Positive Dedalus Programs Tolerate Non-Causality☆

Tom J. Ameloot[a,1], Jan Van den Bussche[a]

[a]*Hasselt University & Transnational University of Limburg, Diepenbeek, Belgium*

Abstract

Declarative networking is a recent approach to programming distributed applications with languages inspired by Datalog. A recent conjecture posits that the delivery of messages should respect causality if and only if they are used in non-monotone derivations. We present our results about this conjecture in the context of Dedalus, a Datalog-variant for distributed programming. We show that both directions of the conjecture fail under a strong semantical interpretation. But on a more syntactical level, we show that positive Dedalus programs can tolerate non-causal messages, in the sense that they compute the correct answer even when messages can be sent into the past.

Keywords: declarative networking, causality, asynchronous communication

1. Introduction

In declarative networking, distributed computations and networking protocols are modeled and programmed using formalisms based on Datalog [2]. Hellerstein has made a number of intriguing conjectures concerning the expressiveness of declarative networking [3]. In the present paper, we are focusing on the CRON conjecture (Causality Required Only for Non-monotonicity).

Causality stands for the physical constraint that an effect can only happen after its cause. Applied to message delivery, this intuitively means that a sent message can only be delivered in the future, not in the past. Now, the conjecture relates the causal delivery of messages to the nature of the computations that those messages participate in, like monotone versus non-monotone, and asks us to think about the cases where causality is really needed.

There seem to be interesting real-world motivations for studying the CRON conjecture, one of which is crash recovery. Distributed computations happen often in large clusters of compute nodes, where failure of nodes is not uncommon [4], and indeed distributed computing software should be robust against failures [5]. Consider the following situation. During crash recovery, a program can read an old checkpointed state and a log of received messages, which is disjoint from that state. At the beginning of a computation, suppose a node x at its local time 0 sends a message A to another node y, and locally sets a flag "waiting". In reply to A, node y sends a message B to x. Upon receiving B, node x sets an output flag and B is added to the message log at x, timestamped with the arrival time. Next, x crashes. There were no previous checkpoints of the state at x, so the recovery procedure has to revert x back to its initial state. But the recovery procedure also has access to the logged message B. However, in some sense, B appears to come from the "future" when put side-by-side with the initial state because according to this state, message A has not yet been sent, i.e., the flag "waiting" is not set. It really depends on the program at x how we should use the message log during recovery. For example, it could be that if we read B during recovery, the program at x will immediately produce the output even though the flag "waiting" is absent. But it could also be that the program at x will block the output if we read B in the absence of the flag "waiting". So, in general, it

☆This paper is the extended version of our conference paper [1].

Email addresses: tom.ameloot@uhasselt.be (Tom J. Ameloot), jan.vandenbussche@uhasselt.be (Jan Van den Bussche)

[1]PhD. Fellow of the Fund for Scientific Research, Flanders (FWO)

requires a fundamental understanding of the program in order to design a good crash recovery mechanism, certainly if negation and more generally non-monotone operations are involved. To aid in this task, it seems useful to attempt a classification of programs and to provide advice for each class.

Indeed, one can understand the CRON conjecture as saying that during recovery, for a non-monotone program, messages from the log should be read in causal order, like the order in which they are received, and they should not be exposed all at once.[2] From the other direction, if you know that the program is monotone, the recovery could perhaps become more efficient by reading the messages all at once.

In this paper we formally investigate the CRON conjecture in the setting of the language Dedalus, which is a Datalog-variant for distributed programming [6, 7, 3, 8, 9]. It turns out that stable models [10] provide a way to reason about non-causality, and we use this to formalize the CRON conjecture. A strong interpretation of the conjecture posits that causality is not needed if and only if the query computed by a Dedalus program is monotone. Neither the "if" nor the "only if" direction holds, however, as we will demonstrate. Therefore we have turned attention to a more syntactic version of the conjecture, and there we indeed find that causal message ordering is not needed for positive Dedalus programs in order to compute meaningful results, if these programs already behave correctly in a causal operational semantics. This is the main result of our paper.

This paper is organized as follows. First, Section 2 relates our work to the literature. Section 3 gives preliminaries on databases, Datalog, and Dedalus. Next, Section 4 investigates the expressivity and complexity of Dedalus. Section 5 states the CRON conjecture and gives the formalization of non-causality. Section 6 contains the results. We conclude in Section 7.

Acknowledgments. We thank Joseph M. Hellerstein for his thoughtful comments on an earlier draft of this paper. We also thank the anonymous reviewers for their constructive comments and for bringing interesting related work to our attention.

2. Related Work

Essentially, Dedalus is a logic programming language to describe *events* that should take place in a distributed system. Other languages have been proposed for this setting, with a flavor similar to Dedalus. For instance, Lobo et al. [11] describe a rule-based language for distributed systems inspired by Dedalus. They give a model-theoretic semantics based on answer set programming, i.e., stable models. To define this semantics, they syntactically translate the rules of their language to Datalog, where all literals are given an explicit location and time variable, to represent the data that each node has during each local time. This translation resembles the model-theoretic semantics for Dedalus [8, 9], that we will also recall in this paper. To enforce natural execution properties in their semantics, like causality, Lobo et al. specify auxiliary rules in the syntactical translation. Interestingly, our paper studies the effect of *omitting* such auxiliary rules, to see how the behavior of the program changes as a result.

Also in the setting of distributed systems, Interlandi et al. [12] give a Dedalus-inspired language for describing *synchronous* systems. In such systems, the nodes of the network proceed in rounds and the messages can not be arbitrarily delayed. During each round, the nodes share the same global clock. Interlandi et al. specify an operational semantics for their language, based on relational transducer networks [13]. They also show that this operational semantics coincides with a model-theoretic semantics of a single holistic Datalog program. It should be noted that Lobo et al. [11], and the current paper, deal with *asynchronous* systems, that in general pose a bigger challenge for a distributed program to be correct, i.e., the program should remain unaffected by nondeterministic effects caused by message delays.

For describing the semantics, Interlandi et al. [12] use that negation is *temporally stratified*: intuitively, this means that negation is applied to relations computed in the past, and thus there are no cyclic dependencies involving negation through time. This way, programs can be given an intuitive semantics. Temporal

[2] In particular, a logged message would only become visible when the recovering node has again reached the timestamp at which the message had originally arrived. Additionally, recovery can be performed as an atomic operation, during which no new messages are received.

stratification is a well-studied concept in temporal deductive databases and temporal logic programming. For example, Zaniolo et al. [14] define XY-stratification for Datalog programs, which is a form of temporal stratification. A similar approach is also given by Lausen et al. [15] and Lu and Cleary [16], where the former call the stratification condition *state stratification*. All these languages extend Datalog with explicit time variables in the head and body of rules: these variables tag facts with a time instant, to say when the facts exist. Seminal work on temporal deductive databases was done by Chomicki and Imieliński [17, 18]. However, these previous works do not model asynchronous communication, as is done with Dedalus. Yet Dedalus without rules for asynchronous communication can probably be (strictly) embedded in some of these languages, because the remaining rules only reason about the current time and the *next* time, which is supported by the above languages, some of which even allow reasoning with larger jumps in time [15, 16].

Languages have also been proposed to reason about events in planning problems. For instance, Eiter et al [19] propose a language based on logic programming. Concretely, their language can describe general systems, and the idea is to generate sequences of *actions* in order to obtain a goal list of ground literals. Time is only implicitly present, although in principle it could be deduced from the number of actions taken so far. Effects of actions always occur in the future. In the same spirit, Greco et al. [20] give a Datalog-inspired language to represent actions and events. However, their language explicitly represents time, to allow actions to trigger events in the future, after a certain delay. Moreover, the language of Greco et al. can represent time at multiple granularities in the same program, to model subtasks of a larger task.

All related work mentioned above contains languages for reasoning about events that are always triggered in the future. By contrast, in this paper we do an initial attempt to find classes of programs for reasoning about events that can be executed without this assumption. Although we have focused on the language Dedalus to have a better connection to the CRON conjecture, we expect that the main insights are transferable to other Datalog inspired languages, like the ones mentioned above, if they are executed under a semantics without causality.

Still from the field of temporal deductive databases, we should also mention the work of Nomikos et al. [21]: they study extensions of Datalog to express linear-time and branching-time problems. For these languages, Nomikos et al. design tests to verify if programs are temporally stratified. Although it is not immediately clear if the asynchronous communication of Dedalus could be modeled in branching-time Datalog, the fragment of Dedalus without asynchronous communication can probably be represented in the languages of Nomikos et al. But since a Dedalus program always derives facts from the past to the future in its standard (causal) semantics, even if it uses asynchronous communication, the stratification test would seem unnecessary. Nevertheless, if we execute Dedalus in a semantics not respecting causality, it seems that programs are in general temporally unstratifiable. Then, designing a test for temporal stratification in such cases appears similar in spirit to this paper: finding classes of programs that have meaning when executed under a non-causal semantics.

3. Preliminaries

3.1. Database Basics

A *database schema* $\mathcal{D}$ is a finite set of pairs (R, k) where R is a *relation name* and $k \in \mathbb{N}$ its associated *arity*. A relation name occurs at most once in a database schema. We often write (R, k) as $R^{(k)}$.

We assume some infinite universe **dom** of atomic data values. A *fact* $\boldsymbol{f}$ is a pair $(R, \bar{a})$, often denoted as $R(\bar{a})$, where R is a relation name and $\bar{a}$ is a tuple of values over **dom**. For a fact $R(\bar{a})$, we call R the *predicate*. We say that a fact $R(a_1, \ldots, a_k)$ is *over* database schema $\mathcal{D}$ if $R^{(k)} \in \mathcal{D}$. A database *instance* I over $\mathcal{D}$ is a set of facts over $\mathcal{D}$. For a subset $\mathcal{D}' \subseteq \mathcal{D}$, we write $I|_{\mathcal{D}'}$ to denote the subset of facts in I whose predicate is a relation name in $\mathcal{D}'$. We write $adom(I)$ to denote the set of values occurring in facts of I.

A *query* $\mathcal{Q}$ is a function from database instances over an input schema $\mathcal{D}_1$ to database instances over an output schema $\mathcal{D}_2$. Query $\mathcal{Q}$ is called *monotone* if for any two instances I and J over $\mathcal{D}_1$, $I \subseteq J$ implies $\mathcal{Q}(I) \subseteq \mathcal{Q}(J)$. We recall that first-order logic (FO) can be used to express a class of database queries [22].

3.2. Datalog with Negation

We recall Datalog with negation [22], abbreviated Datalog$^\neg$. We assume the standard database perspective, where a Datalog$^\neg$ program is evaluated over a given set of facts, i.e., where these facts are not part of the program itself.

Let **var** be a universe of *variables*, disjoint from **dom**. An *atom* is of the form $R(u_1, \ldots, u_k)$ where R is a relation name and $u_i \in \mathbf{var} \cup \mathbf{dom}$ for $i = 1, \ldots, k$. We call R the *predicate*. If an atom contains no data values, we call it *constant-free*. A *literal* is an atom or an atom with "¬" prepended. A literal that is an atom is called *positive* and otherwise it is called *negative*.

A Datalog$^\neg$ *rule* φ is a triple

$$(head_\varphi, pos_\varphi, neg_\varphi)$$

where $head_\varphi$ is an atom, and pos_φ and neg_φ are sets of atoms. The components $head_\varphi$, pos_φ and neg_φ are called respectively the *head*, the *positive body atoms* and the *negative body atoms*. We refer to $pos_\varphi \cup neg_\varphi$ as the *body atoms*. Note, neg_φ contains just atoms, not negative literals. Every Datalog$^\neg$ rule φ must have a head, whereas pos_φ and neg_φ may be empty. If $neg_\varphi = \emptyset$ then φ is called *positive*.

A rule φ may be written in the conventional syntax. For instance, if $head_\varphi = T(\mathtt{u}, \mathtt{v})$, $pos_\varphi = \{R(\mathtt{u}, \mathtt{v})\}$ and $neg_\varphi = \{S(\mathtt{v})\}$, with $\mathtt{u}, \mathtt{v} \in \mathbf{var}$, then we can write φ as

$$T(\mathtt{u}, \mathtt{v}) \leftarrow R(\mathtt{u}, \mathtt{v}),\ \neg S(\mathtt{v}).$$

The specific ordering of literals to the right of the arrow is arbitrary.

The set of variables of φ is denoted $vars(\varphi)$. We call φ *safe* if the variables in φ all occur in pos_φ. If $vars(\varphi) = \emptyset$ then φ is called *ground*, in which case $\{head_\varphi\} \cup pos_\varphi \cup neg_\varphi$ is a set of facts.

Let $\mathcal{D}$ be a database schema. A rule φ is said to be *over schema* $\mathcal{D}$ if for each atom $R(u_1, \ldots, u_k) \in \{head_\varphi\} \cup pos_\varphi \cup neg_\varphi$ we have $R^{(k)} \in \mathcal{D}$. A Datalog$^\neg$ *program* P *over* $\mathcal{D}$ is a set of safe Datalog$^\neg$ rules over $\mathcal{D}$. We write $sch(P)$ to denote the database schema that P is over. We define $idb(P) \subseteq sch(P)$ to be the database schema consisting of all relations in rule-heads of P. We abbreviate $edb(P) = sch(P) \setminus idb(P)$.[3]

Any database instance I over $sch(P)$ can be given as *input* to P. Note, I may already contain facts over $idb(P)$. The need for this will become clear in Section 3.5. Let $\varphi \in P$. A *valuation for* φ is a total function $V : vars(\varphi) \to \mathbf{dom}$. The *application* of V to an atom $R(u_1, \ldots, u_k)$ of φ, denoted $V(R(u_1, \ldots, u_k))$, results in the *fact* $R(a_1, \ldots, a_k)$ where for each $i \in \{1, \ldots, k\}$ we have $a_i = V(u_i)$ if $u_i \in \mathbf{var}$ and $a_i = u_i$ otherwise. In words: applying V replaces the variables by data values and leaves the old data values unchanged. This is naturally extended to a set of atoms, which results in a set of facts. Valuation V is said to be *satisfying for* φ *on* I if $V(pos_\varphi) \subseteq I$ and $V(neg_\varphi) \cap I = \emptyset$. If so, φ is said to *derive* the fact $V(head_\varphi)$.

3.2.1. Positive and Semi-positive

Let P be a Datalog$^\neg$ program. We say that P is *positive* if all rules of P are positive. We say that P is *semi-positive* if for each rule $\varphi \in P$, the atoms of neg_φ are over $edb(P)$. Note, positive programs are semi-positive.

We now give the semantics of a semi-positive Datalog$^\neg$ program P [22]. First, let T_P be the *immediate consequence operator* that maps each instance J over $sch(P)$ to the instance $J' = J \cup A$ where A is the set of facts derived by all possible satisfying valuations for the rules of P on J.

Let I be an instance over $sch(P)$. Consider the infinite sequence I_0, I_1, I_2, etc, inductively defined as follows: $I_0 = I$ and $I_i = T_P(I_{i-1})$ for each $i \geq 1$. The *output of* P *on input* I, denoted $P(I)$, is defined as $\bigcup_j I_j$; this is the *minimal fixpoint* of the T_P operator. Note, $I \subseteq P(I)$. When I is finite, the fixpoint is finite and can be computed in polynomial time (according to data complexity [23]).

[3]The abbreviation "idb" stands for "intensional database schema" and "edb" stands for "extensional database schema" [22].

3.2.2. Stratified Semantics

We now recall the stratified semantics for a Datalog$^\neg$ program P [22]. As a slight abuse of notation, here we will treat $idb(P)$ as a set of only relation names (without associated arities). First, P is called *syntactically stratifiable* if there is a function $\sigma : idb(P) \to \{1, \ldots, |idb(P)|\}$ such that for each rule $\varphi \in P$, having some head predicate T, the following conditions are satisfied:

- $\sigma(R) \leq \sigma(T)$ for each $R(\bar{u}) \in pos_\varphi|_{idb(P)}$;
- $\sigma(R) < \sigma(T)$ for each $R(\bar{u}) \in neg_\varphi|_{idb(P)}$.

