a solution concept, we must appeal to the fact that all the existential quantifiers in a finite-limit theory are modally unique.

Lemma 1 *Given a finite signature (S, P) and finite set of axioms $\mathfrak{A}$, as in Section 2.3, for any pre-model κ, there exists a pre-model $chase_{\mathfrak{A}}(\kappa)$ and morphism $h : \kappa \to chase_{\mathfrak{A}}(\kappa)$ with the following properties:*

[1] From now on, we will omit universal quantifiers and sorts when they can be inferred from context, but we will continue to make existential quantifiers explicit.

1. *$chase_{\mathfrak{A}}(\kappa)$ satisfies $\mathfrak{A}$ (i.e., is an $\mathfrak{A}$-model),*
2. *for any model κ' satisfying $\mathfrak{A}$ and any morphism $h' : \kappa \to \kappa'$, there is a unique morphism $g : chase_{\mathfrak{A}}(\kappa) \to \kappa'$ such that $g \circ h = h'$.*

Proof The proof uses the theory of sketches; see [3] and [22]. Note that understanding the proof is not required to understand our chase algorithm. Given the theory $(S, P, \mathfrak{A})$ there is a category S_P and a set $\mathfrak{R}_P$ of limit cones such that the category of models for the sketch $(S_P, \mathfrak{R}_P)$ is equivalent to the category of $\mathfrak{A}$-models. Indeed, begin with the category with objects $S \sqcup P$ and a morphism $\phi \to s_i$ for each $\phi \in P$ with arity $(s_1, \ldots, s_k)$ and $1 \leq i \leq k$. Now form the free finite limit sketch on this category and add to the sketch a cone for each ϕ that enforces the unique map $\phi \to s_1 \times \cdots \times s_k$ to be a monomorphism. Finally, for each axiom

$$\forall (x_0 : s_0) \cdots (x_n : s_n).\, \phi(x_0, \cdots, x_n) \Rightarrow \exists! (x_{n+1} : s_{n+1}) \cdots\ (x_m : s_m).\, \psi(x_0, \ldots, x_m)$$

in $\mathfrak{A}$, the conjunctions ϕ and ψ are given by pullbacks, say p and q which already exist in S_P, and we finish by adding a morphism $p \to q$. The resulting category sketch is $(S_P, \mathfrak{R}_P)$, and it is tedious but not hard to show that the category of models of this sketch is equivalent to that of $\mathfrak{A}$-models.

The theorem then becomes just a restatement of the fact that the category of models of a limit sketch $(S, \mathfrak{R})$ is a reflective subcategory of the functor category Set^S; see [3, Theorem 4.2.1]. Here $chase_{\mathfrak{A}}$ is the name of the reflection functor, and given $\kappa \in \mathsf{Set}^S$, the map h is the unit of the reflection.

The properties above imply that the chase is a reflector, i.e. left adjoint to the inclusion of the category of $\mathfrak{A}$-models into the category of pre-models. In other words, $chase_{\mathfrak{A}}(\kappa)$ is an *initial object* in the category of $\mathfrak{A}$-models equipped with a map from κ. We will next show that these universal solutions can be used to compute left Kan extensions.

2.7 Left Kan Extensions Using the Chase

To compute a left Kan extension $\Sigma_F(I)$ using a chase algorithm (i.e. any algorithm that produces universal solutions to embedded dependencies), we consider I as a pre-model $\mathcal{I}$ on the theory associated to $col(F)$, compute $chase_{col(F)}(\mathcal{I})$, and then project the part we are interested in. Our main result, up to abuse of notation, is:

Lemma 2 $\Delta_{i_D}(chase_{col(F)}(\mathcal{I})) \cong \Sigma_F(I)$.

Proof Let $J := chase_{col(F)}(\mathcal{I})$; it is a C-instance. By definition, $I = \Delta_{i_C}\mathcal{I}$, so there is a map $I \to \Delta_{i_C} J$ and hence an induced map $\Sigma_{i_C} I \to J$ over $\mathcal{I}$. Also $\Sigma_{i_C} I$ contains $\mathcal{I}$, and by universality (Lemma 1 (2)), there is a unique morphism $J \to \Sigma_{i_C} I$ over $\mathcal{I}$. By a standard argument, we have an isomorphism $J \cong \Sigma_{i_C} I$.

