DEDALUS: Datalog in Time and Space

Peter Alvaro ∗ William R. Marczak ∗ Neil Conway ∗
Joseph M. Hellerstein ∗ David Maier ∗ Russell Sears ∗

ABSTRACT
Recent research has explored using Datalog-based languages to express a distributed system as a set of logical invariants [3, 28]. Two properties of distributed systems proved difficult to model in Datalog. First, the state of any such system evolves with its execution. Second, deductions in these systems may be arbitrarily delayed, dropped, or reordered by the unreliable network links they must traverse. Previous efforts addressed the former by extending the Datalog model to admit updates, key constraints, persistence and events, and the latter by assuming ordered and reliable delivery while ignoring delay. These details have a semantics outside Datalog, which increases the complexity of the language or its interpretation, and forces programmers to think operationally. We argue that the missing component from these previous languages is a notion of time. In this paper we present DEDALUS, a foundation language for programming and reasoning about distributed systems. DEDALUS reduces to a subset of Datalog [41] with negation, aggregate functions, successor and choice, and admits an explicit representation of time into the logic language. We show that DEDALUS provides a declarative foundation for the two signature features of distributed systems: mutable state, and asynchronous processing and communication. Given these two features, we address three important properties of programs in a domain-specific manner: a notion of safety appropriate to non-terminating computations, stratified monotonic reasoning with negation over time, and efficient evaluation over time via a simple execution strategy. We also provide conservative syntactic checks for our temporal notions of safety and stratification. Our experience implementing full-featured systems in variants of Datalog suggests that DEDALUS is well-suited to the specification of rich distributed services and protocols, and provides both cleaner semantics and richer tests of correctness. We validate that DEDALUS is sufficiently expressive to represent interesting security features in a distributed system through a case study in which we implement an extensible taint tracking framework and a stateful firewall.

1. INTRODUCTION
In recent years, there has been a resurgence of interest in Datalog as the foundation for applied, domain-specific languages in a wide variety of areas, including networking [29], distributed systems [7, 13], natural language processing [18], robotics [5], compiler analysis [22], security [26, 44, 21] and computer games [42]. The resulting languages have been promoted for their compact and natural representations of tasks in their respective domains, in many cases leading to code that is orders of magnitude shorter than equivalent imperative programs. Another stated advantage of these languages has been the ability to directly capture intuitive specifications of protocols and programs as executable code.

While most of these efforts were intended to be “declarative” languages, many chose to extend Datalog with operational features natural to their application domain. These operational aspects, though familiar, limit the ability of the language designers to leverage the rich literature on Datalog: program checks like safety and stratifiability, and optimizations like magic sets and materialized recursive view maintenance. In addition, in many of these languages the blend of operational and declarative constructs leads to semantic ambiguities. This is of particular interest to us in the context of networking and other distributed systems, both because we have considerable practical experience with these languages [3, 29], and because others have examined the semantic ambiguities of these languages in some depth [32, 36].

In this paper we reconsider declarative programming for distributed systems from a model-theoretic perspective. We introduce a declarative language called DEDALUS that enables the specification of rich distributed systems concepts without recourse to operational constructs. DEDALUS is a subset of a language with well-studied features: Datalog enhanced with negation, aggregate functions, choice, and a successor relation. DEDALUS provides a model-theoretic foundation for the two key features of distributed systems: mutable state, and asynchronous processing and communication. We show how these features are captured in DEDALUS via a natural incorporation of time as an attribute of Datalog predicates.

Given the ability to express programs with these two features, we address three important properties of DEDALUS programs: a temporal notion of safety appropriate to long-running services and protocols, stratified monotonic reasoning with negation over time, and efficient evaluation via a simple execution strategy. We also provide conservative syntactic checks for our temporal notions of safety and stratification.

1DEDALUS is intended as a precursor language for Bloom, a high-level language for programming distributed systems that will replace Overlog in the BOOM project [3]. As such, it is derived from the character Stephen Dedalus in James Joyce’s Ulysses, whose dense and precise chapters precede those of the novel’s hero, Leopold Bloom. The character Dedalus, in turn, was partly derived from Daedalus, the greatest of the Greek engineers and father of Icarus. Unlike Overlog, which flew too close to the sun, Dedalus remains firmly grounded.
We begin by defining \textsc{deDalus}_0, a restricted sublanguage of Datalog (Section 2). We show how \textsc{deDalus}_0 supports state update in Section 3, prove temporal safety and stratifiability properties of \textsc{deDalus}_0 in Section 4, and describe a simple, efficient evaluation scheme (Section 5). We introduce \textsc{deDalus} by adding support for asynchrony to \textsc{deDalus}_0 in Section 6. Finally, we illustrate the use of \textsc{deDalus} to express security features in a distributed system via case studies of an extensible taint tracking system and a stateful firewall implemented as an expert system. Throughout, we demonstrate the expressivity and practical utility of our work with specific examples, including a number of building-block routines from classical distributed computing, such as sequences, queues, distributed clocks, and reliable broadcast. We also discuss the correspondence between \textsc{deDalus} and our prior work implementing full-featured distributed services in somewhat more operational Datalog variants [3, 29].

2. \textsc{deDalus}_0

We take as our foundation the language Datalog—[41]: Datalog enhanced with negated subgoals. We will be interested in the classes of statically stratifiable and modularly stratifiable [37] programs, which we revisit below. For conciseness, when we refer to “Datalog” below our intent is to admit negation—i.e., Datalog⇒.

As a matter of notation, we refer to a countably infinite universe of constants C—in which C_1 C_2, . . . are representations of individual constants—and a countably infinite universe of variable symbols \textcal{A} = A_1, A_2, . . . . We will capture time in \textsc{deDalus}_0 via an infinite relation successor isomorphic to the successor relation on the integers; for convenience we will in fact refer to the domain \textsc{Z} when discussing time, though no specific interpretation of the symbols in \textsc{Z} is intended beyond the fact that successor(x, y) is true iff y = x + 1.

2.1 Syntactic Restrictions

\textsc{deDalus}_0 is a restricted sublanguage of Datalog. Specifically, we restrict the admissible schemata and the form of rules with the four constraints that follow.

\textbf{Schema:} We require that the final attribute of every \textsc{deDalus}_0 predicate range over the domain \textsc{Z}. In a typical interpretation, \textsc{deDalus}_0 programs will use this final attribute to connote a “timestamp,” so we refer to this attribute as the time suffix of the corresponding predicate.

\textbf{Time Suffix:} In a well-formed \textsc{deDalus}_0 rule, every subgoal uses the same existential variable \textsc{T} as its time suffix. A well-formed \textsc{deDalus}_0 rule must also have a time suffix \textsc{S} as its rightmost head attribute, which must be constrained in exactly one of the following two ways:

1. The rule is said to be \textit{deductive} if \textsc{S} is bound to the value \textsc{T}; that is, the body contains the subgoal \textsc{S} = \textsc{T}.

2. The rule is said to be \textit{inductive} if \textsc{S} is the successor of \textsc{T}; that is, the body contains the subgoal successor(\textsc{T}, \textsc{S}).

In Section 6 when we consider \textsc{deDalus}—a superset of \textsc{deDalus}_0—we will introduce a third kind of rule to capture asynchrony.

\textbf{Example 1.} The following are examples of well-formed deductive and inductive rules, respectively:

deductive: p(A, B, S) ← e(A, B, T), S = T;

inductive: q(A, B, S) ← e(A, B, T), successor(T, S);

2.2 Abbreviated Syntax and Temporal Interpretation

We have been careful to define \textsc{deDalus}_0 as a subset of Datalog; this inclusion allows us to take advantage of Datalog’s well-known semantics and the rich literature on the language.

\textsc{deDalus}_0 programs are intended to capture temporal semantics. For example, a fact, p(C_1 . . . C_n), with some constant C_{n+1} in its time suffix can be thought of as a fact that is true “at time C_{n+1}”. Deductive rules can be seen as instantaneous statements: their deducitons hold for predicates agreeing in the time suffix and describe what is true “for an instant” given what is known at that instant. Inductive rules are temporal—their consequents are defined to be true “at a different time” than their antecedents.