For $R \in idb(P)$, we call $\sigma(R)$ the *stratum number* of R. For technical convenience, we may assume that if there is an $R \in idb(P)$ with $\sigma(R) > 1$ then there is an $S \in idb(P)$ with $\sigma(S) = \sigma(R) - 1$. Intuitively, function σ partitions P into a sequence of semi-positive Datalog$^\neg$ programs P_1, ..., P_k with $k \leq |idb(P)|$ such that for each $i = 1, \ldots, k$, the program P_i contains the rules of P whose head predicate has stratum number i. This sequence is called a *syntactic stratification* of P. We can now apply the *stratified semantics* to P: for an input I over $sch(P)$, we first compute the fixpoint $P_1(I)$, then the fixpoint $P_2(P_1(I))$, etc. The *output of P on input I*, denoted $P(I)$, is defined as $P_k(P_{k-1}(\ldots P_1(I)\ldots))$. It is well known that the output of P does not depend on the chosen syntactic stratification (if more than one exists). Not all Datalog$^\neg$ programs are syntactically stratifiable.

3.2.3. Stable Model Semantics

We now recall the stable model semantics for a Datalog$^\neg$ program P [10, 24]. Let I be an instance over $sch(P)$. Let $\varphi \in P$. Let V be a valuation for φ whose image is contained in $adom(I) \cup C$, where C is the set of all constants appearing in P. Valuation V does not have to be satisfying for φ on I. Together, V and φ give rise to a ground rule ψ, obtained from φ by replacing each $u \in vars(\varphi)$ with $V(u)$. We call ψ a *ground rule of φ with respect to I*. Let $ground(\varphi, I)$ denote the set of all ground rules of φ with respect to I. The *ground program of P on I*, denoted $ground(P, I)$, is defined as $\bigcup_{\varphi \in P} ground(\varphi, I)$. Note, if $I = \emptyset$, the set $ground(P, I)$ contains only rules whose ground atoms are made with C, or atoms that are nullary.

Let M be another instance over $sch(P)$. We write $ground_M(P, I)$ to denote the program obtained from $ground(P, I)$ as follows:

1. remove every rule $\psi \in ground(P, I)$ for which $neg_\psi \cap M \neq \emptyset$;
2. remove the negative (ground) body atoms from all remaining rules.

Note, $ground_M(P, I)$ is a positive program. We say that M is a *stable model of P on input I* if M is the output of $ground_M(P, I)$ on input I. If so, the semantics of positive Datalog$^\neg$ programs implies $I \subseteq M$. Not all Datalog$^\neg$ programs have stable models on every input [10].

3.3. Network and Distributed Databases

A *(computer) network* is a nonempty finite set $\mathcal{N}$ of *nodes*, which are values in **dom**. Intuitively, $\mathcal{N}$ represents the identifiers of compute nodes involved in a distributed system. Communication channels (edges) are not explicitly represented because we allow a node x to send a message to any node y, as long as x knows about y by means of input relations or received messages. When using Dedalus for general distributed or cluster computing, the delivery of messages is handled by the network layer, which is abstracted away. But Dedalus programs can also describe the network layer itself [2, 3], in which case we would restrict attention to programs where nodes only send messages to nodes to which they are explicitly linked; these nodes would again be provided as input.

A *distributed database instance* H over a network $\mathcal{N}$ and a database schema $\mathcal{D}$ is a function that maps every node of $\mathcal{N}$ to an ordinary finite database instance over $\mathcal{D}$. This represents how data over the same schema $\mathcal{D}$ is spread over a network.

As a small example of a distributed database instance, consider the following instance H over a network $\mathcal{N} = \{x, y\}$ and a schema $\mathcal{D} = \{R^{(1)}, S^{(1)}\}$: $H(x) = \{R(a), S(b)\}$ and $H(y) = \{S(c)\}$. In words: we put facts $R(a)$ and $S(b)$ at node x, and we put fact $S(c)$ at node y.

3.4. Dedalus Programs

We now recall the language Dedalus, that can be used to describe distributed computations [6, 7, 3]. Essentially, Dedalus is an extension of Datalog$^\neg$ to represent updatable memory for the nodes of a network and to provide a mechanism for communication between these nodes. To simplify notation, we present Dedalus as Datalog$^\neg$ extended with annotations.[4]

Let $\mathcal{D}$ be a database schema. We write $\mathbf{B}\{\bar{\mathtt{v}}\}$, where $\bar{\mathtt{v}}$ is a tuple of variables, to denote any sequence β of literals over database schema $\mathcal{D}$, such that the variables in β are precisely those in the tuple $\bar{\mathtt{v}}$. Let $R(\bar{\mathtt{u}})$ denote any atom over $\mathcal{D}$. There are three types of Dedalus rules over $\mathcal{D}$:

- A *deductive* rule is a normal Datalog$^\neg$ rule over $\mathcal{D}$.
- An *inductive* rule is of the form
$$R(\bar{\mathtt{u}})\bullet \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}.$$
- An *asynchronous* rule is of the form
$$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}.$$

For inductive rules, the annotation '$\bullet$' can be likened to the transfer of "tokens" in a Petri net from the old state to the new state. For asynchronous rules, the annotation '$\mid \mathtt{y}$' with $\mathtt{y} \in$ **var** means that the derived head facts are transferred ("piped") to the addressee node represented by $\mathtt{y}$. Deductive, inductive and asynchronous rules will express respectively local computation, updatable memory, and message sending (cf. Section 3.5). Like in Section 3.2, a Dedalus rule is called *safe* if all its variables occur in at least one positive body atom.

We already provide some intuition of how asynchronous rules operate. There are four conceptual time points involved in the execution of an asynchronous rule: the time when the body is evaluated; the time when the derived fact is sent to the addressee; the time when the fact arrives at the addressee; and, the time when the arrived fact becomes visible at the addressee. In the semantics of Section 3.5, the first two time points coincide and the last two time points coincide; and, there is no upper bound on the interval between these two pairs, although it will be finite.

To illustrate the syntax, if $\mathcal{D} = \{R^{(2)}, S^{(1)}, T^{(2)}\}$, then the following three rules are examples of, respectively, deductive, inductive and asynchronous rules over $\mathcal{D}$:

$$T(\mathtt{u}, \mathtt{v}) \leftarrow R(\mathtt{u}, \mathtt{v}),\ \neg S(\mathtt{v}).$$

$$T(\mathtt{u}, \mathtt{v})\bullet \leftarrow R(\mathtt{u}, \mathtt{v}).$$

$$T(\mathtt{u}, \mathtt{v}) \mid \mathtt{y} \leftarrow R(\mathtt{u}, \mathtt{v}),\ S(\mathtt{y}).$$

Now consider the following definition:

Definition 3.1. A Dedalus *program over a schema* $\mathcal{D}$ is a set of deductive, inductive and asynchronous Dedalus rules over $\mathcal{D}$, such that all rules are safe, and the set of deductive rules is syntactically stratifiable.

In the current work, we will additionally assume that Dedalus programs are constant-free, as is common in the theory of database query languages, and which is not really a limitation, since constants that are important for the program can always be indicated by unary relations in the input.

Let $\mathcal{P}$ be a Dedalus program. The definitions of $sch(\mathcal{P})$, $idb(\mathcal{P})$, and $edb(\mathcal{P})$ are like for Datalog$^\neg$ programs. An *input* for $\mathcal{P}$ is a *distributed* database instance over some network and the schema $edb(\mathcal{P})$.

Next, we give the operational semantics for Dedalus in Section 3.5. We formalize the output of a Dedalus program in Section 3.6, and we give an example program in Section 3.7.

[4]These annotations correspond to syntactic sugar in the previous presentations of Dedalus.

3.5. Operational Semantics

Dedalus has a formal operational semantics [8, 9]. Since we need this semantics to state the results of the present paper, we recall it in this section.

We describe how a network executes a Dedalus program $\mathcal{P}$ when an input distributed database instance H is given. The essence is as follows. Let $\mathcal{N}$ be the network that H is over. Every node of $\mathcal{N}$ runs the *same* program $\mathcal{P}$, and a node has access only to its own local state and any received messages. The nodes are made active one by one in some arbitrary order, and this continues an infinite number of times. During each active moment of a node x, called a *local (computation) step*, node x receives message facts and applies its deductive, inductive and asynchronous rules. Concretely, the deductive rules, forming a stratified Datalog$^\neg$ subprogram, are applied to the incoming messages and the previous state of x. Deductive rules "complete" the available facts by adding all new facts that can be logically derived from them. Next, the inductive rules are applied to the output of the deductive subprogram, and these allow x to store facts in its memory: these facts become visible in the next local step of x. Finally, the asynchronous rules are also applied to the output of the deductive subprogram, and these allow x to send facts to the other nodes or to itself. These facts become visible at the addressee after some arbitrary delay, which represents asynchronous communication. We will refer to local steps simply as "steps". The next subsections make the above sketch concrete.

3.5.1. Configurations

Let $\mathcal{P}$, H, and $\mathcal{N}$ be like above. A configuration describes the network at a certain point in its evolution. Formally, a *configuration* of $\mathcal{P}$ on H is a pair $\rho = (st, bf)$ where

- st is a function mapping each node of $\mathcal{N}$ to an instance over $sch(\mathcal{P})$; and,
- bf is a function mapping each node of $\mathcal{N}$ to a set of pairs of the form $(i, \boldsymbol{f})$, where $i \in \mathbb{N}$ and $\boldsymbol{f}$ is a fact over $idb(\mathcal{P})$.

We call st and bf the *state* and *(message) buffer* respectively. The state says for each node what facts it has stored in its memory, and the message buffer bf says for each node what messages have been sent to it but that are not yet received. The reason for having numbers i, called *send-tags*, attached to facts in the image of bf is merely a technical convenience: these numbers help separate multiple instances of the same fact when it is sent at different moments (to the same addressee), and these send-tags will not be visible to the Dedalus program. For example, if the buffer of a node x simultaneously contains the pairs $(1, \boldsymbol{f})$ and $(5, \boldsymbol{f})$, this means that fact $\boldsymbol{f}$ was sent to x during the operational network transitions with indices 1 and 5, and that both particular instances of $\boldsymbol{f}$ are not yet delivered to x. This will become more concrete in Section 3.5.3.

The *start configuration of* $\mathcal{P}$ *on input* H, denoted $start(\mathcal{P}, H)$, is the configuration $\rho = (st, bf)$ defined by $st(x) = H(x)$ and $bf(x) = \emptyset$ for each $x \in \mathcal{N}$.

3.5.2. Subprograms

We look at the operations executed locally during each step of a node. We split $\mathcal{P}$ into three subprograms, containing respectively the deductive, inductive and asynchronous rules. These programs are used in Section 3.5.3.

First, we define $deduc_\mathcal{P}$ to be the Datalog$^\neg$ program consisting of all deductive rules of $\mathcal{P}$. Secondly, we define $induc_\mathcal{P}$ to be the Datalog$^\neg$ program consisting of all inductive rules of $\mathcal{P}$ after removing the annotation '$\bullet$'. Thirdly, we define $async_\mathcal{P}$ to be the Datalog$^\neg$ program consisting of all rules

$$T(\mathtt{y}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \mathtt{y}\}$$

where

$$T(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \mathtt{y}\}$$

is an asynchronous rule of $\mathcal{P}$. In $async_\mathcal{P}$, the first head variable represents the addressee. Note, programs $deduc_\mathcal{P}$, $induc_\mathcal{P}$ and $async_\mathcal{P}$ are just Datalog$^\neg$ programs over $sch(\mathcal{P})$. Moreover, the definition of $\mathcal{P}$ implies that $deduc_\mathcal{P}$ is syntactically stratifiable. Possibly $induc_\mathcal{P}$ and $async_\mathcal{P}$ are not syntactically stratifiable.

Now we define the semantics of the three subprograms. Let I be an instance over $sch(\mathcal{P})$. We define the output of $deduc_{\mathcal{P}}$ on input I, denoted $deduc_{\mathcal{P}}(I)$, to be given by the stratified semantics. This implies $I \subseteq deduc_{\mathcal{P}}(I)$. We define the output of $induc_{\mathcal{P}}$ on input I, denoted $induc_{\mathcal{P}}(I)$, to be the set of facts derived by the rules of $induc_{\mathcal{P}}$ for all possible satisfying valuations in I, in just one derivation step (i.e., no fixpoint). The output for $async_{\mathcal{P}}$ on input I, denoted $async_{\mathcal{P}}(I)$, is defined as for $induc_{\mathcal{P}}$, but now using $async_{\mathcal{P}}$ instead of $induc_{\mathcal{P}}$.

Regarding data complexity [23], the output of each subprogram can be computed in PTIME with respect to the size of its input. The overall data complexity of Dedalus is discussed in Section 4.2.

3.5.3. Transitions and Runs

Transitions formalize how to go from one configuration to another. Here we use the subprograms of $\mathcal{P}$. Transitions are chained to form a *run*. Regarding notation, for a set m of pairs of the form $(i, \boldsymbol{f})$, we define $untag(m) = \{\boldsymbol{f} \mid \exists i \in \mathbb{N} : (i, \boldsymbol{f}) \in m\}$.

A *transition with send-tag* $i \in \mathbb{N}$ is a five-tuple $(\rho_1, x, m, i, \rho_2)$ such that $\rho_1 = (st_1, bf_1)$ and $\rho_2 = (st_2, bf_2)$ are configurations of $\mathcal{P}$ on input H, $x \in \mathcal{N}$, $m \subseteq bf_1(x)$, and, letting

$$
\begin{aligned}
&I = st_1(x) \cup untag(m),\\
&D = deduc_{\mathcal{P}}(I),\\
&\delta^{i\to y} = \{(i,\, R(\bar{a})) \mid R(y, \bar{a}) \in async_{\mathcal{P}}(D)\} \text{ for each } y \in \mathcal{N},
\end{aligned}
$$

for x and each $y \in \mathcal{N} \setminus \{x\}$ we have

$$
\begin{aligned}
&st_2(x) = H(x) \cup induc_{\mathcal{P}}(D), \qquad && st_2(y) = st_1(y),\\
&bf_2(x) = (bf_1(x) \setminus m) \cup \delta^{i\to x}, \qquad && bf_2(y) = bf_1(y) \cup \delta^{i\to y}.
\end{aligned}
$$

We call ρ_1 and ρ_2 respectively the *source-* and *target*-configuration, and say this transition is *of* the *active* node x. Intuitively, the transition expresses how x reads its old state together with the received facts in $untag(m)$ (thus without the tags). Subprogram $deduc_{\mathcal{P}}$ completes this information; the new state of x is set to the input facts of x united with all facts derived by subprogram $induc_{\mathcal{P}}$; and, subprogram $async_{\mathcal{P}}$ generates *messages*, whose first component indicates the addressee.[5] Note, $induc_{\mathcal{P}}$ and $async_{\mathcal{P}}$ do not influence each other, and can be thought of as being executed in parallel. Also, for each $y \in \mathcal{N}$, the set $\delta^{i\to y}$ contains all messages addressed to y, with send-tag i attached. Messages with an addressee outside the network are ignored. This way of defining local computation closely corresponds to that of the language Webdamlog [25]. If $m = \emptyset$, we call the transition a *heartbeat*.

A *run* $\mathcal{R}$ *of* $\mathcal{P}$ *on input* H is an infinite sequence of transitions, such that *(i)* the source configuration of the first transition is $start(\mathcal{P}, H)$, *(ii)* the target-configuration of each transition is the source-configuration of the next transition, and *(iii)* the transition at ordinal i of the sequence uses send-tag i. Ordinals start at 0 for technical convenience. The resulting transition system is highly non-deterministic because in each transition we can choose the active node and also what messages to deliver; the latter choice is represented by the set m from above.

It is natural to require certain "fairness" conditions on the execution of a system [26, 27, 28]. A run $\mathcal{R}$ of $\mathcal{P}$ on H is called *fair* if *(i)* every node is the active node in an infinite number of transitions, and *(ii)* every sent message is eventually delivered. Note, a fair run exists for every input because heartbeats remain possible even when there are no messages to deliver. We only consider fair runs.