Now it suffices to show that $\Sigma_F(I) \cong \Delta_{i_D} \circ \Sigma_{i_C}(I)$. This can be seen at a high-level of abstraction using profunctors, but at a hands-on level it follows from Proposition 1 which implies that colimit formula for both sides are the same:

$$\operatorname*{colim}_{\sum_{c \in C} D(F(c), d)} I(c) \cong \operatorname*{colim}_{\sum_{c \in C} col(F)(i_C(c), i_D(d))} I(c).$$

3 Practice

Although performant chase implementations exist [4], at the time of writing none of them were appropriate for CQL's left Kan implementation. First, not all of them support all of finite limit logic (for example, systems based on TGDs would compute nine animals on our running example [12]). Second, many do not have any mechanism for reporting the "lineage" or "provenance" necessary for us to easily construct a term model, rather than a non-term model, as output, a desirable property in data migration. And finally, many existing chase algorithms are non-deterministic, trading predictability for speed but also inhibiting mathematical analysis. For these reasons, we built a deterministic chase algorithm specialized to left Kan extensions, resembling a parallel version of the algorithm in [6].

3.1 Input Specification

The input to the categorical chase for a left Kan extension consists of:

- A finite set C, the elements of which we call *source nodes*
- For each $c_1, c_2 \in C$, a finite set $C(c_1, c_2)$, the elements of which we call source edges from c_1 to c_2. We may write $f : c_1 \to c_2$ or $c_1 \to_f c_2$ to indicate $f \in C(c_1, c_2)$.
- For each $c_1, c_2 \in C$, a finite set $CE(c_1, c_2)$ of pairs of *paths* $c_1 \to c_2$, which we call *source equations*. By a *path* $p : c_1 \to c_2$ we mean a (possibly 0-length) sequence of edges $c_1 \to \ldots \to c_2$.
- A finite set D, the elements of which we call *target nodes*
- For each $d_1, d_2 \in D$, a finite set $D(d_1, d_2)$, the elements of which we call *target edges* from d_1 to d_2.
- For each $d_1, d_2 \in D$, a finite set $DE(d_1, d_2)$ of pairs of paths $d_1 \to d_2$, which we call *target equations*.
- A function $F : C \to D$.
- For each $c_1, c_2 \in C$, a function F_{c_1,c_2} from edges in $C(c_1, c_2)$ to paths $F(c_1) \to F(c_2)$ in D. We will usually drop the subscripts on F when they are clear from context.
- For each $c \in C$, a set $I(c)$, the elements of which we call *input rows*.
- For each edge $g : c_1 \to c_2 \in C$, a function $I(c_1) \to I(c_2)$.

The above data determines category $\mathcal{C}$ (resp. $\mathcal{D}$), whose objects are nodes in C (resp. D), and whose morphisms are equivalence classes of paths in C (resp. D), modulo the equivalence relation induced by CE (resp. DE). Provided that for every two paths p_1 and $p_2 : c_1 \to c_2$ that are equivalent according to CE, the two paths $F(p_1)$ and $F(p_2)$ are equivalent according to DE, the above data determines a functor $\mathcal{F} : \mathcal{C} \to \mathcal{D}$. This semi-decidable condition on F is checked by the CQL automated theorem prover at compile time [18] and does not concern us here. Similarly, provided that $I(p_1)$ and $I(p_2)$ are equal as functions whenever paths p_1 and p_2 are provably equal according to CE, the above data determines a functor $\mathcal{I} : \mathcal{C} \to \mathsf{Set}$. This condition on I is decidable and checked by CQL at runtime. Apart from checking this condition on I, the source equations CE are not actually used by any of the left Kan or chase algorithms we are aware of, including ours.