To simplify \textsc{deDalus}_0 notation for this typical interpretation, we introduce some syntactic “sugar” as follows:

- \textit{Implicit time-suffixes in body predicates:} Since each body predicate of a well-formed rule has an existential variable \textsc{T} in its time suffix, we optionally omit the time suffix from each body predicate and treat it as implicit.

- \textit{Temporal head annotation:} Since the time suffix in a head predicate may be either equal to \textsc{T}, or equal to \textsc{T}’s successor, we omit the time suffix from the head—and its relevant constraints from the body—and instead attach an identifier to the head predicate of each temporal rules, to indicate the change in time suffix. A temporal head predicate is of the form:

r(A_1, A_2, [ . . . , A_n, S) ← r(A_1, A_2, [ . . . , A_n, T) \iff S=T;

The identifier \#next stands in for successor(\textsc{T}, \textsc{S}) in the body.

\textbf{Example 2.} The following are “sugared” versions of deductive and inductive rules:

deductive: p2(A, B) ← e(A, B);

inductive: q2(A, B) \#next ← e(A, B);

Guarded EDB: No well-formed \textsc{deDalus}_0 rule may involve any extensional predicate, except for a rule of the form above.

3. STATE IN LOGIC

\textit{Time is a device that was invented to keep everything from happening at once.}°

°Graffiti on a wall at Cambridge University [2].
Given our definition of DEDALUS₀, we now address the persistence and mutability of data across time—one of the two signature features of distributed systems for which we provide a model-theoretic foundation.

The intuition behind DEDALUS’s successor relation is that it models the passage of (logical) time. In our discussion, we will say that ground atoms with lower time suffixes occur “before” atoms with higher ones. The constraints we imposed on DEDALUS rules restrict how deductions may be made with respect to time. First, rules may only refer to a single time suffix variable in their body, and hence cannot join across different “timesteps”. Second, rules may specify deductions that occur concurrently with their ground facts or in the next timestep—in DEDALUS₀, we rule out induction “backwards” in time or “skipping” into the future.

This notion of time allows us to consider the contents of the EDB—and hence a minimal model of the DDB—with respect to an “instant in time”: we simply bind the time suffix (T) of all body predicates to a constant. Because this produces a sequence of models (one per timestep), it gives us an intuitive and unambiguous way to declaratively express persistence and state changes across time. In this section, we give examples of language constructs that capture state-oriented motifs such as persistent relations, deletion and update, sequences, and queues.

3.1 Simple Persistence

A fact in predicate p at time T may provide ground for deductive rules at time T, as well as ground for deductive rules in timesteps greater than T, provided there exists a simple persistence rule of the form:

\[ \text{p}_\text{pos}(A_1, A_2, \ldots, A_n) @ T + 1 \leftarrow \text{p}_\text{pos}(A_1, A_2, \ldots, A_n); \]

A simple persistence rule ensures that a p fact true at time i will be true \( \forall j \in \mathbb{Z}; j \geq i \).

3.2 Mutable State

To model deletions and updates of a fact, it is necessary to break the induction in a simple persistence rule. Adding a \( \text{p}_\text{neg} \) subgoal to the body of a simple persistence rule accomplishes this:

\[ \text{p}_\text{pos}(A_1, A_2, \ldots, A_n) @ T + 1 \leftarrow \neg \text{p}_\text{pos}(A_1, A_2, \ldots, A_n), \neg \text{p}_\text{neg}(A_1, A_2, \ldots, A_n); \]

If, at any time k, we have a fact \( \text{p}_\text{neg}(C_1, C_2, \ldots, C_n) @ k \), then we do not deduce a \( \text{p}_\text{pos}(C_1, C_2, \ldots, C_n) @ k+1 \) fact. By induction, we do not deduce a \( \text{p}_\text{pos}(C_1, C_2, \ldots, C_n) @ j \) fact for any \( j > k \), unless this \( \text{p}_\text{pos} \) fact is re-derived at some timestep \( j < k \) by another rule. This corresponds to the intuition that a persistent fact, once stated, is true until it is retracted.

Example 3. Consider the following DEDALUS₀ program and ground facts:

\[ \text{p}_\text{pos}(A, B) \leftarrow \text{p}(A, B); \]

\[ \text{p}_\text{pos}(A, B) @ T \leftarrow \text{p}_\text{pos}(A, B), \neg \text{p}_\text{neg}(A, B); \]

\[ \text{p}(1, 2) @ 0101; \]

\[ \text{p}(1, 3) @ 0102; \]

\[ \text{p}_\text{neg}(1, 2) @ 3000; \]

It is easy to see that the following facts are true: \( \text{p}(1, 2) @ 0200 \), \( \text{p}(1, 3) @ 0200 \), \( \text{p}(1, 3) @ 3000 \). However, \( \text{p}(1, 2) @ 0301 \) is false because it was “deleted” at timestep 300.

3.3 Sequences

Since mutable persistence occurs frequently in practice, we provide the \texttt{persist} macro, which takes three arguments: a predicate name, the name of another predicate to hold “deleted” facts, and the (matching) arity of the two predicates. The macro expands to the corresponding mutable persistence rule. For example, the above \texttt{p_pos} persistence rule may be equivalently specified as \texttt{persist [p_pos, p_neg, 2]}.

Mutable persistence rules enable updates. For some time \( T \), an update is any pair of facts:

\[ \text{p}_\text{neg}(C_1, C_2, \ldots, C_n) @ T; \]

\[ \text{p}_\text{pos}(D_1, D_2, \ldots, D_m) @ T + 1; \]

Intuitively, an update represents deleting an old value of a tuple and inserting a new value. Every update is atomic across timesteps, meaning that the old value ceases to exist at the same timestep in which the new value is derived—timestep \( T + 1 \) in the above definition.

3.4 Queues

While sequences are useful constructs for generating or imposing an ordering on tuples, programs will in some cases require that tuples are processed in a particular (partial) order associated with particular timesteps. To this end, we introduce a queue template, which employs inductive persistence and aggregate functions in rule heads to process tuples according to a data-dependent order, rather than as a set.

Aggregate functions simplify our discussion of queues. Mummick and Shmueli observe correspondences in the expressivity of Datalog with stratified negation and stratified aggregation functions [35]. Adding aggregation to our language does not affect its expressive power, but is useful for writing natural constructs for distributed computing including queues and ordering.

In DEDALUS₀, we will allow aggregate functions \( p_1 - p_n \) to appear in the head of a deductive rule of the form:

\[ \text{p}(A_1, \ldots, A_n, p_1(A_{n+1}), \ldots, p_m(A_{n+m})) \leftarrow \eta_1(A_1, \ldots, A_n, \eta_1), \ldots, \eta_m(A_1, \ldots, A_n, \eta_m); \]

According to this rule, the predicate \( p \) contains one row for each satisfying assignment of \( A_1, \ldots, A_n \)—akin to the distinct “groups” of SQL’s “GROUP BY” notation.

Consider a predicate \texttt{priority_queue} that represents a series of tasks to be performed in some predefined order. Its attributes are a string representing a user, a job, and an integer indicating the priority of the job in the queue:

\texttt{priority_queue('bob', 'bash', 200)@123; priority_queue('eve', 'ssh', 201)@123; priority_queue('alice', 'bash', 202)@123; priority_queue('bob', 'ssh', 205)@123;}

Note that all the time suffixes are the same. Given this schema, we note that a program would likely want to process \texttt{priority_queue}
events individually in a data-dependent order, in spite of their coincidence in logical time.