3.5.4. Timestamps

For each transition i of a run, we define the *timestamp* of the active node x during i to be the number of transitions of x that come strictly before i. This can be thought of as the *local* (zero-based) clock of x during i. For example, suppose we have the following sequence of active nodes: x, y, y, x, x, etc. If we would write the timestamps next to the nodes, we get this sequence: $(x, 0)$, $(y, 0)$, $(y, 1)$, $(x, 1)$, $(x, 2)$, etc.

[5] Note, input facts are preserved by the transition. This aligns with the design of Dedalus, where we do not allow facts to be retracted; only negation as failure is permitted.

$$T(\mathtt{u},\mathtt{v}) \leftarrow R(\mathtt{u},\mathtt{v}).$$
$$T(\mathtt{u},\mathtt{v}) \leftarrow R(\mathtt{u},\mathtt{w}),\ T(\mathtt{w},\mathtt{v}).$$
$$T(\mathtt{u},\mathtt{v}) \mid \mathtt{y} \leftarrow T(\mathtt{u},\mathtt{v}),\ \mathtt{Node}(\mathtt{y}).$$
$$T(\mathtt{u},\mathtt{v})\bullet \leftarrow T(\mathtt{u},\mathtt{v}).$$

Figure 1: Dedalus program for transitive closure

3.6. Output and Consistency

Let $\mathcal{P}$ be a Dedalus program. We formalize the output of $\mathcal{P}$. Assume a subset $out(\mathcal{P}) \subseteq idb(\mathcal{P})$, called the *output schema*, is selected: the relation names in $out(\mathcal{P})$ designate the intended output of the program. Following Marczak et al. [29], we define this output based on *ultimate* facts. In a run $\mathcal{R}$ of $\mathcal{P}$ on an input H, we say that a fact $\boldsymbol{f}$ over schema $out(\mathcal{P})$ is *ultimate* at some node x if there is some transition of $\mathcal{R}$ after which $\boldsymbol{f}$ is output by $deduc_{\mathcal{P}}$ during every transition of x. Thus, $\boldsymbol{f}$ is eventually always present at x. The output of $\mathcal{R}$, denoted $output(\mathcal{R})$, is the union of the ultimate facts across all nodes. Note, we ignore what node is responsible for what piece of the output.

Because the operational semantics is nondeterministic, different runs can produce different outputs. Now, program $\mathcal{P}$ is called *consistent* if individually for every input H, every run produces the *same* output, which we denote as $outInst(\mathcal{P}, H)$. Guaranteeing or deciding consistency in special cases is an important research topic [25, 29, 30]. Not surprisingly, consistency of Dedalus programs is undecidable in general; this is argued in Appendix A.

3.7. Example

Figure 1 gives an example Dedalus program. Each node is initialized with a local relation $R^{(2)}$ that represents a graph, and we assume the existence of a local relation $\mathtt{Node}^{(1)}$ that always contains all nodes in the network at hand (cf. Section 4.1). The first two rules are deductive, and they compute the transitive closure of R during each step of a node. The third rule is asynchronous, and it lets each node broadcast its transitive edges to every other node. The fourth rule is inductive, and it lets each node remember the computed or received transitive edges. The inductive rule causes each node to integrate all received transitive edges in its local transitive closure computation (as performed by the deductive rules).

The overall effect is that eventually all nodes have stored the transitive closure of the entire input graph that is the union of all local input graphs. Note, this global transitive closure is obtained on arbitrary distributions of relation R over the nodes: the nodes essentially always share the input facts with each other, implicitly through relation T.

4. Dedalus Fundamental Properties

Using the above definitions, we now investigate the expressivity and complexity of Dedalus. This discussion uses arguments similar to our previous work [13].

4.1. Expressivity

We formalize the expressivity of Dedalus with standard database queries, that are defined in Section 3.1. First, an ordinary database instance I over a schema $\mathcal{D}$ can be *distributed* over a network $\mathcal{N}$ by putting each fact of I on at least one node, resulting in a distributed database instance over $\mathcal{N}$ and $\mathcal{D}$.

We say that a Dedalus program $\mathcal{P}$ *(distributedly) computes* a query $\mathcal{Q}$ if $\mathcal{P}$ is consistent and for every input instance I for $\mathcal{Q}$, for every network $\mathcal{N}$, for every distribution H of I over $\mathcal{N}$, we have $outInst(\mathcal{P}, H) = \mathcal{Q}(I)$. To compute non-monotone queries, every node needs its own identifier and the identifiers of the other nodes, or equivalent information [13]. Therefore, we restrict attention to Dedalus programs $\mathcal{P}$ for which $edb(\mathcal{P})$ contains two unary relations $\mathtt{Id}$ and $\mathtt{Node}$, and each input H, over a network $\mathcal{N}$, has for each $x \in \mathcal{N}$ the facts $\{\mathtt{Id}(x)\} \cup \{\mathtt{Node}(y) \mid y \in \mathcal{N}\}$ over these relations.

We now argue that Dedalus captures the queries expressible in the language While [22].

4.1.1. Upper Bound

Let $\mathcal{P}$ be a Dedalus program that computes a query $\mathcal{Q}$. By assumption, $\mathcal{P}$ computes $\mathcal{Q}$ on a single-node network $\mathcal{N} = \{x\}$ as well as on any other network; in the single-node case, node x is always given all input facts. Moreover, on node x, program $\mathcal{P}$ by assumption also computes $\mathcal{Q}$ when we only consider runs that deliver the entire message buffer during each transition. In such runs, *(i)* the buffer of x degenerates to a set because only a set of messages is sent in each transition; and, *(ii)* messages are always immediately delivered. To achieve the same effect, we can simply replace asynchronous rules by inductive rules. Next, in the language While, we can write a sequence of loops that each simulates the fixpoint computation of one stratum of the deductive subprogram of $\mathcal{P}$. This sequence can be wrapped inside a larger loop, that includes at the end some FO queries to simulate the effect of the inductive rules, that allow computed facts to be remembered between different iterations of the simulated deductive rules. Finally, the While program can be extended with a mechanism to detect infinite looping, using the technique of Abiteboul and Simon [31]. This way, the resulting program will always terminate.[6]

4.1.2. Lower Bound

Let $\mathcal{Q}$ be a query expressible by a While program W. We assume that W consists of a single loop (with no nested loops) followed by a sequence of FO queries setting the final output relations; W can always be rewritten into this form. We construct a Dedalus program $\mathcal{P}$ to compute $\mathcal{Q}$ as follows. First, program $\mathcal{P}$ executes a coordination phase to let each node obtain all available input facts on the network, which requires relations `Id` and `Node` [13]. At each node, the end of this phase is signaled by the derivation of a fact `ready()`, that is persisted by inductive rules. After obtaining `ready()`, every node locally simulates While program W: one local computation step of $\mathcal{P}$ corresponds with one iteration of the loop of W, where deductive rules simulate the FO queries of W, and inductive rules simulate changes to temporary relations of W. Once the condition of the loop of W becomes false, $\mathcal{P}$ simulates the final sequence of FO queries in W to generate the output, that is persisted with inductive rules.

4.2. Complexity

In this section, we argue that reasoning with Dedalus has PSPACE data complexity. A first insight is already provided by the expressivity analysis from above. Indeed, for each Dedalus program $\mathcal{P}$ that expresses a query $\mathcal{Q}$, we can define the following evaluation problem $eval_{\mathcal{P}}^{\text{query}}$: given an ordinary input instance I for $\mathcal{Q}$, and an output fact $\boldsymbol{f}$, output *true* if and only if $\mathcal{P}$ outputs $\boldsymbol{f}$ when given as input an arbitrary distribution H of I over a network. Because Dedalus captures the While-queries, and because the language While is complete for PSPACE [22], each $eval_{\mathcal{P}}^{\text{query}}$ problem is in PSPACE and there is a Dedalus program $\mathcal{P}$ for which $eval_{\mathcal{P}}^{\text{query}}$ is PSPACE-hard.

We also discuss in more detail the data complexity of Dedalus in a manner not tied to queries. For each *consistent* Dedalus program $\mathcal{P}$, we define the evaluation problem $eval_{\mathcal{P}}$ as follows: given an input distributed database instance H for $\mathcal{P}$ and a fact $\boldsymbol{f}$ over $out(\mathcal{P})$, output *true* if and only if $\boldsymbol{f} \in outInst(\mathcal{P}, H)$. The complexity of this evaluation problem is discussed next.[7]

4.2.1. Upper Bound

Let $\mathcal{P}$ be a consistent Dedalus program. We argue that $eval_{\mathcal{P}}$ is in PSPACE. Consider a run $\mathcal{R}$ of $\mathcal{P}$ on H that proceeds in rounds: in each round, we go over the nodes in some arbitrary order and we deliver to each node the message buffer that it had at the *beginning* of the round. By consistency of $\mathcal{P}$, it is sufficient to observe whether $\boldsymbol{f}$ is produced at some node during $\mathcal{R}$. Next, we show that polynomial space suffices for simulating $\mathcal{R}$ and that actually only a finite prefix needs to be examined.

For the space bound, we first observe that we can incrementally modify a single configuration to simulate $\mathcal{R}$. To bound the space of a single configuration, we note that a configuration of $\mathcal{R}$ contains at most

[6] Note, the issue of infinite looping is not important for Dedalus programs in general, because their output is defined over infinite runs.

[7] In this context, a reasonable encoding of H is as follows: we are given a list of node identifiers; and, next follow lists of local input facts, one list for each node. Such lists can be empty.

$G = n(F + 2nF)$ facts, where n is the number of nodes and F is the total number of facts that can be made with the active domain of H. Indeed, in $\mathcal{R}$, for each node, *(i)* the state contains at most F facts; and, *(ii)* the buffer contains at most $2nF$ facts by design of the rounds, where n indicates the number of senders, and where the factor 2 corresponds to the buffer of a node that is made active at the very end of a round. Size G is polynomial: n is linear, and F is polynomial because $\mathcal{P}$ is constant.[8] Note, the number of possible configurations in $\mathcal{R}$ is 2^G.

Lastly, because the configuration at the beginning of a round completely determines the configuration at the end of a round, we may stop simulating $\mathcal{R}$ after 2^G rounds: no new configurations will be encountered afterward.

4.2.2. Lower Bound

We can define a Dedalus program $\mathcal{P}$ for which $eval_{\mathcal{P}}$ is PSPACE-hard. Concretely, $\mathcal{P}$ takes as input the description of a PSPACE Turing machine and an input tape, and simulates the Turing machine on this input. Now, letting A be a problem from PSPACE, for any input string w for A, we encode the Turing machine M for A and w (together) as an input for $\mathcal{P}$, which constitutes the first input for $eval_{\mathcal{P}}$; the second input for $eval_{\mathcal{P}}$ is an output fact of $\mathcal{P}$ representing that M accepts w. The technical details can be found in Appendix B.

5. CRON Conjecture and Non-Causality

We recall the CRON (Causality Required Only for Non-monotonicity) conjecture, which was informally stated as follows [3]:

CRON Conjecture (Informal). *Program semantics require causal message ordering if and only if the messages participate in non-monotonic derivations.*

The CRON conjecture talks about an intuitive notion of "causality" on messages. As mentioned in the Introduction, causality here stands for the physical constraint that an effect can only happen after its cause. Our operational semantics respects causality because a message can only be delivered after it was sent. When the delivery of one message causes another one to be sent, the second one is delivered in a later transition.

In order to obtain a conjecture that can be formally proved or disproved, we need a formal definition of "requiring causal message ordering". To this end, we first recall in Section 5.1 a declarative semantics for Dedalus in which message sending is causal. Next, in Section 5.2, we modify this declarative semantics to obtain non-causality (sending messages "into the past"). This is then used in Section 6 to formally investigate the CRON conjecture.

5.1. Causal Declarative Semantics

We recall a causal declarative semantics for Dedalus [8, 9]. Throughout this subsection, we fix a Dedalus program $\mathcal{P}$. We give $\mathcal{P}$ a declarative semantics based on applying the stable model semantics to a pure Datalog$^\neg$ program $pure(\mathcal{P})$, obtained from $\mathcal{P}$.

5.1.1. Auxiliary Notations and Relations

Before defining $pure(\mathcal{P})$, we introduce some auxiliary notations and relation names.

Let $R^{(k)} \in sch(\mathcal{P})$. We use facts of the form $R(x, s, a_1, \ldots, a_k)$ to express that fact $R(a_1, \ldots, a_k)$ is present at a node x during its local step s, with $s \in \mathbb{N}$, after program $deduc_{\mathcal{P}}$ is executed. We call x the *location specifier* and s the *timestamp*. In order to represent timestamps, we assume $\mathbb{N} \subseteq \mathbf{dom}$.

We write $sch(\mathcal{P})^{\mathrm{LT}}$ to denote the database schema obtained from $sch(\mathcal{P})$ by incrementing the arity of every relation by two. The two extra components will contain the location specifier and timestamp.[9] For

[8] Concretely, $F = ma^k$, where m is the number of *idb*-relations; a is the size of the active domain; and, k is the largest relation arity. Quantities m and k are constant, and a is linear.

[9] The abbreviation 'LT' stands for "location specifier and timestamp".

an instance I over $sch(\mathcal{P})$, $x \in \mathbf{dom}$ and $s \in \mathbb{N}$, we write $I^{\Uparrow x,s}$ to denote the facts over $sch(\mathcal{P})^{\mathrm{LT}}$ that are obtained by prepending location specifier x and timestamp s to every fact of I. Also, if L is a sequence of literals over $sch(\mathcal{P})$, and $\mathtt{x}, \mathtt{s} \in \mathbf{var}$, we write $L^{\Uparrow \mathtt{x},\mathtt{s}}$ to denote the sequence of literals over $sch(\mathcal{P})^{\mathrm{LT}}$ that is obtained by adding location specifier $\mathtt{x}$ and timestamp $\mathtt{s}$ to the literals in L (negative literals stay negative).

We also need auxiliary relation names, that are assumed not to be used in $sch(\mathcal{P})$; these are listed in Table 1.[10] The concrete purpose of these relations will become clear in the following subsections.

New relation names	Meaning
$\mathtt{all}$	network
$\mathtt{time}$, $\mathtt{tsucc}$, $\neq$	timestamps
$\mathtt{before}$	happens-before relation
$\mathtt{cand}_R$, $\mathtt{chosen}_R$, $\mathtt{other}_R$, for each relation name R in $idb(\mathcal{P})$	messages

Table 1: Relation names not in $sch(\mathcal{P})$

We define the following schema

$$\mathcal{D}_{\mathrm{time}} = \{\mathtt{time}^{(1)}, \mathtt{tsucc}^{(2)}, {\neq}^{(2)}\}.$$

The relation '$\neq$' will be written in infix notation in rules. We consider only the following instance over $\mathcal{D}_{\mathrm{time}}$:

$$\begin{aligned} I_{\mathtt{time}} \quad = \quad & \{\mathtt{time}(s), \mathtt{tsucc}(s, s+1) \mid s \in \mathbb{N}\} \\ & \cup \{(s \neq t) \mid s, t \in \mathbb{N} : s \neq t\}. \end{aligned}$$

Intuitively, the instance $I_{\mathtt{time}}$ provides timestamps together with a successor and nonequality relation.

5.1.2. Causal Transformation

We now incrementally construct $pure(\mathcal{P})$. We use facts of the form $\mathtt{before}(x, s, y, t)$ to express that local step s of node x causally happens before local step t of node y. We use facts of the form $\mathtt{all}(x)$ to say that x is a node of the network at hand.

To start, we add the following rules to $pure(\mathcal{P})$ to express local causality at the nodes, unrelated to messages:

$$\mathtt{before}(\mathtt{x}, \mathtt{s}, \mathtt{x}, \mathtt{t}) \leftarrow \mathtt{all}(\mathtt{x}), \mathtt{tsucc}(\mathtt{s}, \mathtt{t}). \tag{5.1}$$

$$\mathtt{before}(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}) \leftarrow \mathtt{before}(\mathtt{x}, \mathtt{s}, \mathtt{z}, \mathtt{u}), \mathtt{before}(\mathtt{z}, \mathtt{u}, \mathtt{y}, \mathtt{t}). \tag{5.2}$$

Rule (5.1) expresses that on every node, a step causally happens before the next step. Rule (5.2) computes the transitive closure on relation $\mathtt{before}$.