3.2 The Chase Step

Like most chase engines, our categorical left Kan chase runs in rounds, possibly forever, transforming a state until a fixed point is reached. Termination is undecidable, but conservative criteria exist based on the acyclicity of the "firing pattern" of the existential quantifiers [9]. The state of a categorical chase for a left Kan extension consists of:

- For each $d \in D$, a set $J(d)$, the elements of which we call *output rows*. J is initialized by setting $J(d) := \bigsqcup_{\{c \in C \mid F(c)=d\}} I(c)$.
- For each edge $d \in D$, an equivalence relation $\sim_d \subseteq J(d) \times J(d)$, initialized to identity.
- For each edge $f : d_1 \to d_2 \in D$, a relation $J(f) \subseteq J(d_1) \times J(d_2)$, initialized to empty.
- For each node $c \in C$, a function $\eta(c) : I(c) \to J(F(c))$. η is initialized to the co-product/disjoint-union injections from the first item, i.e., $\eta(c)(x) = (c, x)$.

Given a path $p : d_1 \to d_2$ in D, we may *evaluate* p on any $x \in J(d_1)$, written $p(x)$, resulting in a (possibly empty) set of values from $J(d_2)$. Each round consists of the following actions, in the following sequence. The step names were chosen to be similar to those in [6]:

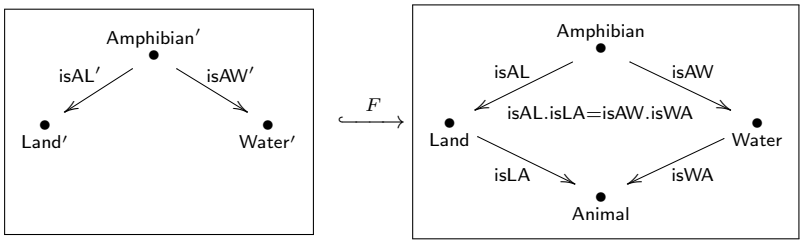
1. Action α: make all edges total. For every edge $g : d_1 \to d_2$ in D and $x \in J(d_1)$ for which there does not exist $y \in J(d_2)$ with $(x, y) \in J(g)$, add a "fresh" symbol $\mathsf{g(x)}$ to $J(d_2)$ and add $(x, \mathsf{g(x)})$ to $J(g)$.
2. Action β_D: add all "coincidences" induced by D. The phrase "add coincidences" is used by the authors of [6] where a database theorist would use the phrase "fire equality-generating dependencies". In this step, for each equation $p = q$ in $DE(d_1, d_2)$ and $x \in J(d_1)$, we update $\sim_{d_2}$ to be the smallest equivalence relation also including $\{(x', x'') \mid x' \in p(x),\ x'' \in q(x)\}$.
3. Action β_F: add all coincidences induced by F. This step is similar to the step above, except that the equation $p = q$ comes from the collage of F and evaluation requires data from η and I in addition to J.
4. Action δ: add all coincidences induced by functionality of edges. For every (x, y) and (x, y') in $J(f)$ for some $f : d_1 \to d_2$ in D with $y \neq y'$, update $\sim_{d_2}$ to be the smallest equivalence relation also including (y, y').
5. Action γ: merge coincidentally equal elements. In many chase algorithms, including [6], elements are equated in place, necessitating complex reasoning and inducing non-determinism. Our algorithm is deterministic: step 1 adds all possible new elements, and the next steps add to $\sim$. In this last step, we replace every entry in J and η with its equivalence class (or representative) from $\sim$, bypassing the need for complex reasoning and allowing parallel replacement. It is also possible to maintain $\sim$ as a list of pairs, and construct an equivalence relation all at once in this last step.