In the program below, we define a table \texttt{m_priority_queue} that serves as a queue to feed \texttt{priority_queue}. The queue must persist across timesteps because it may take multiple timesteps to drain it. At each timestep, for each value of \texttt{A}, a single tuple is projected into \texttt{priority_queue} and deleted (atomic with the projection) from \texttt{m_priority_queue}, changing the value of the aggregate calculated at the subsequent step:

\begin{verbatim}
persist[m_priority_queue, del_m_priority_queue, 3]
  % find the min priority
  omin(A, min<C>) <- m_priority_queue(A, _, C);
  % feed p in the next step
  % with the items of min priority
  p(A, B, C)@next <- m_priority_queue(A, B, C), omin(A, C);
  % delete from the next step
  % those items of min priority
  del_m_priority_queue(A, B, C) <- m_priority_queue(A, B, C), omin(A, C);
\end{verbatim}

Under such a queueing discipline, deductive rules that depend on \texttt{priority_queue} are constrained to consider only min-priority tuples at each timestep per value of the variable \texttt{A}, thus implementing a per-user FIFO discipline. To enforce a global FIFO ordering over \texttt{priority_queue}, we may redefine \texttt{omin} and any dependent rules to exclude the \texttt{A} attribute.

A queue establishes a mapping between \texttt{Dedalus}\textsubscript{0}'s timesteps and the priority-ordering attribute of the input relation. By doing so, we take advantage of the monotonic property of timestamps to enforce an ordering property over our input that is otherwise very difficult to express in a logic language. We return to this idea in our discussion of temporal "entanglement" Section 6.5.2.

4. SAFETY AND STRATIFICATION

In the previous section we demonstrated that \texttt{Dedalus}\textsubscript{0} can capture intuitive notions of persistence and mutability of state via a stylized use of Datalog. However, the alert reader will note that even very simple \texttt{Dedalus}\textsubscript{0} programs make for unusual Datalog: among other concerns, persistence rules produce derivations for an infinite number of values of the time suffix. Traditional Datalog interpreters, which work against static databases, would attempt to enumerate these values, making this approach impractical.

However, in the context of distributed systems and networks, the need for non-terminating "services" or "protocols" is very common. In this section we show that expressing distributed systems properties such as persistence and mutable state in logic does not require dispensing with familiar notions of safety and stratification: we take traditional notions of acceptable Datalog programs, and extend them in a way that admits sensible non-terminating programs.

4.1 Stratification in \texttt{Dedalus}\textsubscript{0}

We next turn our attention to the semantics of programs with negation. As we will see, the inclusion of time introduces a "source of monotonicity" in programs that allows for clean minimal model semantics in some surprising cases, and enables purely syntactic monotonicity checks for a broad class of temporal programs.

**Lemma 1.** A \texttt{Dedalus}\textsubscript{0} program without negation has a unique minimal model.

**Proof.** A \texttt{Dedalus}\textsubscript{0} program without negation is a pure Datalog program. Every pure Datalog program has a unique minimal model. \qed

We define syntactic stratification of a \texttt{Dedalus}\textsubscript{0} program the same way it is defined for a Datalog program:

**Definition 1.** A \texttt{Dedalus}\textsubscript{0} program is syntactically stratifiable if there exists no cycle with a negative edge in the program's predicate dependency graph.

We may evaluate such a program in stratum order as described in the Datalog literature [41]. It is easy to see that any syntactically stratified \texttt{Dedalus}\textsubscript{0} instance has a unique minimal model because it is a syntactically stratified Datalog program.

However, many programs we are interested in expressing are not syntactically stratifiable. Fortunately, we are able to define a syntactically checkable notion of temporal stratifiability of \texttt{Dedalus}\textsubscript{0} programs that maps to a subset of modularly stratifiable [37] Datalog programs.

**Definition 2.** The deductive reduction of a \texttt{Dedalus}\textsubscript{0} program \(P\) is the subset of \(P\) consisting of exactly the deductive rules in \(P\).

**Definition 3.** A \texttt{Dedalus}\textsubscript{0} program is temporally stratifiable if its deductive reduction is syntactically stratifiable.

**Lemma 2.** Any temporally stratifiable \texttt{Dedalus}\textsubscript{0} instance \(P\) has a unique minimal model.

**Proof.** **Case 1:** \(P\) consists of only deductive rules. In this case, \(P\)'s deductive reduction is \(P\) itself. We know \(P\) is syntactically stratifiable, thus it has a unique minimal model.

**Case 2:** \(P\) consists of both deductive and inductive rules. Assume that \(P\) does not have a unique minimal model. This implies that \(P\) is not syntactically stratifiable. Thus, there must exist some cycle through at least one predicate \(q\) involving negation. Furthermore, this cycle must involve an inductive rule, as \(P\) is temporally stratified.

Since the time suffix in the head of an inductive rule is strictly greater than the time suffix of its body, no atom may depend negatively on itself—it may only depend negatively on atoms in the previous timestep. Thus, \(P\) is modularly stratified over time, using the definition of modular stratification according to Ross et al. [37]. This guarantees a unique minimal model achievable via standard bottom-up fixpoint execution. \qed

**Example 4.** A simple temporally stratifiable \texttt{Dedalus}\textsubscript{0} program that is not syntactically stratifiable.

\begin{verbatim}
persist[p, p_neg, 3]
r1
  p(A, B, T) <- insert_p(A, B, T);

r2
  p_neg(A, B, T) <- p(A, B, T), delete_p(T);
\end{verbatim}

In the \texttt{Dedalus}\textsubscript{0} program above, insert\textunderscore p and delete\textunderscore p are captured in EDB relations. This reasonable program is unstratifiable because \(p > p\_neg \land p\_neg > p\). But because the successor relation is constrained such that \(\forall A,B.\ successor(A, B) \rightarrow B > A\), any such program is modularly stratified on successor. Therefore, we have \(p \not\models^+ p\_neg \not\models^+ p_{\text{out}}\); informally, earlier values do not depend on later values.
4.2 Temporal Safety

We begin by considering the issue of finite results raised above. In traditional Datalog, this is a well-studied concern. A Datalog program is considered safe if it has a finite minimal model, and hence has a finite execution. Safety in Datalog is traditionally ensured through the following syntactic constraints:

1. No functions are allowed.
2. Variables are range restricted: all attributes of the head goal appear in a non-negated body subgoal.
3. The EDB is finite.

These constraints ensure that the Herbrand Universe is finite: any atom that may be deduced by a safe program may only take its attributes from the set of all constant symbols appearing in the program and EDB. In fact, the set of all possible assignments of these constants to predicate attributes, representing every possible interpretation, is itself finite.

Since our definition of successor violates these rules, and indeed leads to an infinite set of facts, Dedalus₀ programs violate this definition of safety. However, successor models time, not computation; later sections explain how Dedalus implementations avoid enumerating the contents of successor at runtime. This section introduces a notion of termination that allows us to reason about the safety of Dedalus₀ programs.

A Dedalus₀ program containing only deductive rules is informally equivalent to a Datalog program in which all predicates have no time suffix. If all the rules in such a program meet the conditions above, then clearly we would like them to meet Dedalus₀'s definition of safety.

Definition 4. A rule is instantaneously safe if it is deductive, function-free and range-restricted. A Dedalus₀ program is instantaneously safe if its deductive reduction is instantaneously safe.

The successor relation complicates the discussion of safety, as it introduces the countably infinite set \( \mathbb{Z} \) to our universe of constants.

Consider the following Dedalus₀ program, which derives a single, persistent fact:

Example 5. An unsafe Dedalus₀ instance?

\[
\text{persist(p, p\_neg 2)}
\]
\[
p(1, 2)@123;
\]

The single ground fact will cause one deduction for each tuple in successor. Since successor is infinite, the corresponding Datalog program is unsafe.

However, observe that each of these deductions produces a tuple that changes only in its time suffix. We find it useful to distinguish between unsafe programs and programs that, given a finite EDB, eventually derive only tuples that are equivalent except in their time suffixes.

Definition 5. Two sets of ground atoms \( \Gamma \) and \( \Gamma' \) are equivalent modulo time if each atom \( \gamma \in \Gamma \) has a corresponding atom \( \gamma' \in \Gamma' \) such that \( \gamma \) and \( \gamma' \) have the same predicate symbol, and the same assignment of constants to attributes for all attributes except the time suffix.