Next, for each rule in $\mathcal{P}$, we specify what corresponding rule (or rules) should be added to $pure(\mathcal{P})$. In particular, the aynchronous rules will also have an impact on the happens-before relation.

For technical convenience, we assume that rules of $\mathcal{P}$ always contain at least one positive body atom. This assumption allows us to more elegantly enforce that head variables in rules of $pure(\mathcal{P})$ also occur in at least one positive body atom.[11] Let $\mathtt{x}, \mathtt{s}, \mathtt{t}, \mathtt{t}' \in \mathbf{var}$ be distinct variables not yet occurring in rules of $\mathcal{P}$. We write $\mathbf{B}\{\bar{\mathtt{v}}\}$, where $\bar{\mathtt{v}}$ is a tuple of variables, to denote any sequence β of literals over $sch(\mathcal{P})$, such that the variables in β are precisely those in $\bar{\mathtt{v}}$. Also recall the notations and relation names from Section 5.1.1.

Deductive Rules. For each deductive rule $R(\bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$ in $\mathcal{P}$, we add to $pure(\mathcal{P})$ the following rule:

$$R(\mathtt{x}, \mathtt{s}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x},\mathtt{s}}. \tag{5.3}$$

This rule expresses that deductively derived facts at some node x during step s are (immediately) visible within step s of x. Note, all atoms in this rule are over $sch(\mathcal{P})^{\mathrm{LT}}$.

[10] In practice, auxiliary relations can be differentiated from those in $sch(\mathcal{P})$ by a namespace mechanism.

[11] This assumption is not really a restriction, since a nullary positive body atom is already sufficient.

Inductive Rules. For each inductive rule $R(\bar{\mathtt{u}})\bullet \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$ in $\mathcal{P}$, we add to $\mathit{pure}(\mathcal{P})$ the following rule:

$$R(\mathtt{x}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x},\mathtt{s}}, \mathtt{tsucc}(\mathtt{s}, \mathtt{t}). \tag{5.4}$$

This rule expresses that inductively derived facts becomes visible in the *next* step of the *same* node.

Asynchronous Rules. We use facts of the form $\mathtt{cand}_R(x, s, y, t, \bar{a})$ to express that node x at its step s sends a message $R(\bar{a})$ to node y, and that t could be the arrival timestamp of this message at y.[12] Within this context, we use a fact $\mathtt{chosen}_R(x, s, y, t, \bar{a})$ to say that t is the *effective* arrival timestamp of this message at y. Lastly, a fact $\mathtt{other}_R(x, s, y, t, \bar{a})$ means that t is *not* the arrival timestamp of the message. Now, for each asynchronous rule

$$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}$$

in $\mathcal{P}$, letting $\bar{\mathtt{w}}$ be a tuple of new and distinct variables with $|\bar{\mathtt{w}}| = |\bar{\mathtt{u}}|$, we add to $\mathit{pure}(\mathcal{P})$ the following rules, for which the intuition is given below:

$$\begin{aligned}\mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}^{\Uparrow \mathtt{x},\mathtt{s}}, \mathtt{all}(\mathtt{y}), \mathtt{time}(\mathtt{t}),\\ \neg\mathtt{before}(\mathtt{y}, \mathtt{t}, \mathtt{x}, \mathtt{s}).\end{aligned} \tag{5.5}$$

$$\mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}) \leftarrow \mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}), \neg\mathtt{other}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}). \tag{5.6}$$

$$\mathtt{other}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}) \leftarrow \mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}), \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}', \bar{\mathtt{w}}), \mathtt{t} \neq \mathtt{t}'. \tag{5.7}$$

$$R(\mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}) \leftarrow \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}). \tag{5.8}$$

$$\mathtt{before}(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}) \leftarrow \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{w}}). \tag{5.9}$$

Rule (5.5) represents the messages that are sent. Relation `all` is assumed to contain all nodes of the network, and it thus represents the range of valid addressees. Candidate arrival timestamps are restricted by relation `before` to enforce causality. Intuitively, this restriction prevents cycles from occurring in relation `before`; this aligns with the operational semantics, where the happens-before relation is a strict partial order [9].

Together, rules (5.6) and (5.7) have the effect that exactly one arrival timestamp will be chosen under the stable model semantics. This technical construction is due to Saccà and Zaniolo [24], who show how to express dynamic choice under the stable model semantics.

Rule (5.8) represents the actual arrival of an R-message with the chosen arrival timestamp. Rule (5.9) adds the causal restriction that the local step of the sender happens before the arrival step of the addressee. Together with the previously introduced rules (5.1) and (5.2), this will make sure that when the addressee later *causally* replies to the sender, the reply — as generated by a rule of the form (5.5) — will arrive after this first send-step of the sender.

Note, if multiple asynchronous rules in $\mathcal{P}$ have the same head predicate R, only new $\mathtt{cand}_R$-rules have to be added because the rules (5.6)–(5.9) are general for all R-messages. Moreover, if there are asynchronous rules in $\mathcal{P}$, program $\mathit{pure}(\mathcal{P})$ is not syntactically nor locally stratifiable because relation `before` negatively depends on itself through rules of the following forms, in order: (5.5), (5.6) and (5.9) [9].

To illustrate, applying the above transformation to the Dedalus program in Figure 1 gives the pure Datalog$^\neg$ program shown in Figure 2.

5.1.3. Input and Stable Models

Now we are ready to define the declarative semantics of $\mathcal{P}$. Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$. Let $\mathit{pure}(\mathcal{P})$ be as previously constructed. We define $\mathit{decl}(H)$ to be the following instance over schema $\mathit{edb}(\mathcal{P})^{\mathrm{LT}} \cup \{\mathtt{all}^{(1)}\} \cup \mathcal{D}_{\mathrm{time}}$:

$$\begin{aligned}\mathit{decl}(H) \quad = \quad & \{R(x, s, \bar{a}) \mid x \in \mathcal{N},\ s \in \mathbb{N},\ R(\bar{a}) \in H(x)\}\\ & \cup \{\mathtt{all}(x) \mid x \in \mathcal{N}\} \cup I_{\mathtt{time}}.\end{aligned}$$

[12] Here, '`cand`' abbreviates "candidate".

$$\texttt{before}(\texttt{x},\texttt{s},\texttt{x},\texttt{t}) \leftarrow \texttt{all}(\texttt{x}),\ \texttt{tsucc}(\texttt{s},\texttt{t}).$$
$$\texttt{before}(\texttt{x},\texttt{s},\texttt{y},\texttt{t}) \leftarrow \texttt{before}(\texttt{x},\texttt{s},\texttt{z},\texttt{u}),\ \texttt{before}(\texttt{z},\texttt{u},\texttt{y},\texttt{t}).$$

$$T(\texttt{x},\texttt{s},\texttt{u},\texttt{v}) \leftarrow R(\texttt{x},\texttt{s},\texttt{u},\texttt{v}).$$
$$T(\texttt{x},\texttt{s},\texttt{u},\texttt{v}) \leftarrow R(\texttt{x},\texttt{s},\texttt{u},\texttt{w}),\ T(\texttt{x},\texttt{s},\texttt{w},\texttt{v}).$$

$$\texttt{cand}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}) \leftarrow T(\texttt{x},\texttt{s},\texttt{u},\texttt{v}),\ \texttt{Node}(\texttt{x},\texttt{s},\texttt{y}),\ \texttt{all}(\texttt{y}),\ \texttt{time}(\texttt{t}),\ \neg\texttt{before}(\texttt{y},\texttt{t},\texttt{x},\texttt{s}).$$
$$\texttt{chosen}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}) \leftarrow \texttt{cand}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}),\ \neg\texttt{other}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}).$$
$$\texttt{other}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}) \leftarrow \texttt{cand}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}),\ \texttt{chosen}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t}',\texttt{u},\texttt{v}),\ \texttt{t} \neq \texttt{t}'.$$
$$T(\texttt{y},\texttt{t},\texttt{u},\texttt{v}) \leftarrow \texttt{chosen}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}).$$
$$\texttt{before}(\texttt{x},\texttt{s},\texttt{y},\texttt{t}) \leftarrow \texttt{chosen}_T(\texttt{x},\texttt{s},\texttt{y},\texttt{t},\texttt{u},\texttt{v}).$$

$$T(\texttt{x},\texttt{t},\texttt{u},\texttt{v}) \leftarrow T(\texttt{x},\texttt{s},\texttt{u},\texttt{v}),\ \texttt{tsucc}(\texttt{s},\texttt{t}).$$

Figure 2: Pure Datalog$^\neg$ program, with causality

Intuitively, we make for each node its input facts available at all timestamps; we provide the set of all nodes; and, $I_{\texttt{time}}$ provides the timestamps. Note, $decl(H)$ is infinite because $\mathbb{N}$ is infinite.

Recall the stable model semantics for Datalog$^\neg$ programs from Section 3.2.3. Let M be a stable model of $pure(\mathcal{P})$ on $decl(H)$. We say that M is *locally finite* if M contains for each $(y,t) \in \mathcal{N} \times \mathbb{N}$ only finitely many facts of the form $\texttt{chosen}_R(x,s,y,t,\bar{a})$. This expresses that every node y receives only a finite number of messages on every timestamp t. See also the remarks below.

Now, we call any locally finite stable model M of $pure(\mathcal{P})$ on $decl(H)$ a *model* of $\mathcal{P}$ on input H. Importantly, we are using stable models of $pure(\mathcal{P})$, not of $\mathcal{P}$.

Remark (Choice of stable models)**.** We have used the stable model semantics for $pure(\mathcal{P})$ because this semantics appears to be commonly used for Datalog$^\neg$. Moreover, it allows a seemingly elegant way to think about asynchronous communication: the rules of $pure(\mathcal{P})$ just say when a distributed computation is allowable, and we do not have to think in operational terms like in what order the rules should fire. Nonetheless, it might be an option for future work to define the declarative semantics with Datalog extensions for nondeterminism [32].

Remark (Local finiteness)**.** Regarding the local finiteness constraint, we first note that this property emerges spontaneously in the operational semantics, because only a finite number of transitions come before the local step of a node, and because each transition can only send a finite number of messages. In the declarative semantics, however, using the construction of $pure(\mathcal{P})$ as given above, this constraint is not automatically satisfied for every stable model. In principle, the constraint could be directly enforced with additional rules of $pure(\mathcal{P})$ [9].[13] But in this paper, we just need the assumption that the local finiteness constraint is satisfied, no matter how it is enforced.

Remark (Well-formedness)**.** We call a model M of $\mathcal{P}$ on H *well-formed* if *(i)* for each $R(x,s,\bar{a}) \in M|_{sch(\mathcal{P})}$, we have $x \in \mathcal{N}$ and $s \in \mathbb{N}$; *(ii)*, letting $c \in \{\texttt{cand},\texttt{chosen},\texttt{other}\}$, for each $c_R(x,s,y,t,\bar{a}) \in M$ we have $x,y \in \mathcal{N}$ and $s,t \in \mathbb{N}$; and, *(iii)* for each $\texttt{before}(x,s,y,t) \in M$, we have $x,y \in \mathcal{N}$ and $s,t \in \mathbb{N}$. Using the definition of stable model, it can be shown that model M is always well-formed.

[13] Intuitively, these additional rules check for each receiving node, that for each of its timestamps, there are only a finite number of sending-timestamps over all arriving messages.

5.1.4. Correspondence with the Operational Semantics

We now recall the connection of the declarative semantics with the operational semantics [8, 9]. Towards this end, we first capture the computed data during a run as a set of facts that we call the *trace*.

Let $\mathcal{R}$ be a run of $\mathcal{P}$ on some input H, over a network $\mathcal{N}$. For each transition $i \in \mathbb{N}$, let x_i denote the active node, and let D_i denote the output of subprogram $deduc_{\mathcal{P}}$ during i. Intuitively, D_i contains *all* local facts over $sch(\mathcal{P})$ that x_i has during transition i. For each transition $i \in \mathbb{N}$, let $loc_{\mathcal{R}}(i)$ denote the timestamp of x_i during transition i, as defined in Section 3.5.4. Also recall the notations for the location specifier and timestamp from Section 5.1.1. Now, the *trace* of $\mathcal{R}$ is the following instance over $sch(\mathcal{P})^{\mathrm{LT}}$:

$$trace(\mathcal{R}) = \bigcup_{i \in \mathbb{N}} D_i^{\Uparrow x_i,\, loc_{\mathcal{R}}(i)}.$$

In words: the trace represents all locally computed facts during each transition, additionally carrying the location specifier and timestamp of the active node. The trace shows in detail what happens in the run.

Now we can present the connection between the operational and causal declarative semantics:

Theorem 5.1 ([8, 9])**.** *Let $\mathcal{P}$ be a Dedalus program. For each input H,*

(i) *for every fair run $\mathcal{R}$ of $\mathcal{P}$ there is a model M of $\mathcal{P}$ such that $trace(\mathcal{R}) = M|_{sch(\mathcal{P})}$, and*
(ii) *for every model M of $\mathcal{P}$ there is a fair run $\mathcal{R}$ of $\mathcal{P}$ such that $trace(\mathcal{R}) = M|_{sch(\mathcal{P})}$.*

5.2. Non-Causal Declarative Semantics

In Section 5.1 we have seen that the operational semantics of Dedalus is equivalent to a declarative semantics based on stable models. Concretely, we have given a transformation to convert a Dedalus program $\mathcal{P}$ to a pure Datalog$^\neg$ program $pure(\mathcal{P})$ that contains extra rules to enforce causality on message sending in every stable model. In this section, we remove these causality rules and explain how stable models can now represent non-causal message sending.

5.2.1. Non-Causal Transformation

Let $\mathcal{P}$ be a Dedalus program. To model non-causality, we describe the *SZ-transformation* to transform $\mathcal{P}$ into $pure_{\mathrm{SZ}}(\mathcal{P})$, which is obtained in a manner very much like $pure(\mathcal{P})$, the only crucial difference being that the use of relation `before` is completely removed.[14] This is detailed below.

We again assume that each rule of $\mathcal{P}$ has at least one positive body atom. The deductive and inductive rules of $\mathcal{P}$ are transformed just as in $pure(\mathcal{P})$. For each asynchronous rule $R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}$ in $\mathcal{P}$, letting `x`, `s` and `t` be new variables, the old rule transformation (5.5) is modified to become:

$$\mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}^{\Uparrow \mathtt{x},\mathtt{s}},\ \mathtt{all}(\mathtt{y}),\ \mathtt{time}(\mathtt{t}). \tag{5.10}$$

Note, we have simply omitted relation `before`. Also, we retain rule transformations (5.6), (5.7), and (5.8). We omit rule transformations (5.1), (5.2), and (5.9).

To illustrate, applying the SZ-transformation to the Dedalus program in Figure 1 gives the pure Datalog$^\neg$ program obtained from Figure 2 by removing relation `before`: we omit literal $\neg\mathtt{before}(\mathtt{y}, \mathtt{t}, \mathtt{x}, \mathtt{s})$ from the rule for $\mathtt{cand}_T$, and we omit the rules with head predicate `before`.

5.2.2. Input and Stable Models

The semantics for $pure_{\mathrm{SZ}}(\mathcal{P})$ is the same as for $pure(\mathcal{P})$, but we repeat this for clarity. Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$. We give $pure_{\mathrm{SZ}}(\mathcal{P})$ the input $decl(H)$, as defined in Section 5.1.3.

Recall the local finiteness constraint from Section 5.1.3. Now, we call any locally finite stable model M of $pure_{\mathrm{SZ}}(\mathcal{P})$ on $decl(H)$ an *SZ-model* of $\mathcal{P}$ on input H. Program $pure_{\mathrm{SZ}}(\mathcal{P})$ does not enforce causality on the messages in M because the arrival timestamps can be chosen arbitrarily, even into the past.