To see that this algorithm is equivalent to chasing with $col(F)$, note that each TGD or EGD in $col(F)$ is fired by one of the above actions, and that none of the above actions are taken that do not correspond to firing an ED in $col(F)$. Completeness (that our algorithm terminates whenever a finite left Kan extension exists) is still an open question that we strongly suspect to be true. The algorithm of [6] is complete; however there are theories in regular logic for which a parallel chase will

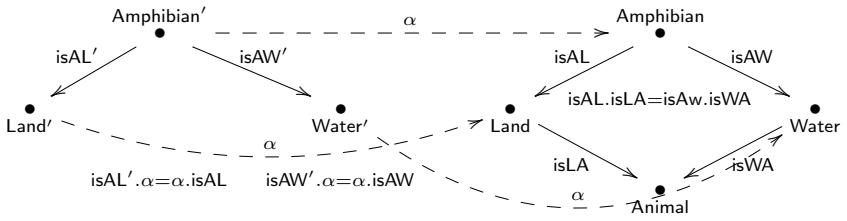
diverge but a standard chase will converge [9], so the answer is not immediately obvious. In practice, the possible divergence of a parallel chase is tolerated because of the significant speed-ups possible compared to a sequential chase.

3.3 Example Chase Sequence

Recall that our source schema C is the span Land' $\leftarrow$ Amphibian' $\rightarrow$ Water', our target schema D extends C into a commutative square with new sort / terminal object Animal and no ' marks, and the functor F is the inclusion:



The collage of F is:



Which, expressed as a finite limit theory over binary relation symbols, has combined functionality-totality conditions, for example:

$$\forall(x : \mathsf{Amphibian}).\, \exists!(y : \mathsf{Land}).\, \mathsf{IsAL}(x, y)$$

and three implications, the first from D and the other two from F:

$$\mathsf{IsAL}(x, y) \wedge \mathsf{IsLA}(y, z) \wedge \mathsf{IsAW}(x, y') \wedge \mathsf{IsWA}(y', z') \Rightarrow z = z'$$

$$\mathsf{isAL}'(x, y) \wedge \alpha_{\mathsf{Land}'}(y, z) \wedge \alpha_{\mathsf{Amphibian}'}(x, y') \wedge \mathsf{IsAL}(y', z') \Rightarrow z = z'$$

$$\mathsf{isAW}'(x, y) \wedge \alpha_{\mathsf{Water}'}(y, z) \wedge \alpha_{\mathsf{Amphibian}'}(x, y') \wedge \mathsf{IsAW}(y', z') \Rightarrow z = z'$$

Our input functor $I : C \rightarrow \mathsf{Set}$, displayed with one table per object, is:

Land'
lizard
toad
human
cow
horse

Water'
fish
salamander
newt
dolphin

Amphibian'	isAL'	isAW'
gecko	lizard	salamander
frog	toad	newt

The chase state is initialized to:

Land	isLA
lizard	
toad	
human	
cow	
horse	

Water	isWA
fish	
salamander	
newt	
dolphin	

Amphibian	isAL	isAW
gecko		
frog		

Animal

Next, we add new elements (Animal remains empty):

Land	isLA
lizard	isLA(lizard)
toad	isLA(toad)
human	isLA(human)
cow	isLA(cow)
horse	isLA(horse)

Water	isWA
fish	isWA(fish)
salamander	isWA(salamander)
newt	isWA(newt)
dolphin	isWA(dolphin)

Amphibian	isAL	isAW
gecko	isAL(gecko)	isAW(gecko)
frog	isAL(frog)	isAW(frog)

Animal

Next, we add coincidences. The single target equation in D induces no effect, because there are no Animals that can possibly be equated yet. The two naturality conditions for α essentially state that isAL and isAW should be copies of isAL' and isAW', requiring the following equivalences:

$$\mathsf{isAL}(\mathsf{gecko}) \sim \mathsf{lizard} \quad \mathsf{isAW}(\mathsf{gecko}) \sim \mathsf{salamander}$$
$$\mathsf{isAL}(\mathsf{toad}) \sim \mathsf{frog} \quad \mathsf{isAW}(\mathsf{toad}) \sim \mathsf{newt}$$

And so we end round one with no Animals and:

Land	isLA
lizard	isLA(lizard)
toad	isLA(toad)
human	isLA(human)
cow	isLA(cow)
horse	isLA(horse)