Definition 6. We say a Dedalus₀ instance is quiescent at time \( T \) if the set of all atoms with time suffix \( T \) is equivalent modulo time to the set of all atoms with time suffix \( T - 1 \).

Observation 1. A Dedalus₀ instance that is quiescent at time \( T \) will be quiescent until timestamp of the next EDB fact \( V \), i.e., for all \( U \in \mathbb{Z} : V > U >= T \). If no EDB fact has a timestamp greater than \( T \), then the instance will be henceforth quiescent.

Proof. A Dedalus₀ program admits only instantaneous and inductive rules, which derive new tuples at the same time as their ground tuples, or in the immediate next timestep. Thus, the set of tuples true at time \( T \) is completely determined by any tuples true at time \( T - 1 \), and any EDB facts true at time \( T \). Observe that the integer value of the timestamp does not influence the derivation.

If the instance is quiescent at \( T \), then given \( A \), the set of atoms with timestamp \( T - 1 \), and the EDB at \( T \), the program entails \( A \) at timestamp \( T \). Thus in the absence of EDB facts at \( T + 1 \), it entails \( A \) at \( T + 1 \).

Definition 7. A Dedalus₀ instance with finite EDB is temporally safe if it is henceforth quiescent after some time \( T \).

Definition 8. Given the depends-on relation \( \sigma \) and its transitive closure \( \sigma^* \), an intensional predicate \( e \) in a program \( P \) is called an instantaneous predicate if for every predicate \( p \) for which \( e \sigma^* p \) (i.e., \( e \) depends transitively on \( p \)), either \( p \) appears in the head of no inductive rules, or the body of each inductive rule with head \( p \) contains at least one positive instantaneous predicate.

We propose the following conservative test for temporal safety. A program is guaranteed to be temporally safe if every rule is either:

1. An instantaneously safe rule, or
2. An inductive rule in which the head predicate occurs also in the body with the same variable bindings for all attributes save the time suffix, or
3. An inductive rule that has at least one instantaneous predicate as a positive subgoal in the body.

While a temporally safe program is henceforth quiescent after some time \( T \), a temporally unsafe program changes infinitely. Note that the Dedalus₀ program in Example 5 is temporally safe because \( r1 \) satisfies the second condition above.

Lemma 3. A temporally stratifiable Dedalus₀ instance is temporally safe if it has a finite EDB and every rule is one of the kinds 1-3 above.
Proof. Assume the program is temporally unsafe. That is, there exists no time \( T \) such that \( \forall U \geq T \), the set of all atoms with timestamp \( U \) is equivalent modulo time to the set of all atoms with timestamp \( T - 1 \). Let \( E \) be the maximum timestamp of any fact in the EDB.

Observe that every rule \( r \) of kind 3 may only entail a finite number of facts—as the EDB is finite—and thus may entail no facts at a timestamp greater than some maximum timestamp \( V_r \leq E + 1 \in \mathbb{Z} \). Since a \textsc{Dedalus} program has a finite set of rules we know \( \exists V \in \mathbb{Z} : \forall r : V_r = V \), and thus \( V \leq E + 1 \).

We now consider times \( T \) such that \( T > E + 1 \). By the above argument, no rules of kind 3 entail any facts with a timestamp greater than \( E + 1 \). Recall that no EDB atoms are true at any timestamp greater than \( E \). Thus, any facts with timestamp greater than \( E + 1 \) are entailed by rules of kind 1 or 2.

Consider the set of equivalence classes modulo time of all possible atoms, \( A \), given the Herbrand universe. We know \( A \) is finite, as the Herbrand Universe is finite. Therefore, if the program is temporally unsafe, then \( B \), the set of atoms entailed by the program, both contains and excludes infinitely many members of at least one equivalence class in \( A \) (i.e. something "infinitely oscillates in time" between being true and false). Since the program has finitely many rules, at least one rule must entail infinitely many atoms (from at least one of the equivalence classes from \( A \)). Thus, it is easy to see that there must be a cycle that contains some predicate \( P \) and \( \neg P \).

We show there exists such a cycle containing only rules of kind 1, which implies that the program is temporally unstratifiable. In order for such a cycle to exist, \( P \) must transitively depend on \( \neg P \), and \( \neg P \) must transitively depend on \( P \). Thus, the program contains a rule \( J_1 \) with \( \neg P \) in its body, and some predicate \( R \) in its head, as well as a rule \( J_2 \) that is transitively dependent on \( R \). With \( P \) in its head.

**Case 1:** \( P \neq R \). In this case, \( J_1 \) must be of kind 1, as for any \( Q \neq P \), a rule of kind 2 with \( P \) in the head may not directly entail \( Q \) given \( P \). \( J_2 \) must also be of kind 1—if it is of kind 2, then it necessarily contains \( P \) in its body, so it cannot entail \( P \) unless \( P \) is entailed by some other rule. If \( J_2 \) contains \( R \) in its body, then the program is syntactically unstratifiable. But if \( J_2 \) does not contain \( R \) in its body, then it contains some predicate \( S \) transitively entailed by \( R \); wlog the body contains \( R \). Thus, the program is syntactically unstratifiable.

**Case 2:** \( P = R \). In this case, \( J_1 \) and \( J_2 \) are the same rule: \( P \leftarrow \neg P \). Thus, the program is syntactically unstratifiable.

Thus, the program is temporally unstratifiable, which contradicts our assumption. □

**Example 6.** A \textsc{Dedalus} instance with a temporally unsafe deductive rule.

\[
p(A, B) \leftarrow q(A);
\]

The program above has a temporally unsafe deductive rule that corresponds to an unsafe rule in Datalog: it is not range-restricted. The head variable \( B \) could range over an infinite set of constants.

**Example 7.** A \textsc{Dedalus} instance that is temporally unsafe due to infinite oscillation.

\[
\text{flip_flop}(B, A)@\text{next} \leftarrow \text{flip_flop}(A, B);
\]

\[
\text{flip_flop}(\text{flip}(A, 1), 1)@\text{flip}(1);
\]

The above program exemplifies temporally unsafe induction. Even though it contains no function symbols, and all variables are range-restricted, it entails infinite oscillation of the \( p \) predicate.

We can imagine interesting examples of temporally unsafe programs, and do not forbid them in \textsc{Dedalus}.

5. **Evaluation**

In previous sections, we extended the notions of stratifiability and safety to \textsc{Dedalus} programs. In this section, we address the third and final property of \textsc{Dedalus} programs that we want to ensure—efficient execution.

Unfortunately, the direct application of traditional bottom-up Datalog execution strategies like semi-naive evaluation results in a rather literal and inefficient notion of the idea of "persistence." If a fact is true across a long sequence of timesteps, bottom-up evaluation will persistently "re-derive" that fact inductively for each timestep, and the number of derivations in a program will be infinite simply to maintain persistence in time. Instead, we would like an incremental evaluation strategy that allows an external agent to examine the state of the database at any timestep without requiring \( O(\tau) \) inductive derivations for persistence. The intuitive strategy would be to use a memory device, "storing" a fact on first derivation and "deleting" it at the timestep that the induction is broken. In this section we derive such a strategy via a combination of program rewriting and an operational evaluation loop, in the style of semi-naive evaluation.

5.1 Temporal Evaluation Over Storage

The traditional description of semi-naive evaluation takes a recursive Datalog program, rewrites it to a non-recursive "delta" program, and executes that program in a loop bracketed by state modifications. In that spirit, we present a strategy we call "temporal evaluation," which takes a \textsc{Dedalus} program, rewrites it to a Datalog program that refers to a single timestep, and executes that program in a loop—once per timestep—bracketed by state modifications. Algorithm 1 presents this strategy. Note that the \textsc{Dedalus} rules are written in their native "unsugared" syntax because we rewrite them into Datalog that strays from \textsc{Dedalus} conventions:

Observe that the final loop of Algorithm 1 binds the time suffix of each rule by replacing it with a constant value from a previous timestep. This "marches" through time in order, skipping steps that have no changes. A simple proof by induction shows that for each timestep \( t \), the temporal evaluation yields a database that corresponds to the minimal model of the original \textsc{Dedalus} program with the successor relation truncated to the prefix ending at \( t \).