[14] The non-causal semantics for asynchronous rules is thus completely specified by the technical construct due to Saccà and Zaniolo [24]. This explains the subscript "SZ".

But causality could be respected in some models. In fact, $\mathcal{P}$ has at least one causal SZ-model on every input. This is because $\mathcal{P}$ has at least one run on every input (possibly with only heartbeats), and because each run can be naturally encoded into an SZ-model: we take the trace of the run united with message sending and arrival events represented as $\texttt{cand}_R$-, $\texttt{chosen}_R$- and $\texttt{other}_R$-facts.

Remark (Local finiteness)**.** Note, we have again assumed the local finiteness constraint. This property plays a crucial role in the proof technique of our main result (see Section 6). Essentially, when a stable model of $pure_{\mathrm{SZ}}(\mathcal{P})$ satisfies the constraint, the model has no "sinks" where an infinite number of arriving messages are collected without contributing to computations on other local timestamps. From this viewpoint, the constraint represents a notion of "progress" by the overall computation expressed by the stable model. If viewed from the crash recovery scenario of the Introduction, the constraint expresses that no node can crash infinitely often on the same timestamp; hence, the node has to process only a finite message log during each recovery.

Remark (Well-formedness)**.** Again, it can be shown that all SZ-models are well-formed in the sense of Section 5.1.3.

5.2.3. Output and Tolerating Non-Causality

Let M be an SZ-model of $\mathcal{P}$ on an input H, over a network $\mathcal{N}$. The *output* of M, denoted $output(M)$, is defined with ultimate facts like in the operational semantics (see Section 3.6):

$$output(M) = \bigcup_{R^{(k)} \in out(\mathcal{P})} \{R(\bar{a}) \mid \exists x \in \mathcal{N},\, \exists s \in \mathbb{N},\, \forall t \in \mathbb{N} :\, t \geq s \Rightarrow R(x, t, \bar{a}) \in M\}.$$

Now, we say that an already consistent Dedalus program $\mathcal{P}$ *tolerates non-causality* if individually for every input H, every SZ-model M yields the output $outInst(\mathcal{P}, H)$. Intuitively, if a consistent program tolerates non-causality, then it also computes the same result when messages can be sent into the past.

6. Results

We have recalled the original CRON conjecture in Section 5. We now consider a semantical and syntactical interpretation of this conjecture, and we present the results thereof.

6.1. Semantical Interpretation

In the first interpretation, we relate causality to the monotonicity of the queries computed by Dedalus programs. Recall from Section 4.1 how Dedalus programs can compute queries.

In this context, we have looked at the following formalization of the CRON conjecture:

CRON Conjecture (Semantical). *A Dedalus program computes a monotone query if and only if it tolerates non-causality.*

Both directions of this conjecture can be refuted by counterexamples, as we do in the following two subsections. So, contrary to the CALM conjecture [3, 33, 13], a formalization of the CRON conjecture that is situated purely on the semantical level does not seem insightful.[15]

6.1.1. If Direction

To refute the if-direction of the semantical CRON conjecture, we give a Dedalus program tolerating non-causality that computes a non-monotone query.

Figure 3 gives a Dedalus program to compute the non-monotone emptiness query on a nullary relation S, that is, output "true" (encoded by a nullary relation T) if and only if S is empty (on all nodes). The

[15] For completeness, we should mention that depending on the formalization, the CALM conjecture either holds [13] or does not hold [33].

$\texttt{empty(x)} \mid \texttt{y} \leftarrow \neg S(\,), \texttt{Id(x)}, \texttt{Node(y)}.$
$\texttt{empty(y)}\bullet \leftarrow \texttt{empty(y)}.$
$\texttt{missing}(\,) \leftarrow \texttt{Node(y)}, \neg\texttt{empty(y)}.$
$T(\,) \leftarrow \texttt{Id(x)}, \neg\texttt{missing}(\,).$

Figure 3: Program for emptiness query

$A(\,) \mid \texttt{x} \leftarrow S(\,), \texttt{Id(x)}.$
$A(\,)\bullet \leftarrow A(\,).$
$B(\,) \mid \texttt{x} \leftarrow A(\,), \neg\texttt{sent}_B(\,), \texttt{Id(x)}.$
$\texttt{sent}_B(\,)\bullet \leftarrow A(\,).$
$T(\,) \leftarrow A(\,), B(\,).$
$T(\,)\bullet \leftarrow T(\,).$

Figure 4: Program for non-emptiness query

asynchronous rule lets each node broadcast its own identifier if its relation S is empty. The inductive rule lets a node remember all received node identifiers. The deductive rules let a node output $T(\,)$ starting at the moment that it has all identifiers (including its own).[16] The program is consistent.

Now we consider the tolerance to non-causality. Intuitively, in an SZ-model for this program, even if messages are sent into the past, the inductive rule persists any received identifier towards the future. If S is empty on all nodes, each node x still has a timestamp s after which it has all node identifiers. Then, for all timestamps $t \geq s$, the deductive rules at x will no longer derive $\texttt{missing}(\,)$ and instead derive $T(\,)$. Thus every SZ-model yields the output $T(\,)$ if and only if all nodes have an empty relation S. So, the program tolerates non-causality. A formal proof can be found in Appendix C.1.

6.1.2. Only-If Direction

To refute the only-if direction of the semantical CRON conjecture, we give a Dedalus program computing a monotone query that does not tolerate non-causality.

Figure 4 gives a (contrived) Dedalus program to compute the monotone non-emptiness query on a nullary relation S, that is, output "true" if and only if S is not empty (on at least one node). In the program, a node with nonempty relation S sends $A(\,)$ to itself. On receipt of $A(\,)$, the node stores $A(\,)$ and sends $B(\,)$ to itself if it has not previously done so. Thus, if a node sends $A(\,)$ then it sends $B(\,)$ precisely once. When the $B(\,)$ is later received, it is paired with the stored $A(\,)$, producing the fact $T(\,)$ that is stored by an inductive rule. The program is consistent.

However, the program does not tolerate non-causality, which we now explain. Let H be the input over singleton network $\{z\}$ with $H(z) = \{S(\,)\}$. On input H, we can exhibit an SZ-model M in which $A(\,)$-facts arrive at node z starting at timestamp 1, which implies that $\texttt{sent}_B(\,)$ will exist starting at timestamp 2. This implies that $B(\,)$ is sent precisely once in M, namely, at timestamp 1. Now, the trick is to violate the causal dependency between relations A and B, by letting $B(\,)$ arrive in the past, at timestamp 0 of z, which is before any $A(\,)$ is received. Then the arriving $B(\,)$ cannot pair with any stored or arriving $A(\,)$. Since $B(\,)$ itself is not stored, we have thus erased the single chance of producing $T(\,)$. Hence $\mathit{output}(M) = \emptyset$, and the program does not tolerate non-causality. Formal details can be found in Appendix C.2.

[16] The atom $\texttt{Id(x)}$ in the rule for relation T is to satisfy the assumption that every rule has at least one positive body atom (cf. Section 5.2.1).

$A(\,) \mid \mathtt{x} \leftarrow \mathtt{Id}(\mathtt{x}).$
$B(\,) \mid \mathtt{x} \leftarrow \mathtt{Id}(\mathtt{x}).$
$T(\,) \leftarrow A(\,),\ B(\,).$
$T(\,)\bullet \leftarrow T(\,).$

Figure 5: Positive but not consistent

6.2. Syntactical Interpretation

Now we look at the CRON conjecture from a more syntactical point of view. A Dedalus program without negation is called *positive*. Our main result is that the following does hold:

Theorem 6.1. *Every positive consistent Dedalus program tolerates non-causality.*

The converse direction of Theorem 6.1, to the effect that every consistent Dedalus program tolerating non-causality is equivalent to a positive program, cannot hold by our counterexample for the if-direction of the semantical CRON conjecture (see Section 6.1.1).

It is worth noting that Theorem 6.1 is not a purely syntactical interpretation of the CRON conjecture. First, a positive program is not automatically consistent; Figure 5 gives a simple example.[17] This example is inspired by the work of Marczak et al. [29]: in any fair run, a node will send $A(\,)$ and $B(\,)$ to itself during every transition, but the output $T(\,)$ is only created when we deliver $A(\,)$ and $B(\,)$ simultaneously; some fair runs never do this. Second, consistency of Dedalus programs is undecidable in general, as remarked in Section 3.6.

The following subsections prove Theorem 6.1. In particular, we have to show for each positive consistent Dedalus program $\mathcal{P}$, and each input H, that every SZ-model of $\mathcal{P}$ on H produces *(i)* at least $\mathit{outInst}(\mathcal{P}, H)$ and *(ii)* at most $\mathit{outInst}(\mathcal{P}, H)$, respectively shown in Sections 6.2.1 and 6.2.2.

6.2.1. At Least All Operational Outputs

Let $\mathcal{P}$ be a positive and consistent Dedalus program. Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$, and let M be an SZ-model of $\mathcal{P}$ on H. We show $\mathit{outInst}(\mathcal{P}, H) \subseteq \mathit{output}(M)$. We construct a fair run $\mathcal{R}$ of $\mathcal{P}$ on H such that $\mathit{output}(\mathcal{R}) \subseteq \mathit{output}(M)$. Then, since $\mathit{output}(\mathcal{R}) = \mathit{outInst}(\mathcal{P}, H)$ by consistency of $\mathcal{P}$, we have $\mathit{outInst}(\mathcal{P}, H) \subseteq \mathit{output}(M)$, as desired.

Notations. We need some auxiliary notations. For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $\mathit{all}_M(x, s)$ be the set of facts $R(\bar{a})$ for which $R(x, s, \bar{a}) \in M|_{\mathit{sch}(\mathcal{P})}$, i.e., the set of all facts over $\mathit{sch}(\mathcal{P})$ in M at node x on timestamp s.

For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $\mathit{rcv}_M(x, s)$ be the set of facts $R(\bar{a})$ for which there is some y and t such that $\mathtt{chosen}_R(y, t, x, s, \bar{a}) \in M$, i.e., the set of all messages arriving at (x, s) in M. Note, $\mathit{rcv}_M(x, s) \subseteq \mathit{all}_M(x, s)$ by rules of the form (5.8) in $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$.

For each $x \in \mathcal{N}$, let $\mathit{snd}_M(x)$ be the set of pairs $(y, R(\bar{a}))$ for which there is some s and t such that $\mathtt{chosen}_R(x, s, y, t, \bar{a}) \in M$, i.e., the set of all messages (with addressee) that x ever sends in M.

We define $\mathit{sndFin}_M(x) \subseteq \mathit{snd}_M(x)$ to be the subset of pairs $(y, R(\bar{a}))$ for which there are only a finite number of times s such that $\mathtt{chosen}_R(x, s, y, t, \bar{a}) \in M$ for some $t \in \mathbb{N}$, i.e., there are only a finite number of times s on which x sends $R(\bar{a})$ to y in M. Now, for each $x \in \mathcal{N}$, we define $\mathit{start}_M(x) = 0$ if $\mathit{sndFin}_M(x) = \emptyset$ and otherwise we define $\mathit{start}_M(x)$ to be 1 plus the largest timestamp on which x sends a pair of $\mathit{sndFin}_M(x)$ in M. Intuitively, $\mathit{start}_M(x)$ is the first local timestamp of x at which x no longer sends messages in $\mathit{sndFin}_M(x)$, so the messages that x sends starting from $\mathit{start}_M(x)$ are sent infinitely often.

[17] Relation `Id` is from Section 4.1.

Main idea. We construct $\mathcal{R}$ as follows. Assuming some arbitrary order on $\mathcal{N}$, consider the following (co-lexical) total order $\leq$ on $\mathcal{N} \times \mathbb{N}$:

$$(x, s) \leq (y, t) \iff s < t \text{ or } (s = t \text{ and } x \leq y).$$

For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $ord(x, s)$ denote the ordinal of (x, s) in this total order. For each transition index i of $\mathcal{R}$, we define the active node, denoted x_i, to be the unique node $x \in \mathcal{N}$ that satisfies $ord(x, s) = i$ for some $s \in \mathbb{N}$.

Although the actual message deliveries of $\mathcal{R}$ are defined later, we can already give the approach. We define for each transition index $i \in \mathbb{N}$ the *arrival function* $\alpha_{\mathcal{R}}^{(i)}$ that contains for each transition j with $j \leq i$ mappings of the form $(j, y, R(\bar{a})) \mapsto k$, where $R(\bar{a})$ is a message with addressee y sent in transition j, to say that $R(\bar{a})$ is delivered to y in transition k (with $j < k$), which implies $x_k = y$.[18] In particular, letting $\alpha_{\mathcal{R}}^{(-1)} = \emptyset$, we define $\alpha_{\mathcal{R}}^{(i)}$ as an extension of $\alpha_{\mathcal{R}}^{(i-1)}$ to choose arrival transitions for the newly sent messages during transition i. The message deliveries of the entire run $\mathcal{R}$ are thus defined by $\bigcup_{i \in \mathbb{N}} \alpha_{\mathcal{R}}^{(i)}$. As a small remark, we denote mappings $(j, y, R(\bar{a})) \mapsto k$ simply as $(j, y, R(\bar{a}), k)$.

We now give properties that we want the message deliveries to satisfy. For each $i \in \mathbb{N}$, we write D_i, x_i and s_i to denote respectively the deductive fixpoint, active node and timestamp (of the active node) during transition i. For each $i \in \mathbb{N}$, we want the following properties to be satisfied, for which the intuition is provided below:

$$D_i \subseteq all_M(x_i, s_i) \tag{6.1}$$

$$\forall (j, y, \boldsymbol{f}, k) \in \alpha_{\mathcal{R}}^{(i)} : \boldsymbol{f} \in rcv_M(x_k, s_k) \tag{6.2}$$

$$\forall (j, y, \boldsymbol{f}, k) \in \alpha_{\mathcal{R}}^{(i)} : s_k \geq start_M(y) \tag{6.3}$$

Property (6.1) ensures all ultimate facts of $\mathcal{R}$ are ultimate facts of M, resulting in $output(\mathcal{R}) \subseteq output(M)$, as desired. Property (6.2) ensures we do not have more opportunities in $\mathcal{R}$ for messages to arrive "together" when compared to M, so that induction property (6.1) can be satisfied. To explain property (6.3), note that some messages in M are sent only a finite number of times, even into the past. Such messages are the result of a coincidence, like the coincident arrival of messages, and because such messages can not be sent into the past in $\mathcal{R}$, we would have to deliver them somewhere in the future, risking a violation of induction property (6.2). Now, induction property (6.3) will ensure that we only send messages in $\mathcal{R}$ that are sent an infinite number of times in M, and this can be used to satisfy induction property (6.2).

We now define the message deliveries of $\mathcal{R}$, i.e., the arrival functions, by induction on the transitions.

Inductive construction. For uniformity, we start with $i = -1$, and define $\alpha_{\mathcal{R}}^{(-1)} = \emptyset$ and $D_{-1} = \emptyset$. So, properties (6.1) through (6.3) are trivially satisfied for $i = -1$.

For the induction hypothesis, we assume $\mathcal{R}$ has been partially constructed up to and including transition $i - 1$, where $i \geq 0$, and we assume the properties hold for all transitions $j = -1, 0, \ldots, i - 1$. For the inductive step, we show that property (6.1) is satisfied for i, and we show how to extend $\alpha_{\mathcal{R}}^{(i-1)}$ to $\alpha_{\mathcal{R}}^{(i)}$ such that properties (6.2) and (6.3) are satisfied. First, the set m_i of (tagged) messages to be delivered in transition i consists of all pairs $(j, \boldsymbol{f})$ for which $\alpha_{\mathcal{R}}^{(i-1)}$ contains $(j, x_i, \boldsymbol{f}, i)$, i.e., all messages scheduled by $\alpha_{\mathcal{R}}^{(i-1)}$ to be delivered in transition i.[19]

Property (6.1) We show $D_i \subseteq all_M(x_i, s_i)$. Let $\rho_i = (st_i, bf_i)$ denote the source-configuration of transition i. By definition, $D_i = deduc_{\mathcal{P}}(st_i(x_i) \cup untag(m_i))$. Now, by Claim 6.2 (below), it suffices to show $st_i(x_i) \cup untag(m_i) \subseteq all_M(x_i, s_i)$.