Water	isWA
fish	isWA(fish)
salamander	isWA(salamander)
newt	isWA(newt)
dolphin	isWA(dolphin)

Amphibian	isAL	isAW
gecko	lizard	salamander
frog	toad	newt

Animal

From here forward, the Amphibians table will not change, so we will not display it, and the naturality conditions on α will always be satisfied. We begin the second round by creating nine new animals:

Land	isLA
lizard	isLA(lizard)
toad	isLA(toad)
human	isLA(human)
cow	isLA (cow)
horse	isLA(horse)

Water	isWA
fish	isWA(fish)
salamander	isWA(salamander)
newt	isWA(newt)
dolphin	isWA(dolphin)

Animal
isLA(lizard)
isLA(toad)
isLA(human)
isLA (cow)
isLA(horse)
isWA(fish)
isWA(salamander)
isWA(newt)
isWA(dolphin)

The single target equation in D induces the equivalences:

$$\mathsf{isLA}(\mathsf{lizard}) \sim \mathsf{isWA}(\mathsf{salamander}) \quad \mathsf{isLA}(\mathsf{toad}) \sim \mathsf{isWA}(\mathsf{newt})$$

for a final result of:

Land	isLA
lizard	isLA(lizard)
toad	isLA(toad)
human	isLA(human)
cow	isLA (cow)
horse	isLA(horse)

Water	isWA
fish	isWA(fish)
salamander	isWA(salamander)
newt	isWA(newt)
dolphin	isWA(dolphin)

Animal
isLA(lizard)
isLA(toad)
isLA(human)
isLA (cow)
isLA(horse)
isWA(fish)
isWA(dolphin)

This is obviously uniquely isomorphic to the original example output:

Land	isLA
lizard	lizard
toad	frog
human	human
cow	cow
horse	horse

Water	isWA
fish	fish
salamander	lizard
newt	frog
dolphin	dolphin

Animal
fish
frog
dolphin
human
cow
horse
gecko

The actual choice of names in the above tables is not canonical, as we would expect for a set-valued functor defined by a universal property, and different naming "strategies" are possible. In our categorical approach to data migration, we treat names not as values per se, but as meaningless identifiers, a choice elaborated upon in this paper's conclusion.

3.4 Comparison to Previous Work

The authors of [6] identify four actions that leave invariant the left Kan extension denoted by a state, and consider a run of the chase algorithm to be any "fair" sequence of these actions:

1. Action α: add a new element. This step is similar to our α step, except it only adds one element.
2. Action β: add a coincidence. This step is similar to our β_F and β_D, except it only considers one equation.
3. Action δ: delete non-determinism. This is similar to our δ step, except it only applies to one edge at a time. If $(x, y) \in P(g)$ and $(x, y') \in P(g)$ but $y \neq y'$, add (y, y') and (y', y) to $\sim$ and delete (x, y') from $P(g)$. This process is biased towards keeping older values to ensure fairness.
4. Action γ: delete a coincidence. If $(x, y) \in \sim_d$ for some $d \in D$, then replace y by x in various places, and add new coincidences. In the first computational left Kan paper [7], this action took an entire companion technical report to

justify [8]; the authors of [6] reduced this step to about a page. One reason this step is complicated to write in [6] is because the relation $\sim$ is not required to be transitive; another reason is that the way deletion is done in the various places depends on the particular place; another is that deletion is done in place.

Similar to the algorithm in [6], our algorithm generalizes to product categories, because left Kan extensions for product categories can be axiomatized as finite limit theories (where some symbols may have arity > 1). Readers porting the functionality from the CQL implementation in the next section to the sequential algorithm above should note that ensuring the fairness condition of action δ above requires making the path-compression strategy for $\sim$ aware of the age of each output row.