6. **Ordering and Asynchrony**

Until now we have restricted our discussion to \textsc{Dedalus}. In this section we introduce \textsc{Dedalus}, a superset of \textsc{Dedalus} that also admits the \texttt{choice} construct [19] to bind time suffixes. Choice allows us to model the inherent nondeterminism in communication over unreliable networks that may delay, lose or reorder the results of logical deductions. We also describe a syntactic convention to employ this communication model for "horizontal partitions" of relations on different machines.

6.1 Choice

An important property of distributed systems is that individual computers cannot control or observe the temporal interleaving of their computations with other computers. One aspect of this uncertainty is captured in network delays: the arrival "time" of messages cannot be directly controlled by either sender or receiver. In this section, we enhance our language with a traditional model of nondeterminism from the literature to capture these issues: the \texttt{choice} construct as defined by Greco and Zaniolo [19].

The subgoal \( \texttt{choose}(X) \), \( \texttt{choose}(X) \) may appear in the body of a rule, where \( X_1 \) and \( X_2 \) are vectors whose constituent variables occur elsewhere in the body. Such a subgoal enforces the functional...
dependency $X_1 \rightarrow X_2$, “choosing” a single assignment of values to the variables in $X_2$ for each variable in $X_1$.

The choice construct is nondeterministic. In a model-theoretic interpretation of logic programming, a nondeterministic program must have a multiplicity of stable models—that is it must be unstratifiable. Greco and Zaniolo define choice precisely this fashion: the choice construct is expanded into an unstratifiable strongly-connected component of rules, and each possible choice is associated with a different model. Each such model has a unique, nondeterministic assignment that respects the given functional dependencies. In our discussion, it may be helpful to think of one such model chosen non-deterministically—a non-deterministic “assignment of timestamps to tuples”.

6.2 Distribution Model

The choice construct will capture the non-determinism of multiple communicating agents in a distributed system, but we want to use it in a stylized way to model typical notions of distribution. To this end Dedalus adopts the “horizontal partitioning” convention introduced by Loo, et al. and used in many subsequent efforts [30]. To represent a distributed system, we consider some number of agents, each running a copy of the same program against a disjoint subset (horizontal partition) of each predicate’s contents. We require one attribute in each predicate to be used to identify the partitioning for tuples in that predicate. We call such an attribute a location specifier, and prefix it with a # symbol in Dedalus.

Finally, we constrain Dedalus rules so that the location specifier variable in each body predicate be the same—i.e. the body contains tuples from exactly one partition of the database, logically colocated (on a single “machine”). If the head of the rule has the same location specifier variable as the body, we call the rule “local”, since its results can remain on the machine where they are computed. If the head has a different variable in its location specifier, we call the rule a communication rule. We now proceed to our model of the asynchrony of this communication, which is captured in a syntactic constraint on the heads of communication rules.

6.3 Asynchronous Rules

In order to represent the nondeterminism introduced by distribution, we admit a third type of rule, called an asynchronous rule. A rule is asynchronous if the relationship between the head time suffix $S$ and the body time suffix $T$ is unknown. Furthermore, $S$ (but not $T$) may take on the special value $\top$ which means “never.” Derivation at $T$ indicates that the deduction is “lost,” as time suffixes in rule bodies do not range over $T$.

We model network nondeterminism using the choice construct to choose from a value in the special time predicate, which is defined using the following Datalog rules:

```
\text{time}(\top); \
\text{time}(S) \leftarrow \text{successor}(S, \_); \
```

Each asynchronous rule with head predicate $p(A_1, \ldots, A_n)$ has the following additional subgoals in its body:

```
\text{time}(S), \text{choose}((A_1, \ldots, A_n, T)) \rightarrow (S)); \
```

where $S$ is the timestamp of the rule head. Note that our use of choose incorporates all variables of each head predicate tuple, which allows a unique choice of $S$ for each head tuple.

Example 8. A well-formed asynchronous Dedalus rule:

```
r(A, B, S) \leftarrow e(A, B, T), \text{time}(S), \text{choose}((A, B, T)), (S)); \
```

We admit a new temporal head annotation to sugar the rule above. The identifier async implies that the rule is asynchronous, and stands in for the additional body predicates. The above example expressed using async is:

```
Example 9. A sugared asynchronous Dedalus rule:
```

```
r(A, B)@async \leftarrow e(A, B); \
```

6.4 Asynchrony and Distribution in Dedalus

As a syntactic constraint of Dedalus, the communication rules introduced in the previous section (rules that differ in head and body location specifiers) are required to be asynchronous. This restricts our model of communication between agents in two important ways. First, by restricting bodies to a single agent, the only communication modeled in Dedalus occurs via communication rules. Second, because all communication rules are asynchronous, agents may only learn about time values at another agent by receiving messages (with unbounded delay) from that agent. Note that this model says nothing about the relationship between the agents’ clocks; they could be non-monotonically increasing, or they could respect a global order.

6.5 Temporal Monotonicity

Nothing in our definition of asynchronous rules prevents tuples in the head of a rule from having a timestamp that precedes the timestamp in the rule’s body. This is a significant departure from
DEDALUS, since it violates the monotonicity assumptions upon which we based both Algorithm 1 and our proof of temporal stratification. On an intuitive level, it may also trouble us that rules can derive head tuples that exist “before” the body tuples on which they are grounded; this violates intuitive notions of causality and admits the possibility of temporal paradoxes.

We have avoided restricting DEDALUS to rule out such issues, as doing so would reduce its expressiveness. Recall that simple monotonic Datalog (without negation) is insensitive to the values in any particular attribute. Hence DEDALUS programs without negation are also well-defined regardless of any “temporal ordering” of deductions: in monotonic programs, even if tuples with timestamps “in the future” are used to derive tuples “from the past”, there is an unambiguous least minimal model.

For non-monotonic DEDALUS, the monotonicity of the time suffix ensures us a unique minimal model in many cases. Whenever we can guarantee monotonicity of the time suffix for DEDALUS programs, our results from Section 4.1 still apply for all models produced by the choice construct.

6.5.1 Practical Implications

Given this discussion, in practice we are interested in three asynchronous scenarios: (a) monotonic programs (even with non-monotonicity in time), (b) non-monotonic programs whose semantics guarantee monotonicity of time suffixes and (c) non-monotonic programs where we have domain knowledge guaranteeing monotonicity of time suffixes. Each represents practical scenarios of interest.

The first category captures the spirit of many simple distributed implementations that are built atop unreliable asynchronous substrates. For example, in some Internet publishing applications (we blogs, online fora), it is possible due to caching or failure that a “thread” of discussion arrives out of order, with responses appearing before the comments they reference. In many cases a monotonic “bag semantics” for the comment program is considered a reasonable interface for readers, and the ability to tolerate temporal anomalies simplifies the challenge of scaling a system through distribution.

The second scenario is achieved in DEDALUS, via the use of successor for the time suffix. The asynchronous rules of DEDALUS require additional program logic to guarantee monotonically increases in time for predicates with dependencies. In the theoretical literature of distributed computing, this is known as a causal ordering, and is enforced by distributed clock protocols. We review one classic protocol in the DEDALUS context in Section 6.6: including this protocol into DEDALUS programs ensures temporal monotonicity.

Finally, certain computational substrates guarantee monotonicity in both timestamps and message ordering – for example, some multiprocessor cache coherence protocols achieve this. When temporal monotonicity is given, the proofs of temporal stratification and Algorithm [?] both apply.

6.5.2 Entanglement

Consider the asynchronous rule below:

\[ p(A, B, N) \leftarrow q(A, B) \cap N \]

Due to the async keyword in the rule head, each \( p \) tuple will take some unspecified time suffix value. Note however that the time suffix \( N \) of the rule body appears also in an attribute of \( p \) other than the time suffix, recording a binding of both the time value of the deduction and the time value of its consequence. We call such a binding an entanglement. Note that in order to write the rule it was necessary to not sugar away the time suffix in the rule body.