First, we know $untag(m_i) \subseteq rcv_M(x_i, s_i) \subseteq all_M(x_i, s_i)$ by applying the induction hypothesis for property (6.2) to $\alpha_{\mathcal{R}}^{(i-1)}$.

[18] To satisfy fairness (see Section 3.5.3), all messages sent in transitions $j \leq i$ will get a mapping in $\alpha_{\mathcal{R}}^{(i)}$.

[19] Note, the design of $\alpha_{\mathcal{R}}^{(i-1)}$ implies for each $(j, y, \boldsymbol{f}, i) \in \alpha_{\mathcal{R}}^{(i-1)}$ that $y = x_i$.

We are left to show $st_i(x_i) \subseteq all_M(x_i, s_i)$. We have $st_i(x_i)|_{edb(\mathcal{P})} \subseteq all_M(x_i, s_i)$ because $st_i(x_i)|_{edb(\mathcal{P})} = H(x_i)$ by the operational semantics and $H(x_i)^{\Uparrow x_i, s_i} \subseteq decl(H) \subseteq M$ by definition of M. Next, if i is the first transition of x_i, we have $st_i(x_i)|_{idb(\mathcal{P})} = \emptyset \subseteq all_M(x_i, s_i)$. Otherwise, we consider the last transition j of $\mathcal{R}$ before i in which x_i was also the active node. By the operational semantics, $st_i(x_i)|_{idb(\mathcal{P})} = induc_{\mathcal{P}}(D_j)$. Because $D_j \subseteq all_M(x_i, s_j)$ by the induction hypothesis for property (6.1), Claim 6.3 (below) gives $induc_{\mathcal{P}}(D_j) \subseteq all_M(x_i, s_j + 1) = all_M(x_i, s_i)$, as desired.

Claim 6.2. Let $(x, s) \in \mathcal{N} \times \mathbb{N}$ and let I be an instance over $sch(\mathcal{P})$. If $I \subseteq all_M(x, s)$ then $deduc_{\mathcal{P}}(I) \subseteq all_M(x, s)$. (Shown in Appendix D.)

Claim 6.3. Let $(x, s) \in \mathcal{N} \times \mathbb{N}$ and let D be an instance over $sch(\mathcal{P})$. If $D \subseteq all_M(x, s)$ then $induc_{\mathcal{P}}(D) \subseteq all_M(x, s + 1)$. (Shown in Appendix D.)

Properties (6.2) and (6.3) We extend $\alpha_{\mathcal{R}}^{(i-1)}$ to $\alpha_{\mathcal{R}}^{(i)}$ so that properties (6.2) and (6.3) are satisfied. Suppose node x_i during transition i sends a message $R(\bar{a})$ to an addressee $y \in \mathcal{N}$. We have to choose a transition k with $i < k$ in which to deliver $R(\bar{a})$ to y. We start by showing there are an infinite number of timestamps s on which x_i sends $R(\bar{a})$ to y in M. We differentiate between two cases.

First, suppose $s_i < start_M(x_i)$. The induction hypothesis for property (6.3) implies x_i has only done heartbeats up to and including transition i, i.e., no messages have been delivered to x_i yet. Then by Claim 6.4 (below), node x_i sends $R(\bar{a})$ to y during an infinite number of timestamps in M.

Now suppose $s_i \geq start_M(x_i)$. We know $D_i \subseteq all_M(x_i, s_i)$ (shown above), $R(y, \bar{a}) \in async_{\mathcal{P}}(D_i)$, and $y \in \mathcal{N}$; hence, by Claim 6.5 (below), there is a $t \in \mathbb{N}$ for which $\texttt{chosen}_R(x_i, s_i, y, t, \bar{a}) \in M$. So, in M, node x_i sends $R(\bar{a})$ to y during a timestamp that is at least $start_M(x_i)$, which, by definition of $start_M(x_i)$, implies that node x_i sends $R(\bar{a})$ to y during an infinite number of timestamps in M.

Now, because x_i sends $R(\bar{a})$ to y during an infinite number of timestamps in M, and y receives only a finite number of messages on each timestamp by the local finiteness assumption (Section 5.2.2), there must be an infinite number of timestamps on which y receives $R(\bar{a})$ from x_i in M. Among these arrival timestamps, we can surely choose some timestamp $t \in \mathbb{N}$ for which $ord(y, t) > i$ and $t \geq start_M(y)$. Next, we extend $\alpha_{\mathcal{R}}^{(i-1)}$ by adding the mapping $(i, y, R(\bar{a}), k)$ where $k = ord(y, t)$. Note, $s_k = t$ by design of the above total order on $\mathcal{N} \times \mathbb{N}$. So, by choice of t, this added mapping satisfies properties (6.2) and (6.3).

Claim 6.4. Let S be the set of transition ordinals of $\mathcal{R}$ up to and including i in which x_i is the active node. Suppose all transitions in S are heartbeat transitions. Let $R(\bar{a})$ be a message that x_i sends during transition i to a node $y \in \mathcal{N}$. In M, the number of timestamps on which x_i sends $R(\bar{a})$ to y is infinite. (Shown in Appendix D.)

Claim 6.5. Let $(x, s) \in \mathcal{N} \times \mathbb{N}$ and let D be an instance over $sch(\mathcal{P})$. Suppose $D \subseteq all_M(x, s)$. For each $R(y, \bar{a}) \in async_{\mathcal{P}}(D)$ with $y \in \mathcal{N}$ there exists a value t such that $\texttt{chosen}_R(x, s, y, t, \bar{a}) \in M$. (Shown in Appendix D.)

6.2.2. No Wrong Outputs

Let $\mathcal{P}$ be a positive and consistent Dedalus program. Let H be an input for $\mathcal{P}$, and let M be an SZ-model of $\mathcal{P}$ on H. We show $output(M) \subseteq outInst(\mathcal{P}, H)$. We construct a fair run $\mathcal{R}$ of $\mathcal{P}$ on H such that $output(M) \subseteq output(\mathcal{R})$. Then, using $output(\mathcal{R}) = outInst(\mathcal{P}, H)$ by consistency of $\mathcal{P}$, we obtain $output(M) \subseteq outInst(\mathcal{P}, H)$, as desired.

Run $\mathcal{R}$ proceeds in rounds: in each round we let each node become active precisely once to receive its entire buffer at the beginning of the round. Messages sent in each round are accumulated and are delivered only during the next round. The number of rounds is infinite. Because $\mathcal{P}$ is positive, the programs $deduc_{\mathcal{P}}$, $induc_{\mathcal{P}}$, and $async_{\mathcal{P}}$ are monotone. Then, since always the entire buffer is delivered to each node, the set of deductively derived facts monotonously grows over the steps of a node.

For each transition i of $\mathcal{R}$, let D_i denote the output of $deduc_{\mathcal{P}}$ during i. For each fact $R(x, s, \bar{a}) \in M|_{sch(\mathcal{P})}$, we show there is a transition i of x in $\mathcal{R}$ with $R(\bar{a}) \in D_i$. This gives $output(M) \subseteq output(\mathcal{R})$ because for each ultimate fact $R(\bar{a})$ in M at some node x, surely $R(x, s, \bar{a}) \in M$ for some $s \in \mathbb{N}$, and so if

$R(\bar{a}) \in D_i$ for some transition i of x then $R(\bar{a}) \in D_j$ for all subsequent transitions j of x by the monotonous nature of $\mathcal{R}$.

Abbreviate $G_M(\mathcal{P}) = \mathit{ground}_M(\mathcal{P}', I)$ where $\mathcal{P}' = \mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ and $I = \mathit{decl}(H)$. Because $M = G_M(\mathcal{P})(I)$ by definition of stable model, we can consider the infinite sequence M_0, M_1, M_2, ..., such that $M = \bigcup_l M_l$; $M_0 = I$; and, for each $l \geq 1$ the instance M_l is obtained from M_{l-1} by applying the immediate consequence operator of $G_M(\mathcal{P})$. This implies $M_{l-1} \subseteq M_l$ for each $l \geq 1$. By induction on l, we show that for each $R(x, s, \bar{a}) \in M_l|_{sch(\mathcal{P})}$ there is a transition i of x in $\mathcal{R}$ with $R(\bar{a}) \in D_i$.

For the base case, $R(x, s, \bar{a}) \in M_0|_{sch(\mathcal{P})}$ implies $R(\bar{a}) \in H(x)$. Then $R(\bar{a}) \in D_i$ for any transition i of x because each state of x contains $H(x)$ by the operational semantics. For the induction hypothesis, assume the property holds for M_{l-1} where $l \geq 1$. Now, let $R(x, s, \bar{a}) \in M_l|_{sch(\mathcal{P})} \setminus M_{l-1}$. Let $\psi \in G_M(\mathcal{P})$ be a ground rule responsible for deriving this fact, i.e., $\mathit{pos}_\psi \subseteq M_{l-1}$ and $\mathit{head}_\psi = R(x, s, \bar{a})$. Rule ψ must have one of the following three forms: the deductive form (5.3), the inductive form (5.4), or the delivery form (5.8). We handle each case in turn.

Deductive. Let $\varphi \in \mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ be the rule corresponding to ψ, so φ is of the form (5.3). Let V be the valuation for φ such that ψ results from applying V to φ. In turn, let $\varphi' \in \mathcal{P}$ be the original deductive rule on which φ is based. Note, $\varphi' \in \mathit{deduc}_\mathcal{P}$. By the syntactical correspondence between φ and φ', we can apply V to φ'. Now, it suffices to show $V(\mathit{pos}_{\varphi'}) \subseteq D_i$ for some transition i of x in $\mathcal{R}$, resulting in $V(\mathit{head}_{\varphi'}) = R(\bar{a}) \in D_i$ by the fixpoint semantics of $\mathit{deduc}_\mathcal{P}$, as desired.

Let $S(\bar{b}) \in V(\mathit{pos}_{\varphi'})$. By the syntactical correspondence between φ' and φ, we have $S(x, s, \bar{b}) \in V(\mathit{pos}_\varphi) = \mathit{pos}_\psi$. Using $\mathit{pos}_\psi \subseteq M_{l-1}$ gives $S(x, s, \bar{b}) \in M_{l-1}|_{sch(\mathcal{P})}$. Then the induction hypothesis implies there is a transition j of x in $\mathcal{R}$ satisfying $S(\bar{b}) \in D_j$. And because deductive facts monotonously grow at x in $\mathcal{R}$, there is a transition i of x such that $S(\bar{b}) \in D_i$ for each $S(\bar{b}) \in V(\mathit{pos}_{\varphi'})$.

Inductive. Let φ and V be like in the deductive case, but now φ is of the form (5.4). Let $\varphi' \in \mathit{induc}_\mathcal{P}$ be the rule corresponding to φ. Again, we can apply V to φ', and it suffices to show $V(\mathit{pos}_{\varphi'}) \subseteq D_i$ for some transition i of x in $\mathcal{R}$, causing $V(\mathit{head}_{\varphi'}) = R(\bar{a})$ to be stored in the next state of x. Then, with j being the first transition of x after i, we obtain $R(\bar{a}) \in D_j$ by the operational semantics, as desired. The existence of i is established similarly to the deductive case.

Delivery. Rule ψ is of the form (5.8), having a body fact $\mathtt{chosen}_R(y, t, x, s, \bar{a}) \in M_{l-1}$ with $(y, t) \in \mathcal{N} \times \mathbb{N}$. We show there is a transition i of y in $\mathcal{R}$, in which y sends $R(\bar{a})$ to x. Then, in the next round of $\mathcal{R}$ following i, we deliver $R(\bar{a})$ to x in some transition j. Then $R(\bar{a}) \in D_j$ by the operational semantics, as desired.

Now, because $\mathtt{chosen}_R(y, t, x, s, \bar{a}) \in M_{l-1}$, there is some $k \in \mathbb{N}$ with $0 < k < l - 1$ such that $\mathtt{cand}_R(y, t, x, s, \bar{a}) \in M_k \setminus M_{k-1}$. Let $\psi' \in G_M(\mathcal{P})$ be a rule responsible for deriving the $\mathtt{cand}_R$-fact. Let $\varphi' \in \mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ be the rule corresponding to ψ', and let V' be the valuation for φ' giving rise to ψ'. In turn, let $\varphi'' \in \mathit{async}_\mathcal{P}$ be the rule corresponding to φ'. By the syntactical correspondence between φ' and φ'', we can apply V' to φ''. Note, $V'(\mathit{head}_{\varphi''}) = R(x, \bar{a})$. To make y send $R(\bar{a})$ to x in some transition i, we need $V'(\mathit{pos}_{\varphi''}) \subseteq D_i$. The existence of transition i is again established like in the deductive case.

7. Discussion

In this paper, we have presented an initial investigation of the CRON conjecture by Hellerstein. For the formalization, we have used the language Dedalus, a language that has inspired multiple other languages in the field of declarative networking. We have confirmed that positive Dedalus programs tolerate non-causality if they already behave correctly in the operational semantics, where non-causality means that messages can be sent into the past. Also, to better understand its fundamental properties, we have argued that Dedalus captures the While queries and has PSPACE data complexity.

In future work, the spectrum of causality needs to be better understood. Indeed, besides positive programs, perhaps richer classes of programs can tolerate some relaxations of causality as well. Also, it might be useful to link the CRON conjecture more concretely to crash recovery mechanisms, or other application scenarios.

As argued in the related work, the inductive rules of Dedalus have a strong connection to temporal deductive databases and temporal logic programming [14, 15, 16, 21]. It appears interesting to further investigate how the asynchronous rules of Dedalus relate to this prior work, because their semantics is challenging to represent and reason about. Prior work, like the language Datalog LITE [34], might also reveal the existence of Dedalus fragments with PTIME data complexity and characterize the expressive power of such fragments.

In practice, there might be a need for finite representations of stable models for Dedalus programs, for both the causal declarative semantics and the non-causal declarative semantics. Such representations could be used for simulations and testing purposes. The work of Baudinet et al. [35] provides strong indications that only periodic phenomena can be finitely represented (in a logic programming framework). In our semantics for Dedalus, however, asynchronous rules do not observe this restriction, since messages can be arbitrarily delayed. Hence, it remains a challenge to find finite representations for such cases. A possible idea could be to focus on just the ultimate facts. Alternatively, asynchronous rules could be given a more restricted semantics, where the delay on messages is limited, and such that this semantics still corresponds with practical scenarios.

Lastly, it might also be useful to investigate alternative semantics for Dedalus, for example by adopting some Datalog extensions proposed by Abiteboul and Vianu [32], and see if the results of this paper can be extended to such frameworks.

References

[1] T. Ameloot, J. Van den Bussche, On the CRON conjecture, in: P. Barceló, R. Pichler (Eds.), Datalog in Academia and Industry, volume 7494 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 44–55.

[2] B. Loo, et al., Declarative networking, Commun. ACM 52 (2009) 87–95.

[3] J. Hellerstein, The declarative imperative: experiences and conjectures in distributed logic, SIGMOD Record 39 (2010) 5–19.

[4] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, Journal of Internet Services and Applications 1 (2010) 7–18.

[5] H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations, and Advanced Topics, Wiley, 2004.

[6] P. Alvaro, W. Marczak, et al., Dedalus: Datalog in time and space, Technical Report EECS-2009-173, University of California, Berkeley, 2009.

[7] P. Alvaro, W. Marczak, et al., Dedalus: Datalog in time and space, in: O. de Moor, G. Gottlob, T. Furche, A. Sellers (Eds.), Datalog Reloaded: First International Workshop, Datalog 2010, volume 6702 of *Lecture Notes in Computer Science*, 2011, pp. 262–281.

[8] P. Alvaro, T. Ameloot, J. Hellerstein, W. Marczak, J. Van den Bussche, A Declarative Semantics for Dedalus, Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, 2011. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-120.html`.

[9] P. Alvaro, T. Ameloot, J. Hellerstein, W. Marczak, J. Van den Bussche, A Declarative Semantics for Dedalus, Technical Report, Hasselt University, 2013. URL: `http://hdl.handle.net/1942/14572`.