3.5 Implementation in CQL

Our CQL implementation minimizes memory usage of the algorithm sketched above by storing cardinalities instead of meaningless identifiers and using lists instead of sets, and so a CQL left Kan chase state as benchmarked in this paper consists of:

1. For each $d \in D$, a number $J(d) \geq 0$.
2. For each $d \in D$, a list of length $J(d)$, where each element has the form (c, x, p), for some $c \in C$, $x \in I(c)$, and $p : F(c) \to d$.
3. For each $d \in D$, a union-find data structure [16] based on path-compressed trees $\sim_d \subseteq J(d) \times J(d)$ [20].
4. For each edge $f : d_1 \to d_2 \in D$, a list of length $J(d_1)$, each element of which is a number between 0 and $J(d_2)$.
5. For each $c \in C$, a function $\eta(c) : I(c) \to J(F(c))$.

From a theoretical viewpoint, the above state is more precisely considered as a functor to the *skeleton* [2] of the category of sets. The CQL implementation runs the Java garbage collector between rounds, uses "hash-consed" [1], tree-based terms, and uses strings for symbol and variable names.

3.6 Performance in CQL

Scalability tests, for both time (rows/sec) and space (rows/mb of RAM) based on randomly constructed instances of the running example taken on a 13" 2018 MacBook Air with a 1.6ghz i5 CPU and 16gb RAM, on Oracle Java 11, are shown in Figure 3.6. Perhaps not as familiar as time throughput, memory throughput, measured here in rows/mb, measures the memory used by the algorithm during its execution as a function of input size; the periodic spikes in Figure 3.6 are likely do to the "double when size exceeded" behavior of the many hash-set and hash-map data structures [20] in our Java implementation. Memory throughput improves as the input gets larger, we believe, because the path-compressed union-find data structure of item 3 above scales logarithmically in space. Time throughput (rows / sec) gets worse as the input gets larger, we believe, because that same union-find structure scales linearly times logarithmically in time. Although performance on random instances may not be representative of performance in practice, our algorithm is fast enough to support multi-gigabyte real-world use cases, such as [5].

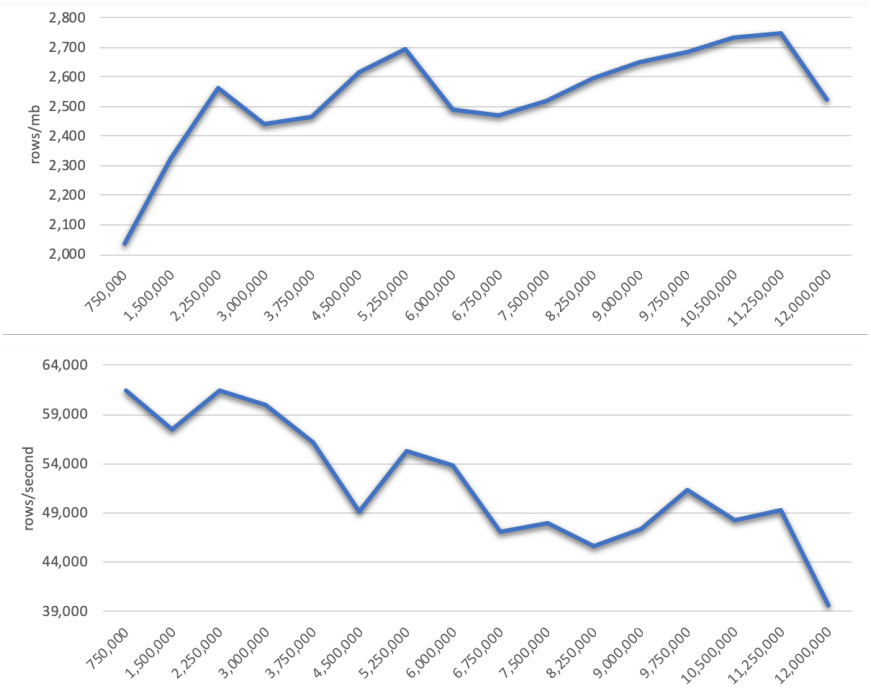


Fig. 1 Left Kan Chase Throughput, Pushout of Sets

To demonstrate the significant speed-up of our algorithm compared to all the other algorithms we are aware of, Figure 3.6 shows time throughput for the same experiment using three *previous* algorithms: the "substitute and saturate" algorithm of [17] using either specialized Knuth-Bendix completion ("monoidal" [13]) or congruence closure [16], and the sequential chase-like algorithm of [6]. Each algorithm is 1-2 orders of magnitude faster than the last, although the three algorithms in the last image do not target the skeleton of the category of sets, as our algorithm from the first two images does. All the algorithms in that figure share hash-consed terms and the other non-trivial implementation techniques.