Entanglement is a powerful construct. It allows a rule to refer-
broadcast completes, all nodes that have not failed have received the message.

Our simple two-rule broadcast program is augmented with the following rules, so that if a node receives a message, it also multicasts it to every member before delivering the message locally:

\[
\text{smessage(Agent, Sender, Message)} \leftarrow \\
\text{rmessage(Agent, Sender, Message);}
\]

\[
\text{buf_bcast(Sender, Me, Message)} \leftarrow \\
\text{sdeliver(Me, Sender, Message);}
\]

\[
\text{smessage(Me, Sender, Message)} \leftarrow \\
\text{buf_bcast(Sender, Me, Message);}
\]

\[
\text{rdeliver(Me, Sender, Message)@next} \leftarrow \\
\text{buf_bcast(Sender, Me, Message);}
\]

Note that all network communication is initiated by the @async rule from the original simple broadcast. The @next is required in the rdeliver definition in order to prevent nodes from taking actions based upon the broadcast before it is guaranteed to meet the reliability guarantee.

Implementing other disciplines like FIFO and atomic broadcast and consensus are similar exercises, requiring the use of ordered queueing and sequences.

7. CASE STUDIES: SECURITY APPLICATIONS FOR DEDALUS

Concurrent to the development of declarative logic-based languages for distributed systems, the security community has developed declarative trust management systems [17, 6, 25, 21], designed to support multi-user access control in a distributed environment. The latter shares similarities with the former – both involve the notion of context (location) to reason about components (nodes) in a distributed system. Furthermore, adding integrity constraints to Datalog allows one to express a more general class of enforceable safety properties, such as properties necessary to write authentication and delegation constructs (e.g. “all facts must have valid signatures”) [33].

We explore further applications to security, in the areas of taint tracking and stateful firewalls. We demonstrate that conservative static taint tracking in a program is a transitive closure computation over the program, and that dynamic taint tracking is a simple program rewrite. Both can be expressed in our system, as DEDALUS represents the program as data, enabling introspection and rewriting.

Our stateful firewall example makes use of DEDALUS’s persistence and update capabilities. Since our language admits security constructs that can reason about change in data over time, it can express a broader class of security properties than previously expressible in logic-based security languages. This also suggests the intriguing possibility of vertical integration of security in a single framework, enabling cross-layer optimizations, such as performing a relatively simple access control check before an expensive firewall rule.

7.1 System Model

For the purposes of this discussion, we make a set of simplifying assumptions. Some will be essential to the viability of any of our approaches; others simplify our exposition and could be relaxed by pushing more constraints into the implementation. We present them in an order that reflects this:

1. Trusted Computing Base. We assume a trustworthy DEDALUS interpreter running on a trusted operating system running on trusted hardware. Thus we expect that certain safety invariants are enforced that we could not possibly enforce ourselves, such as the invariant that arbitrary memory addresses cannot be overwritten.

2. Static program. For the purposes of the examples below, we assume a program that does not itself change over time. We will later relax this assumption.

To ground our presentation in some intuition, consider a site middleware system implemented in DEDALUS, managing communication paths between web and XML services, databases and caches. End points are potentially vulnerable to SQL or command injection attacks, cross-site scripting attacks, or others. Other endpoints (e.g. web clients, untrusted services) may produce data that can exploit these vulnerabilities. Still others may attempt to compromise services running on other ports.

7.2 The Catalog

A longstanding tradition in database systems [10, 40] is storing metadata that describes tables in the database itself. The set of meta-relations describing the other relations in the system is known as the catalog; like any relation, the catalog relations can be read and manipulated by the query language. As in Condie et al. [15], we extend the catalog to include metadata describing the rules themselves as data. Thus program introspection and automatic program rewrites are queries over, and updates to, the catalog respectively. Our catalog is designed mainly to enable convenience in introspection and rewriting – the schema does not ensure the well-formedness of the program.

Figure 2 describes the subset of the DEDALUS catalog relevant to this discussion. Logical primary keys (the set of columns that should functionally determine the others) are underlined. Most of the columns have names that reflect their contents. The column ord reflects the (irrelevant to the interpretation but convenient for rewrites) position of a subgoal in a rule, while arg_ord reflects the position of an attribute in a predicate (i.e., the column ordinal). Only one of relname or text is non-NULL for a given tuple in subgoal; if the latter, the tuple contains an expression that defines or restricts a body variable. Note that one may choose to add views to ease the process of querying the catalog:

\[
\text{rule_head(Rule, Relname)} \leftarrow \\
\text{subgoal(Rule, Ord, Relname, \text{...}, \text{...}),} \\
\text{Ord = \emptyset;}
\]

To maintain the integrity of the catalog, we also introduce some constraints using Ullman’s panic construct [20]:

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule</td>
<td>relname, arity</td>
</tr>
<tr>
<td>relation</td>
<td>relname, arity</td>
</tr>
<tr>
<td>subgoal</td>
<td>relname, ord, relname, expr</td>
</tr>
<tr>
<td>rule_arg</td>
<td>relname, ord, arg_ord, text, is_constant</td>
</tr>
</tbody>
</table>

Figure 2: The Catalog
subgoal_cnt(Rule, count<Ord>) ← subgoal(Rule, Ord, _, _, _);

predarg_cnt(Relation, max<Arg_ord>) ← subgoal(Rule, Ord, Relation, _), rule_arg(Rule, Ord, Arg_ord, _, _);

panic() ← rule(Rule, Subgoals), subgoal_cnt(Rule, Cnt), Subgoals != Cnt;

panic() ← relation(Relation, Attrs), predarg_cnt(Relation, MaxArgs), Attrs != MaxArgs;

These rules, and others, ensure that the catalog is not semantically inconsistent; the 0-ary predicate panic represents an exception. These rules can be read as database integrity constraints: rules must have the same number of subgoals as their arity, relations must appear as predicates in rules with the same arity as the recorded relation. Note that one may write rules that contain panic in their body to resolve exceptions in the next timestamp. If the exception is unresolved, panic will be re-derived.

We now demonstrate how Dedalus can enable rewrites. Consider a case in which we want to add a global constraint to a program, and “push down” the predicate as far as it can go to reduce computation.

rule(Rule, Subgoals)+1@next ← rule(Rule, Subgoals), do_rewrite();

subgoal(Rule, Subgoals+1, NULL, Text + " > 10")@next ← rule(Rule,Subgoals),

subgoal(Rule, Ord, Relation, _),

rule_arg(Rule, Ord, Arg_ord, Text, _),

Relation = "foobar", Arg_ord = 2,

do_rewrite();

The example above rewrites a program in such a way that every rule with a subgoal referencing the relation “foobar” is rewritten to include a clause that restricts the value bound to the second argument of foobar to be less than 10. That is to say, the second two rules in the program

p(A, B) ← q(A, B), B < 10;
p(A, B) ← r(A, B), B < 10;
q(A, B) ← foobar(A); r(A, B) ← foobar(A, B), t(B, C);

would be rewritten to:

q(A, B) ← foobar(A, B), B < 10;
r(A, B) ← foobar(A, B), t(B, C), C < 10;

7.3 Taint Analysis

Taint analysis has been proposed as a viable defense against attacks that exploit dataflows that mix data and control paths, including SQL injection, cross-site scripting and shell command injection [1, 12, 43]. Retrofitting taint propagation infrastructure to imperative languages is attractive for supporting legacy applications, but is notoriously difficult because of the subtle ways in which information can flow through an imperative program. Language features like aliasing, branching via conditionals and array lookups obfuscate the dataflow and potentially provide an implicit information flow through which tainted data can bypass detection and change program behavior. In all such systems, there is an obvious tradeoff between soundness and completeness; some [43] provide a basic policy that guarantees neither, while others [12] begin with a complete system and work to reduce the false positive rate.