[10] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the Fifth International Conference on Logic Programming, MIT Press, 1988, pp. 1070–1080.

[11] J. Lobo, J. Ma, A. Russo, F. Le, Declarative distributed computing, in: E. Erdem, J. Lee, Y. Lierler, D. Pearce (Eds.), Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz, volume 7265 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 454–470.

[12] M. Interlandi, L. Tanca, S. Bergamaschi, Datalog in time and space, synchronously, in: 7th Alberto Mendelzon International Workshop on Foundations of Data Management, 2013.

[13] T. Ameloot, F. Neven, J. Van den Bussche, Relational transducers for declarative networking, Journal of the ACM 60 (2013) 15:1–15:38.

[14] C. Zaniolo, N. Arni, K. Ong, Negation and aggregates in recursive rules: the ldl++ approach, in: S. Ceri, K. Tanaka, S. Tsur (Eds.), Deductive and Object-Oriented Databases, volume 760 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1993, pp. 204–221.

[15] G. Lausen, B. Ludäscher, W. May, On active deductive databases: The statelog approach, in: B. Freitag, H. Decker, M. Kifer, A. Voronkov (Eds.), Transactions and Change in Logic Databases, volume 1472 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1998, pp. 69–106.

[16] L. Lu, J. Cleary, An operational semantics of starlog, in: G. Nadathur (Ed.), Principles and Practice of Declarative Programming, volume 1702 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1999, pp. 294–310.

[17] J. Chomicki, T. Imieliński, Finite representation of infinite query answers, ACM Transactions on Database Systems 18 (1993) 181–223.

[18] J. Chomicki, Depth-bounded bottom-up evaluation of logic programs, The Journal of Logic Programming 25 (1995) 1–31.

[19] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres, A logic programming approach to knowledge-state planning: Semantics and complexity, ACM Transactions on Computational Logic 5 (2004) 206–263.
[20] G. Greco, A. Guzzo, D. Saccà, F. Scarcello, Event choice datalog: a logic programming language for reasoning in multiple dimensions, in: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP, ACM Press, 2004, pp. 238–249.
[21] C. Nomikos, P. Rondogiannis, M. Gergatsoulis, Temporal stratification tests for linear and branching-time deductive databases, Theoretical Computer Science 342 (2005) 382–415.
[22] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[23] M. Vardi, The complexity of relational query languages, in: Proceedings 14th ACM Symposium on the Theory of Computing, 1982, pp. 137–146.
[24] D. Saccà, C. Zaniolo, Stable models and non-determinism in logic programs with negation, in: Proceedings of the Ninth ACM Symposium on Principles of Database Systems, ACM Press, 1990, pp. 205–217.
[25] S. Abiteboul, M. Bienvenu, A. Galland, et al., A rule-based language for Web data management, in: Proceedings 30th ACM Symposium on Principles of Database Systems, ACM Press, 2011, pp. 293–304.
[26] N. Francez, Fairness, Springer-Verlag New York, Inc., New York, NY, USA, 1986.
[27] K. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming, Distributed Computing 2 (1988) 226–241.
[28] L. Lamport, Fairness and hyperfairness, Distributed Computing 13 (2000) 239–245.
[29] W. Marczak, P. Alvaro, N. Conway, J. Hellerstein, D. Maier, Confluence Analysis for Distributed Programs: A Model-Theoretic Approach, Technical Report UCB/EECS-2011-154, EECS Department, University of California, Berkeley, 2011. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-154.html`.
[30] T. Ameloot, J. Van den Bussche, Deciding eventual consistency for a simple class of relational transducer networks, in: Proceedings of the 15th International Conference on Database Theory, ACM Press, 2012, pp. 86–98.
[31] S. Abiteboul, E. Simon, Fundamental properties of deterministic and nondeterministic extensions of Datalog, Theor. Comput. Sci. 78 (1991) 137–158.
[32] S. Abiteboul, V. Vianu, Datalog extensions for database queries and updates, J. Comput. Syst. Sci. 43 (1991) 62–124.
[33] D. Zinn, T. Green, B. Ludaescher, Win-move is coordination-free (sometimes), in: Proceedings of the 15th International Conference on Database Theory, ACM Press, 2012, pp. 99–113.
[34] G. Gottlob, E. Grädel, H. Veith, Datalog lite: a deductive query language with linear time model checking, ACM Transactions on Computational Logic 3 (2002) 42–79.
[35] M. Baudinet, J. Chomicki, P. Wolper, Temporal deductive databases, in: A. Tansel, et al. (Eds.), Temporal Databases: Theory, Design, and Implementation, Benjamin/Cummings, 1993.

Appendix

Appendix A. Consistency Undecidability

Using similar arguments as in our previous work [13], we show that consistency of Dedalus programs is undecidable. Concretely, we reduce finite satisfiability of FO to inconsistency of Dedalus programs.

Let φ be an FO sentence, over a database schema $\mathcal{D}$. Let S, A, B, and T be relation names not yet used in $\mathcal{D}$. First, there exists a stratified Datalog$^\neg$ program P over $\mathcal{D}$ that simulates φ: program P has a nullary *idb*-relation S such that for each instance I over $\mathcal{D}$, relation S is made nonempty by P if and only if φ is satisfied on I.

Next, we extend P to a Dedalus program that is inconsistent if and only if φ is finitely satisfiable. The idea is that each node of the network is given a local input fragment over $\mathcal{D}$, and by running program P as the deductive rules, the nodes can discover whether φ is satisfied on their input fragment. If so, a node sends messages $A(\,)$ and $B(\,)$ to itself. The output fact $T(\,)$ is produced at a node when $A(\,)$ and $B(\,)$ are delivered simultaneously. Some fair runs never do this, and the program is thus inconsistent. This behavior is achieved with the additional rules shown in Figure A.6, where we assume the existence of relation `Id` from Section 4.1. Note, if φ is not finitely satisfiable, all runs produce the empty output, and the program is consistent.

Appendix B. Complexity Lower Bound

We define a specific Dedalus program $\mathcal{P}$ for which $eval_{\mathcal{P}}$ is PSPACE-hard. Concretely, $\mathcal{P}$ takes as input the description of a (deterministic) PSPACE Turing machine and an input tape, and simulates the Turing machine on this input. We first describe $\mathcal{P}$ and then we argue PSPACE-hardness for $eval_{\mathcal{P}}$.

$$A(\,) \mid \mathtt{x} \leftarrow S(\,),\ \mathtt{Id}(\mathtt{x}).$$
$$B(\,) \mid \mathtt{x} \leftarrow S(\,),\ \mathtt{Id}(\mathtt{x}).$$
$$T(\,) \leftarrow A(\,),\ B(\,).$$
$$T(\,)\bullet \leftarrow T(\,).$$

Figure A.6: Rules for Inconsistency

Appendix B.1. Turing Machine Simulator

Let $\mathbf{l}, \mathbf{r}, \mathbf{s} \in \mathbf{dom}$ be three distinct symbols representing the head movements of a Turing machine, called respectively "left", "right" and "stay". The *edb* relations of $\mathcal{P}$ are as follows:

- $\mathtt{delta}^{(5)}$ containing tuples (q_1, s_1, q_2, s_2, h) expressing that if the Turing machine reads symbol s_1 in state q_1 then it goes to state q_2, writes symbol s_2, and moves the head according to h where $h \in \{\mathbf{l}, \mathbf{r}, \mathbf{s}\}$;
- relations $\mathtt{left}^{(1)}$, $\mathtt{right}^{(1)}$, and $\mathtt{stay}^{(1)}$ to contain respectively $\mathbf{l}$, $\mathbf{r}$, and $\mathbf{s}$;
- relation $\mathtt{initState}^{(1)}$ to represent the initial state;
- relation $\mathtt{initTape}^{(2)}$ to represent the initial tape; and,
- relations $\mathtt{first}^{(1)}$ and $\mathtt{succ}^{(2)}$ to represent tape addresses, where `first` contains the smallest address.

The output relation of $\mathcal{P}$ is the relation $\mathtt{state}^{(1)}$, that represents the state of the simulated Turing machine. The rules of $\mathcal{P}$ are given below, where we also use some auxiliary *idb*-relations: relation $\mathtt{tape}^{(2)}$ represents the current tape; relation $\mathtt{head}^{(1)}$ represent the current location of the head; relation $\mathtt{go}^{(0)}$ helps to separate the first step of $\mathcal{P}$ from subsequent ones; and, relation $\mathtt{fires}^{(5)}$ contains the transition from `delta` that is currently applicable.

$$\mathtt{go}(\,)\bullet \leftarrow .$$

$$\mathtt{state}(\mathtt{q})\bullet \leftarrow\neg\ \mathtt{go}(\,),\ \mathtt{initState}(\mathtt{q}).$$
$$\mathtt{tape}(\mathtt{a},\mathtt{s})\bullet \leftarrow\neg\ \mathtt{go}(\,),\ \mathtt{initTape}(\mathtt{a},\mathtt{s}).$$
$$\mathtt{head}(\mathtt{a})\bullet \leftarrow\neg\ \mathtt{go}(\,),\ \mathtt{first}(\mathtt{a}).$$

$$\mathtt{fires}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}) \leftarrow \mathtt{go}(\,),\ \mathtt{delta}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}),\ \mathtt{state}(\mathtt{q}_1),\ \mathtt{head}(\mathtt{a}),\ \mathtt{tape}(\mathtt{a},\mathtt{s}_1).$$

$$\mathtt{state}(\mathtt{q}_2)\bullet \leftarrow \mathtt{go}(\,),\ \mathtt{fires}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}).$$

$$\mathtt{tape}(\mathtt{a},\mathtt{s}_2)\bullet \leftarrow \mathtt{go}(\,),\ \mathtt{fires}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}),\ \mathtt{head}(\mathtt{a}).$$
$$\mathtt{tape}(\mathtt{a},\mathtt{s})\bullet \leftarrow \mathtt{go}(\,),\ \mathtt{tape}(\mathtt{a},\mathtt{s}),\ \neg\mathtt{head}(\mathtt{a}).$$

$$\mathtt{head}(\mathtt{a})\bullet \leftarrow \mathtt{go}(\,),\ \mathtt{fires}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}),\ \mathtt{stay}(\mathtt{h}),\ \mathtt{head}(\mathtt{a}).$$
$$\mathtt{head}(\mathtt{b})\bullet \leftarrow \mathtt{go}(\,),\ \mathtt{fires}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}),\ \mathtt{left}(\mathtt{h}),\ \mathtt{head}(\mathtt{a}),\ \mathtt{succ}(\mathtt{b},\mathtt{a}).$$
$$\mathtt{head}(\mathtt{b})\bullet \leftarrow \mathtt{go}(\,),\ \mathtt{fires}(\mathtt{q}_1,\mathtt{s}_1,\mathtt{q}_2,\mathtt{s}_2,\mathtt{h}),\ \mathtt{right}(\mathtt{h}),\ \mathtt{head}(\mathtt{a}),\ \mathtt{succ}(\mathtt{a},\mathtt{b}).$$

Note, $\mathcal{P}$ is consistent on each input because there are no asynchronous rules.

Appendix B.2. Hardness

Let A be a problem from PSPACE. We reduce A to $eval_{\mathcal{P}}$ with $\mathcal{P}$ as above.

Let M be a PSPACE Turing machine M for A. Let $k \in \mathbb{N}$ be a number such that on each input string w for A, machine M consumes at most n^k space, where $n = |w|$. We regard M and k as constant during the reduction. We can naturally encode M over *edb* relations `delta`, `left`, `right`, and `stay`. We also create the fact $\mathtt{initState}(q)$ where q is the start state of M.

Now, let w be an input for A. Let $n = |w|$. We encode the sequence of natural numbers $1, \ldots, n^k$ in *edb* relations `first` and `succ`; these are all tape cell addresses that M can visit during its computation on w. For w itself, for each $i = 1, \ldots, n$, we create the fact $\texttt{initTape}(i, w[i])$; and, for each $i = n+1, \ldots, n^k$, we create the fact $\texttt{initTape}(i, b)$, where b is the blank symbol of M.

Let us refer to the set of all facts over $edb(\mathcal{P})$ from above as $enc_{\mathcal{P}}(w)$. Now, to reduce input w for A, we give $eval_{\mathcal{P}}$ *(i)* the facts $enc_{\mathcal{P}}(w)$ placed on a single-node network, and *(ii)* the output fact $\texttt{state}(q_a)$, where q_a is the accept state of M. We assume here that, when q_a is reached, machine M remains in state q_a and no longer moves the head. So, M accepts w if and only if $\texttt{state}(q_a)$ is an ultimate fact of $\mathcal{P}$ on $enc_{\mathcal{P}}(w)$ (on the single-node network). Note, the reduction can be done in PTIME.[20]

Appendix C. Semantical CRON

Before presenting the proof of the counterexamples, we consider the following lemma:

Lemma Appendix C.1. Let $\mathcal{P}$ be a Dedalus program. Let H be an input for $\mathcal{P}$, and let M be an SZ-model of $\mathcal{P}$ on H. For each fact $\texttt{cand}_R(x, s, y, t, \bar{a}) \in M$ there is a value $t' \in \mathbb{N}$ such that $\texttt{chosen}_R(x, s, y, t', \bar{a}) \in M$.

Proof. Abbreviate $G_M(\mathcal{P}) = ground_M(\mathcal{P}', I)$ where $\mathcal{P}' = pure_{\text{SZ}}(\mathcal{P})$ and $I = decl(H)$. Towards a proof by contradiction, suppose there is no such timestamp t'. Consider the following ground rule of the form (5.6), after removing the negative body literal:

$$\texttt{chosen}_R(x, s, y, t, \bar{a}) \leftarrow \texttt{cand}_R(x, s, y, t, \bar{a}).$$

This rule can not be in $G_M(\mathcal{P})$ because otherwise $\texttt{cand}_R(x, s, y, t, \bar{a}) \in M$ implies $\texttt{chosen}_R(x, s, y, t, \bar{a}) \in M$, which is assumed to be false. The absence of the above ground rule from $G_M(\mathcal{P})$ implies $\texttt{other}_R(x, s, y, t, \bar{a}) \in M$. This $\texttt{other}_R$-fact must be derived by a ground rule of the form (5.7):

$$\texttt{other}_R(x, s, y, t, \bar{a}) \leftarrow \texttt{cand}_R(x, s, y, t, \bar{a}),\ \texttt{chosen}_R(x, s, y, t', \bar{a}),\ t \neq t'.$$

So, $\texttt{chosen}_R(x, s, y, t', \bar{a}) \in M$ after all, which is the desired contradiction. □

Appendix C.1. If Direction

Let $\mathcal{Q}$ and $\mathcal{P}$ be respectively the emptiness query and the Dedalus program from Figure 3. We show that $\mathcal{P}$ tolerates non-causality.

Appendix C.1.1. Empty Input

Let H be an input for $\mathcal{P}$ over a network $\mathcal{N}$ that assigns each $x \in \mathcal{N}$ an empty relation S. So, $outInst(\mathcal{P}, H) = \{T(\,)\}$. Let M be an SZ-model of $\mathcal{P}$ on H. We have to show that $output(M) = \{T(\,)\}$. Because T is the only output relation, it suffices to show $T(\,) \in output(M)$. Abbreviate $G_M(\mathcal{P}) = ground_M(\mathcal{P}', I)$ where $\mathcal{P}' = pure_{\text{SZ}}(\mathcal{P})$ and $I = decl(H)$.