We expect chase engines designed by the database community to soon exceed the performance of our algorithm, at least for the left Kan extensions we encounter in data migration. The reason is that techniques based on indexing and statistical query optimization such as found in many SQL engines work well for computing right Kan extensions, which correspond to joins and selections and projections, and these same techniques tend to enable chase engine performance [4]. We hope that this paper encourages the development of chase engines using the fully deterministic (up to isomorphism) parallel chase strategy described above, and believe that the reduction of left Kan extensions to chases in general, rather than our new chase algorithm above, will be the longer-lived contribution of this paper.

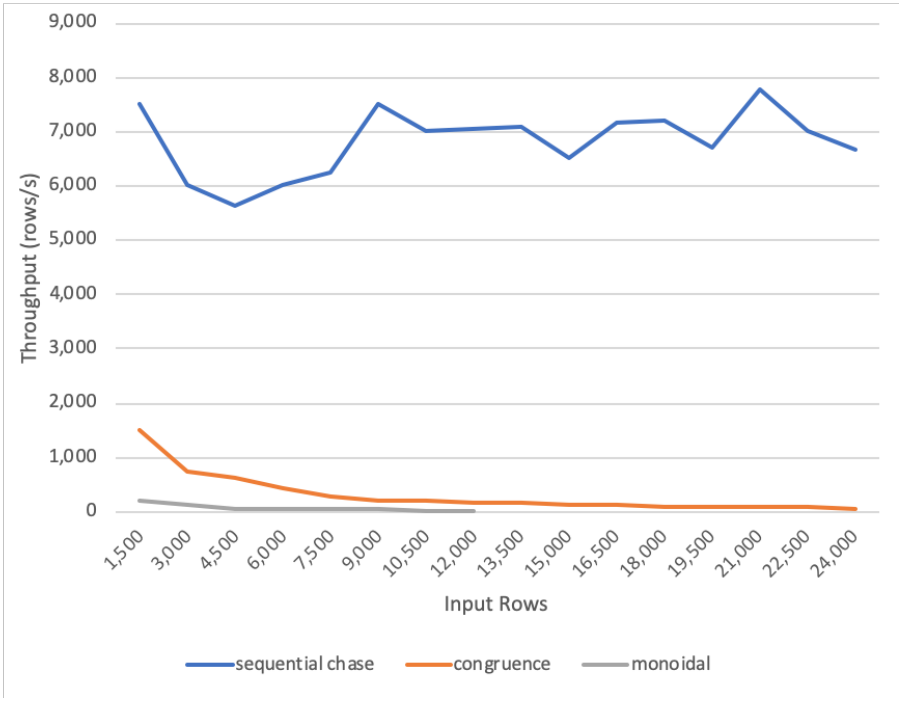


Fig. 2 Prior Left Kan Throughput, Pushout of Sets

4 Conclusion: Left Kan Extensions and Database Theory

We conclude by briefly summarizing how our use of the chase in this paper relates to its use in data migration. Readers not interested in data migration may skip this section.

In data migration, instances are defined to hold two kinds of values: *constants* and *labelled nulls*. Constants are regarded as having inherent meaning, such as numerals 1 or 2 or a social security number; nulls, sometimes called *Skolem variables* [10], can be created during migration and are regarded as distinct from constants and not innately meaningful; they are considered up to isomorphism. In particular, in data migration, when an equality-generating dependency $n = c$ is encountered, where n is a null and c a constant, then n is replaced by c, and never vice-versa; moreover if $c = c'$ is encountered, where c and c' are distinct constants, then the chase *fails*. Hence, from the data migration point of view, because our chases never fail, our instances in this paper must be thought of as being made entirely of nulls. The many consequences of adopting an unfailing, nulls-only chase procedure in the context of data migration are explored in [18, 21, 19].