Dedalus maps directly to a dataflow representation, and its syntax reflects this, making taint tracking straightforward. We observe that the taint propagation problem is a restricted case of the provenance problem [9] in database systems: from whence did a given piece of data come? In a logic-based language like Dedalus in which implication is the basic primitive, all information flow is explicit: data propagates in the direction of the implication “arrow,” from body to head, making it easy to reason about data provenance and dependencies.

We want our approach to remain as general as possible, so we describe only taint propagation in detail, leaving source tainting and sink checking to individual applications. Source tainting is a matter of policy: certain external inputs (conservatively, all of them) are marked as suspect. The static approach described below also differs from the dynamic approach in that source tainting is performed at the predicate rather than at the tuple level. Implementing this tuple-level marking will require a custom queueing discipline, similar to those described in Section 3.4. Sink checking is likely to be endpoint-specific.

7.3.1 Static Analysis

depends(Head, Body) ←

rule_head(Rule, Head),

subgoal(Rule, Ord, Relname, _, _),

Ord > 0;

depends(Head, Body) ←

depends(Head, Intermediate),

depends(Intermediate, Body);

tainted(Sink) ←
taint(Source), depends(Sink, Source);

It is easy to see that this dependency analysis-based taint detection mechanism is complete, but it is far from sound because it is both coarse-grained and conservative. Consider a predicate side_effect which, when inserted into, launches a missile (or equivalently, prints a document), and a predicate dirty_source that may contain data generated by an untrusted source. If the query tainted(side_effect) ? has a true result, then one or more of the columns in side_effect may have been (perhaps transitively or indirectly) produced by dirty_source – but this is all we know. Converting this metaprogram to a finer-grained equivalent that tracks taint statically at the column level is left as an exercise for the reader.

7.3.2 Dynamic Analysis

Consider the program below:

r1 response(A, B) ← timeout(A,B);

r2 response(A, B) ← request(A, C), data_source(C, D);

timeout(1, 2);
data_source(4, 5);
tainted(response)?;

Even a column-wise static analysis of the above program would incorrectly determine that p is tainted. We know, however, that because the relation r is empty, no tuples in p could have been derived from s. If we add to the program a further fact t(5, 6), then we will have two tuples in p, one of which comes from a tainted derivation. It would be useful to distinguish these cases – the problem is analo-
gous to the problem of tracking taint through conditional branches in imperative languages.

What is required is a limited notion of provenance: a tuple should carry with it an attribute describing the derivation of the tuple from ground facts. We call this attribute the taint attribute. Such derivation information will capture the “branches” actually taken in the dataflow. The addition of this attribute to all relations and modification of all rules to propagate the derivation information from body to head of each rule can be captures as a rewrite; i.e., a query over the catalog that updates the catalog. Ideally, this attribute should be succinct, as propagating provenance information may be expensive.

```prolog
relation(Rename, Arity+1)@next ←
  rule(Rename, Arity), do_rewrite();
rule(Rule, Subgoals+1)@next ←
  rule(Rule, Subgoals), do_rewrite();
// add the last argument, P-n, to each predicate.
rule_arg(Rule, Ord, Arity + 1, "P", Ord, false) ←
  subgoal(Rule, Ord, Rename, _, _),
  relation(Rename, Arity), do_rewrite();
// add the last subgoal to each rule.
subgoal(Rule, Arity+1, NULL,
  "P0=\"r1(\"+\"P1\"+\")\")@next ←
  rule(Rule, Arity), do_rewrite(),
  subgoal(Rule, Ord, _, _, _);
```

Here, mklst is an aggregate function that, in this simple example, takes a set as input and returns a string in which the elements of the set are joined with the string “+”. For readability in these examples we use string concatenation to capture tuple provenance as a string in the recursive form `rule_arg(subgoal, [...]`, where `subgoal` is defined in the same way. The terminals are the taint status ‘clean’ or ‘tainted’. For example, a tuple derivation might read “r2(r1(clean), r3(tainted))”, indicating that the tuple was derived from rule r2, via tuples that were derived via r1 and r3, the latter of which was tainted. This is a toy example intended to substitute for an object representing a provenance tree and acted upon by appropriate methods.

The rewritten program is:

```prolog
r1
  response(A, B, P0) ←
  timeout(A,B, P1), P0 = \"r1(\"+\"P1\"+\")\";
```

```prolog
r2
  response(A, B, P0) ←
  request(A, C, P1), data_source(C, D, P2),
  P0 = \"r1(\"+\"P1\"+\")\";
  timeout(1, 2, \"clean\");
  data_source(4, 5, \"tainted\");
  data_source(9, 10, \"clean\");
```

These dynamic checks will lower the false positive rate by determining taint status based on derivation history rather than predicate dependencies. Clearly, however, they are still not sound. Specifically, a derived (head) atom is considered tainted if any of the atoms participating in the derivation was tainted, even if the tainted columns of that atom are never projected into the head. We can consider rewrites that carry more information into the provenance datastructure, to test for the case of tainted data being “projected out,” but we must be very careful that by performing this optimization we have not compromised our completeness property. Weakening our assumptions in any way allows for implicit information flows from rule body to head.

A question arises: what if we obtain after a series of derivations two tuples that are identical modulo their provenance string? This is similar to the question posed by Chin and Wagner [12] about equality of tainted strings. Does such a derivation capture a false positive or a false negative? In general there is no correct answer, so we are forced to leave it as a matter of policy, also expressible via program rewrite.

### 7.3.3 Discussion

In logic, we say (informally) that a system of reasoning is sound if it is impossible to prove a false proposition in that system. We say that a system is complete if all true propositions can be proven in the system.

The taint tracking literature borrows this terminology and its informal meaning. In such a system, all propositions are in the form “such-and-such data is tainted.” Hence, a sound taint tracking system is one in which it is impossible to prove “X is tainted” unless X truly is tainted; i.e., the system has no false positives. Conversely, a complete taint tracking system is one in which, if X is indeed tainted, “X is tainted” is always provable; i.e., no false negatives.

Taint tracking is a security feature, and as such focuses on worst-case behavior. Implementing a tracking system that was not complete would be foolhardy, as by definition such a system could be compromised. On the other hand, systems that are not sound produce warnings even when the program is correct, and train programmers to ignore warnings. Clearly we should require that systems be complete, but a trivially complete system that marks all data as suspect is of little use.

Like previous systems, we propose a complete but conservative system. Our approach leverage both static and dynamic program verification techniques by casting both as meta-queries against a catalog describing the program and its data structures. Beginning from a complete but unsound base, our approach allows for iterative refinement of the taint-propagation logic via program rewrites to reduce false positives. Inexpensive static checks can be run at compile time: any predicates marked tainted by the static checks could be tainted in an execution. We may then consider only this set for dynamic instrumentation: only rules in which predicates in this set appear are rewritten, as are the table definitions of predicates occurring in those rules (as shown above).

Static taint tracking is straightforward in any database language; in part, it is something that falls out naturally from an explicit schema and meta-relations. DEDALUS drastically simplifies the effort of expressing queries that update the catalog relations that they read, enabling powerful rewrite capabilities, from DEDALUS to DEDALUS in DEDALUS. Because rules are data and DEDALUS allows stored data to evolve over time with straightforward declarative semantics, allowing a program itself to evolve over time is natural. Inductive rules provide a strong atomicity guarantee: we may express an update as a program that alters the catalog in such a way that all of the rewrite’s effects are seen together in the next timestep. Again, we need to be careful that our taint tracking meta-rewrites are not subject to race conditions on the catalog.

### 7.3.4 Proofs of completeness

It is easy to see that the static method described above is complete if all the sources are correctly marked as tainted. The set of tainted predicates is precisely the set of predicates that are transitively dependent on tainted sources. Thus, every output that could hold tainted data is in the set of tainted predicate.

It is also easy to see that the dynamic method described above is complete, if every tainted base fact is marked as such in its taint
argument. This is because taint information from all body atoms is always copied into the taint argument of the head atom in any derivation. Thus, every output atom that may have been influenced by tainted data contains the taint string, e.g. “tainted,” in its taint argument.

7.4 Stateful firewalls

While previous work has explored representing access control and trust management in logic-based languages, a particular difficulty that precludes compact and intuitive encoding of more general security constructs is the inability of traditional logic-based languages to express state update.

7.4.1 Threat Model

As before, we assume that the Dedalus interpreter is trusted, executing on a trusted operating system with trusted hardware. Users attempt to send packets that evade the firewall rules.

7.4.2 Stateful Rules

Consider an intrusion detection system rule that blocks a host whenever it probes two consecutive ports:

```prolog
//persist ultimate port
host_ultimate_port@next(Src, Port) :-
  packet(Src, Port, _, _).
//persist penultimate port
host_penultimate_port@next(Src, Port) :-
  host_last_port(Src, Port).
block_host(Src) :-
  host_ultimate_port(Src, Port),
  host_penultimate_port(Src, Port+1).
block_host(Src) :-
  host_ultimate_port(Src, Port),
  host_penultimate_port(Src, Port-1).
```

Expressing this rule in earlier logic-based languages requires the use of an update construct that adds to the semantics of Datalog. Given the ambiguities with this construct, gaining confidence in the code would require testing to ensure the code behaved as expected. Our clear and unambiguous semantics model updateable state with ease.

It is easy to see that this code will block any host that scans two consecutive ports: the `host_ultimate_port` and `host_penultimate_port` relations will always contain the ultimate and penultimate ports to which a given host sent packets. `block_host` will be derived whenever the ultimate and penultimate ports are consecutive integers.

7.5 Future Work

One of the design goals of Dedalus was to enable more natural expression and verification of safety and liveness properties of distributed systems. Informally, a safety property specifies that certain bad things never happen, while a liveness property states that certain good things must eventually happen.

In our case study, we focused on expressing certain security concepts as safety properties: invariants that must hold over the data store at all timesteps, such as the assertion that tainted data cannot be delivered to a sink, or that a packet not explicitly allowed is denied. We conjecture that many security properties that are equally difficult to achieve, such as robustness to (distributed and otherwise) denial of service attacks, can be encoded as liveness properties. We are confident that with its explicit treatment of time Dedalus will enable more natural expression of liveness properties.

8. RELATED WORK

8.1 Updateable State

Many deductive database systems, including LDL [11] and Glue-Nail [16], admit procedural semantics to deal with updates using an assignment primitive. In contrast, languages proposed by Cleary and Liu [14, 27, 31] retain a purely logical interpretation by admitting temporal extensions into their syntax and interpreting assignment or update as a composite operation across timesteps [27] rather than as a primitive. We follow the latter approach, but differ in several significant ways. First, we model persistence explicitly in our language, so that like updates, it is specified as a composite operation across timesteps. Partly as a result of this, we are able to enforce stricter constraints on the allowable time suffixes in rules: a program may only specify what deductions are visible in the current timestep, the immediate next timestep, and some future timestep, as opposed to the free use of intervals allowed in rules in Liu et al. Our simple inductive approach to persistence obviates the need to evaluate stabbing queries on time “ranges.”

U-Datalog [8] addresses updates using syntax annotations that establish different interpretations for the set of updated relations and the IDB, interpreting update atoms as constraints and using constraint logic programming techniques to test for inconsistent derivations. Similarly, Timed Concurrent Constraint Programming (TCCP) [38, 39] handles nonmonotonic constructs in a CLP framework by outputting a new (possibly diminished) store and constraint program at each timestep.

Lamport’s TLA+ [24] is a language for specifying concurrent systems in terms of constraints over valuations of state, and temporal logic that describes admissible transitions. The notion of state predicates being distinguishable from actions in that they are “invariant under stuttering” is similar to our declarative definition of table persistence. Two distinguishing features of Dedalus with respect to TLA+ is our minimalist use of temporal constructs (next and later), and our unified treatment of temporal and other attributes of facts, enabling the full literature of Datalog to be applied both to temporal and instantaneous properties of programs.

8.2 Distributed Systems

Significant recent work ([3, 7, 13, 29], etc.) has focused on applying deductive database languages extended with networking primitives to the problem of specifying and implementing network protocols and distributed systems. Implementing distributed systems entails a data store that changes over time, so any useful implementation of such a language addresses the updateable state issue in some manner. Existing distributed deductive languages like NDlog and Overlog adopt a chain of fixpoints interpretation. All rules are expressed as straightforward Datalog, and evaluation proceeds in three phases:

1. Input from the external world, including network messages, clock interrupts and host language calls, is collected.
2. Time is frozen, the union of the local store and the batch of events is taken as EDB, and the program is run to fixpoint.
3. The deductions which cause side effects, including messages, writes and deletions of values in the local store, and host language callbacks are dealt with.

Unfortunately, the language descriptions give no careful specification of how and when deletions and updates should be made visible, so the third step is a “black box.” Loo et al. [28] proved that classes of programs with certain monotonicity properties (i.e.
programs without negation or fact deletion) are equivalent (specifically, eventually consistent) when evaluated globally (via a single fixpoint computation) or in a distributed setting in which the chain of fixpoints interpretation is applied at each participating node, and no messages are lost. Navarro et al. [36] proposed an alternate syntax that addressed key ambiguities in Overlog, including the event creation vs. effect ambiguity. Their solution solves the problem by introducing procedural semantics to the interpretation of the augmented Overlog programs. A similar analysis was offered by Mao [32].

9. CONCLUSION

Datalog has inspired a variety of recent applied work, which touts the benefits of declarative specifications for practical implementations. We have developed substantial experience building significant distributed systems [3, 4, 13, 29] using hybrid declarative/imperative languages such as Overlog [29]. While our experience with those languages was largely positive, the combination of Datalog and imperative constructs often clouded our understanding of the “correct” execution of single-node programs that performed state updates. This work developed in large part as a reaction to the semantic difficulties presented by these distributed logic languages.

Through its reification of time as data, Dedalus allowed us to achieve the goal of a declarative language without sacrificing critically expressive features for the distributed systems domain. We believe that Dedalus is as expressive as Overlog, whose operational semantics [3] are essentially the same as those described in Algorithm 1. Formalizing this intuition is difficult because the semantics of Overlog are not well specified. Instead, we are currently validating our practicality by “porting” many of our Overlog programs to Dedalus.

In Dedalus, state update and communication differ from logical deductions only in terms of timing. In the local case, this allows us to express state update without giving up the clean semantics of Datalog; unlike Datalog extensions that use imperative constructs to provide such functionality, each Dedalus rule expresses a logical invariant that will hold over all program executions. However, interactions with external processes, and primitives such as asynchronous and unreliable communication introduce nondeterminism which Dedalus models with choose. Our hope is that modeling external process and events with a single primitive will simplify formal program verification techniques for the distributed systems domain. Two natural directions in this vein are to determine for a given Dedalus program whether Church-Rosser confluence holds for all models produced by choose, and to capture finer-grained notions like serializability of such models with respect to transaction identifiers embedded in EDB facts.

10. REFERENCES

11. BREAKDOWN OF EFFORT

Dedalus is conceptually my invention, but I couldn’t have pulled off the paper without all of the other authors. Bill was a solid second author, and was there for some of the early brainstorming sessions with Joe and I. Joe clearly played a large part in everything, from articulating the problem to writing the introduction, and everything in between. Neil was involved in many of the foundational conversations and helped to edit the draft. Rusty got involved late but was a big contributor, particularly to the later sections of the paper. David Maier attended one of the original brainstorming sessions and provided copious feedback on the various drafts.

I “discovered” the formalism (though it evolved as such things do as we wrote, and everyone contributed to the final formalism), the notions of temporal safety and temporal stratifiability, and the various design patterns for state update and asynchrony, as well as penning the initial and approving the “final” draft. Bill wrote all of the proofs, although in some cases he started with a proof sketch that I provided. The Evaluation section was entirely written by Bill and Joe (but I like it very much). Only Bill and myself worked on the case study. I wrote the taint tracking, system model and catalog sections of the security case study, while Bill wrote the firewall section and the case study introduction.

Peter Alvaro