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, New York, NY, USA (1998)
2. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
3. Barr, M., Wells, C.: Toposes, Triples and Theories (2002). URL http://www.cwru.edu/artsci/math/wells/pub/ttt.html
4. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '17, pp. 37–52. ACM, New York, NY, USA (2017). URL http://doi.acm.org/10.1145/3034786.3034796
5. Brown, K.S., Spivak, D.I., Wisnesky, R.: Categorical data integration for computational science. Computational Materials Science **164**, 127 – 132 (2019). URL http://www.sciencedirect.com/science/article/pii/S0927025619302046
6. Bush, M.R., Leeming, M., Walters, R.F.C.: Computing left Kan extensions. J. Symb. Comput. **35**(2), 107–126 (2003). URL http://dx.doi.org/10.1016/S0747-7171(02)00102-5
7. Carmody, S., Leeming, M., Walters, R.: The Todd-Coxeter procedure and left Kan extensions. J. Symb. Comput. **19**(5), 459–488 (1995). URL http://dx.doi.org/10.1006/jsco.1995.1027
8. Carmody, S., Walters, R.F.C.: Computing quotients of actions of a free category. In: A. Carboni, M.C. Pedicchio, G. Rosolini (eds.) Category Theory, pp. 63–78. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
9. Deutsch, A., Nash, A., Remmel, J.: The chase revisited. In: Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '08, pp. 149–158. ACM, New York, NY, USA (2008). URL http://doi.acm.org/10.1145/1376916.1376938
10. Doan, A., Halevy, A., Ives, Z.: Principles of Data Integration, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012)
11. Garner, R., Shulman, M.: Enriched categories as a free cocompletion. Advances in Mathematics **289**, 1 – 94 (2016). URL http://www.sciencedirect.com/science/article/pii/S0001870815004715
12. Haas, L.M., Hernández, M.A., Ho, H., Popa, L., Roth, M.: Clio grows up: From research prototype to industrial tool. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, pp. 805–810. ACM, New York, NY, USA (2005). URL http://doi.acm.org/10.1145/1066157.1066252
13. Kapur, D., Narendran, P.: The Knuth-Bendix completion procedure and Thue systems. SIAM Journal on Computing **14**(4) (1985)
14. Makowsky, J.: Why horn formulas matter in computer science: Initial structures and generic examples. Journal of Computer and System Sciences **34**(2), 266 – 292 (1987). URL http://www.sciencedirect.com/science/article/pii/0022000087900274
15. Mossakowski, T., Krumnack, U., Maibaum, T.: What is a derived signature morphism? In: M. Codescu, R. Diaconescu, I. Ţuţu (eds.) Recent Trends in Algebraic Development Techniques, pp. 90–109. Springer International Publishing, Cham (2015)
16. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980). URL http://doi.acm.org/10.1145/322186.322198
17. Schultz, P., Spivak, D.I., Vasilakopoulou, C., Wisnesky, R.: Algebraic databases. Theory and Applications of Categories **32**(16), 547–619 (2017)
18. Schultz, P., Spivak, D.I., Wisnesky, R.: Algebraic model management: A survey. In: P. James, M. Roggenbach (eds.) Recent Trends in Algebraic Development Techniques, pp. 56–69. Springer International Publishing, Cham (2017)
19. Schultz, P., Wisnesky, R.: Algebraic data integration. Journal of Functional Programming **27**, e24 (2017)
20. Sedgewick, R., Wayne, K.: Algorithms, 4th edn. Addison-Wesley Professional (2011)
21. Spivak, D.I., Wisnesky, R.: Relational foundations for functorial data migration. In: Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015, pp. 21–28. ACM, New York, NY, USA (2015). URL http://doi.acm.org/10.1145/2815072.2815075
22. Wells, C.: Sketches: Outline with references. In: Dept. of Computer Science, Katholieke Universiteit Leuven (1994)