Let $y \in \mathcal{N}$ be arbitrary. We start by showing there is a timestamp s of y such that for all timestamps $t \geq s$ and all $x \in \mathcal{N}$ we have $\texttt{empty}(y, t, x) \in M$. Let $x \in \mathcal{N}$. We show that x at every local timestamp u sends $\texttt{empty}(x)$ to y. We have $S(x, u) \notin decl(H)$ by assumption on H. Hence, $S(x, u) \notin M$. Therefore, $G_M(\mathcal{P})$ contains a ground rule of the following form, obtained by applying transformation (5.10) to the asynchronous rule of $\mathcal{P}$, where $v \in \mathbb{N}$ is arbitrary:

$$\texttt{cand}_{\texttt{empty}}(x, u, y, v, x) \leftarrow \texttt{Id}(x, u, x),\ \texttt{Node}(x, u, y),\ \texttt{all}(y),\ \texttt{time}(v).$$

The body facts of this rule are in M by definition of $decl(H)$. Hence, $\texttt{cand}_{\texttt{empty}}(x, u, y, v, x) \in M$ because M is stable. Then, by Lemma Appendix C.1, there is a timestamp $w \in \mathbb{N}$ such that $\texttt{chosen}_{\texttt{empty}}(x, u, y, w, x) \in M$. Next, a ground rule of the form (5.8) derives $\texttt{empty}(y, w, x) \in M$, and inductive ground rules for relation

[20] In particular, each of the numbers $1, \ldots, n^k$ has a logarithmic representation size in n.

$\texttt{empty}$ will derive $\texttt{empty}(y, w', x) \in M$ for all $w' \geq w$. Because x is arbitrary in the above reasoning, there is a timestamp s on which $\texttt{empty}(y, s, x) \in M$ for *each* $x \in \mathcal{N}$.

Now we can show $T(\,) \in \mathit{output}(M)$. Let y and s be from above. It suffices to show for each $t \geq s$ that $\texttt{missing}(y, t) \notin M$, because then $G_M(\mathcal{P})$ contains the following ground rule, based on the last deductive rule of $\mathcal{P}$:

$$T(y, t) \leftarrow \texttt{Id}(y, t, y).$$

We show $G_M(\mathcal{P})$ contains no rule with head $\texttt{missing}(y, t)$. Towards a contradiction, if $G_M(\mathcal{P})$ would contain a ground rule with head-fact $\texttt{missing}(y, t)$, then it has the following form, where $x \in \mathcal{N}$ is arbitrary:

$$\texttt{missing}(y, t) \leftarrow \texttt{Node}(y, t, x).$$

The presence of this rule would imply $\texttt{empty}(y, t, x) \notin M$, which is impossible by selection of s.

Appendix C.1.2. Nonempty Input

Let H be an input for $\mathcal{P}$ over a network $\mathcal{N}$ that assigns $S(\,)$ to some $z \in \mathcal{N}$. So, $\mathit{outInst}(\mathcal{P}, H) = \emptyset$. Let M be an SZ-model of $\mathcal{P}$ on H. We have to show $\mathit{output}(M) = \emptyset$. Abbreviate $G_M(\mathcal{P}) = \mathit{ground}_M(\mathcal{P}', I)$ where $\mathcal{P}' = \mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ and $I = \mathit{decl}(H)$.

Towards a proof by contradiction, suppose $\mathit{output}(M) \neq \emptyset$. This means $T(\,) \in \mathit{output}(M)$ because T is the only output relation. We will show z has an empty relation S, which is the desired contradiction. First, $T(\,) \in \mathit{output}(M)$ implies there is a node $x \in \mathcal{N}$ and a local timestamp s of x, such that $T(x, t) \in M$ for all $t \geq s$. We start by showing $\texttt{empty}(x, s, z) \in M$. The following ground rule must be in $G_M(\mathcal{P})$ to derive $T(x, s) \in M$:

$$T(x, s) \leftarrow \texttt{Id}(x, s, x).$$

The existence of this rule implies $\texttt{missing}(x, s) \notin M$. Now, if $\texttt{empty}(x, s, z) \notin M$ then the following ground rule would be in $G_M(\mathcal{P})$:

$$\texttt{missing}(x, s) \leftarrow \texttt{Node}(x, s, z).$$

Then, since $\texttt{Node}(x, s, z) \in \mathit{decl}(H)$, we would have $\texttt{missing}(x, s) \in M$, which is false. So, $\texttt{empty}(x, s, z) \in M$.

Now we show relation S is empty at z. The fact $\texttt{empty}(x, s, z) \in M$ can only be explained by ground rules in $G_{\mathcal{P}}(\mathcal{P})$ of the following two forms, where the first one is obtained by applying transformation (5.8) to the asynchronous rule of $\mathcal{P}$ and the second one is based on the inductive rule of $\mathcal{P}$:

$$\texttt{empty}(x, s, z) \leftarrow \texttt{chosen}_{\texttt{empty}}(y, t, x, s, z).$$

$$\texttt{empty}(x, s, z) \leftarrow \texttt{empty}(x, r, z), \texttt{tsucc}(r, s).$$

Intuitively, the second form is like a chain we can follow backwards in time, locally at node x. So we must eventually use the first form: there is a value $u \in \mathbb{N}$ such that $\texttt{empty}(x, u, z) \in M$ and $\texttt{chosen}_{\texttt{empty}}(y, t, x, u, z) \in M$ for some $y \in \mathcal{N}$ and $t \in \mathbb{N}$. We have $y = z$ because the sender sends its own identifier. Now, the fact $\texttt{chosen}_{\texttt{empty}}(z, t, x, u, z)$ was derived by a ground rule in $G_M(\mathcal{P})$ of the form (5.6):

$$\texttt{chosen}_{\texttt{empty}}(z, t, x, u, z) \leftarrow \texttt{cand}_{\texttt{empty}}(z, t, x, u, z).$$

Hence, $\texttt{cand}_{\texttt{empty}}(z, t, x, u, z) \in M$. This $\texttt{cand}_{\texttt{empty}}$-fact is derived by a ground rule in $G_M(\mathcal{P})$ obtained by applying transformation (5.10) to the asynchronous rule of $\mathcal{P}$:

$$\texttt{cand}_{\texttt{empty}}(z, t, x, u, z) \leftarrow \texttt{Id}(z, t, z), \texttt{Node}(z, t, x), \texttt{all}(x), \texttt{time}(u).$$

The existence of this rule in $G_M(\mathcal{P})$ implies $S(z, t) \notin M$ and thus $S(z, t) \notin \mathit{decl}(H)$, which by definition of $\mathit{decl}(H)$ implies that S is empty at z.

Appendix C.2. Only-if Direction

Let $\mathcal{P}$ be the program in Figure 4. Let H be the input over singleton network $\mathcal{N} = \{z\}$ that assigns $S(\,)$ to z. So, $\mathit{outInst}(\mathcal{P}, H) = \{T(\,)\}$. We define an SZ-model M of $\mathcal{P}$ on H such that $\mathit{output}(M) = \emptyset$, showing $\mathcal{P}$ does not tolerate non-causality.

In any fair run of $\mathcal{P}$ on H, message $B(\,)$ always arrives *after* message $A(\,)$ at z. In M we will not respect this causality: we let node z send $B(\,)$ into the past, before any $A(\,)$ has arrived, thus erasing the chance of having relations A and B nonempty simultaneously. Formally, we define

$$M = \mathit{decl}(H) \cup M_A^{\text{snd}} \cup M_A^{\text{rcv}} \cup M_B^{\text{snd}} \cup M_B^{\text{rcv}},$$

where

$$\begin{aligned} M_A^{\text{snd}} = {} & \{\texttt{cand}_A(z,u,z,v) \mid u,v \in \mathbb{N}\} \\ & \cup \{\texttt{chosen}_A(z,u,z,u+1) \mid u \in \mathbb{N}\} \\ & \cup \{\texttt{other}_A(z,u,z,v) \mid u \in \mathbb{N},\, v \in \mathbb{N},\, v \neq u+1\}; \end{aligned}$$

$$M_A^{\text{rcv}} = \{A(z,u) \mid u \in \mathbb{N},\, u \geq 1\};$$

$$\begin{aligned} M_B^{\text{snd}} = {} & \{\texttt{cand}_B(z,1,z,u) \mid u \in \mathbb{N}\} \\ & \cup \{\texttt{chosen}_B(z,1,z,0)\} \\ & \cup \{\texttt{other}_B(z,1,z,u) \mid u \in \mathbb{N},\, u \neq 0\} \\ & \cup \{\texttt{sent}_B(z,u) \mid u \in \mathbb{N},\, u \geq 2\}; \end{aligned}$$

$$M_B^{\text{rcv}} = \{B(z,0)\}.$$

Intuitively, set M_A^{snd} expresses that $A(\,)$ is sent on every timestamp of z and this message arrives already on the next timestamp. Set M_A^{rcv} expresses that $A(\,)$ is available starting at timestamp 1. Note, the effect of the inductive rule for relation A is made invisible by these tight message deliveries. Set M_B^{snd} expresses that precisely one $B(\,)$ is sent on timestamp 1, which is when the first $A(\,)$ is delivered. Set M_B^{rcv} expresses that the single $B(\,)$ arrives on timestamp 0, violating the causal relationship with the message $A(\,)$ on timestamp 1.

Note, M is locally finite as required by Section 5.2.2. Also, because M contains no T-facts, we have $\mathit{output}(M) = \emptyset$ as desired.

Next, we show that M is a stable model of $\mathit{pure}_{\text{SZ}}(\mathcal{P})$ on $\mathit{decl}(H)$. We only provide a sketch; the technical details can be filled in with straightforward arguments. Abbreviate $\mathcal{P}' = \mathit{pure}_{\text{SZ}}(\mathcal{P})$ and $I = \mathit{decl}(H)$. Consider the ground program $G = \mathit{ground}_M(\mathcal{P}', I)$. To show that M is stable, we have to show $M = G(I)$. This consists of two inclusions $M \subseteq G(I)$ and $G(I) \subseteq M$, that we handle below.

Appendix C.2.1. First Inclusion

To show $M \subseteq G(I)$, the idea is to argue for each fact of M that there is a ground rule in G to derive it on input I. For example, consider $\texttt{cand}_A(z,u,z,v) \in M$. Consider the following ground rule, obtained by applying transformation (5.10) to the asynchronous rule for relation A:

$$\texttt{cand}_A(z,u,z,v) \leftarrow S(z,u), \texttt{Id}(z,u,z), \texttt{all}(z), \texttt{time}(v).$$

This rule is in G because it is positive and uses values from I. Moreover, this rule derives $\texttt{cand}_A(z,u,z,v) \in G(I)$ because its body facts are in $\mathit{decl}(H)$. For the other facts of M, because some rules of $\mathit{pure}_{\text{SZ}}(\mathcal{P})$ contain negative body literals, the existence of a ground rule in G has to be argued by checking that certain facts are purposely omitted from M. Moreover, in order to derive their head, these ground rules could require the presence of certain facts of M in $G(I)$; to achieve this, the inclusion of facts of M in $G(I)$ should be argued in the order that they are given above.

Appendix C.2.2. Second Inclusion

To show $G(I) \subseteq M$, we consider the fixpoint computation of $G(I)$. Concretely, there is an infinite sequence of sets N_0, N_1, N_2, etc, so that $G(I) = \bigcup N_i$, with $N_0 = I$ and N_i for each $i \geq 1$ is obtained by applying the immediate consequence operator of G to N_{i-1}. This implies $N_{i-1} \subseteq N_i$ for each $i \geq 1$. We show by induction on $i = 0, 1, \ldots$, that $N_i \subseteq M$. For the base case, we immediately have $N_0 = I \subseteq M$ by definition of M. For the inductive step, we show $N_i \subseteq M$, where $i \geq 1$. We illustrate the approach for one relation. Let $\texttt{cand}_A(x, u, y, v) \in N_i \setminus N_{i-1}$. This fact can only be derived by a ground rule in G of the following form, based on applying transformation (5.10) to the asynchronous rule for relation A:

$$\texttt{cand}_A(x, u, y, v) \leftarrow S(x, u), \texttt{Id}(x, u, y), \texttt{all}(y), \texttt{time}(v).$$

By the induction hypothesis, the body facts are in $N_{i-1} \subseteq M$. More specifically, because there are no rules in $pure_{\text{SZ}}(\mathcal{P})$ with head predicate S, `Id`, `all`, or `time`, the body facts are in I. So, $x = y = z$ and $u, v \in \mathbb{N}$. Hence, we have explicitly added $\texttt{cand}_A(x, u, y, v) \in M_A^{\text{snd}} \subseteq M$, as desired.

Appendix D. Syntactical CRON

Proof of Claim 6.2

We proceed by induction on the fixpoint computation of $deduc_{\mathcal{P}}$. Formally, $deduc_{\mathcal{P}}(I) = \bigcup_j D^j$ where $D^0 = I$ and for each $j \geq 1$, the set D^j is obtained by applying the immediate consequence operator of $deduc_{\mathcal{P}}$ to D^{j-1}. For the base case, we have $D^0 = I \subseteq all_M(x, s)$ by the given assumption.

For the induction hypothesis, we assume $D^{j-1} \subseteq all_M(x, s)$ with $j \geq 1$. For the inductive step, let $R(\bar{a}) \in D^j \setminus D^{j-1}$. We show $R(x, s, \bar{a}) \in M$. We first establish the existence of a ground rule ψ with $head_\psi = R(x, s, \bar{a})$ in the ground program $G_M(\mathcal{P}) = ground_M(\mathcal{P}', J)$ where $\mathcal{P}' = pure_{\text{SZ}}(\mathcal{P})$ and $J = decl(H)$. Let $\varphi \in deduc_{\mathcal{P}}$ and V be a rule and valuation that have derived $R(\bar{a}) \in D^j$. Let $\varphi' \in pure_{\text{SZ}}(\mathcal{P})$ be obtained by applying transformation (5.3) to φ. Let V' be V extended to assign x and s to respectively the location variable and timestamp variable of φ'. Let ψ be the ground rule based on φ' and V'. Note, $head_\psi = R(x, s, \bar{a})$. Moreover, because φ is positive by assumption on $\mathcal{P}$, rule ψ is also positive; hence, $\psi \in G_M(\mathcal{P})$.

Lastly, we show $pos_\psi \subseteq M$, which, together with $\psi \in G_M(\mathcal{P})$, implies $head_\psi = R(x, s, \bar{a}) \in M$.[21] Since $V(pos_\varphi) \subseteq D^{j-1} \subseteq all_M(x, s)$ by the induction hypothesis, we have $pos_\psi = V(pos_\varphi)^{\Uparrow x,s} \subseteq all_M(x, s)^{\Uparrow x,s} \subseteq M$. □

Proof of Claim 6.3

This is similar to the proof of Claim 6.2. Let $R(\bar{a}) \in induc_{\mathcal{P}}(D)$. We show $R(x, s+1, \bar{a}) \in M$. Let φ and V be a rule and valuation deriving $R(\bar{a}) \in induc_{\mathcal{P}}(D)$. Let $\varphi' \in pure_{\text{SZ}}(\mathcal{P})$ be obtained by applying transformation (5.4) to φ. Let V' be the extension of V to assign x to the location variable and to assign s and $s+1$ to the timestamp variable in respectively the body and head. Let ψ denote the ground rule based on φ' and V'. Note, $head_\psi = R(x, s+1, \bar{a})$. Abbreviate $G_M(\mathcal{P}) = ground_M(\mathcal{P}', J)$ where $\mathcal{P}' = pure_{\text{SZ}}(\mathcal{P})$ and $J = decl(H)$. We have $\psi \in G_M(\mathcal{P})$ because ψ is positive; indeed, φ is positive by assumption on $\mathcal{P}$, and, transformation (5.4) does not introduce negative literals. We are left to show $pos_\psi \subseteq M$. Since $V(pos_\varphi) \subseteq D$ and $D \subseteq all_M(x, s)$ by the given assumption, we have $pos_\psi = V(pos_\varphi)^{\Uparrow x,s} \cup \{\texttt{tsucc}(s, s+1)\} \subseteq M$. □

Proof of Claim 6.4

Necessarily, $R(y, \bar{a}) \in async_{\mathcal{P}}(D_i)$. Suppose we would know $D_i \subseteq all_M(x_i, t)$ for each $t \geq s_i$. Then Claim 6.5 would imply that for each $t \geq s_i$ there is a value u such that $\texttt{chosen}_R(x_i, t, y, u, \bar{a}) \in M$, as desired.

Now, we show by induction on $j \in S$ that

$$D_j \subseteq all_M(x_j, t) \text{ for all } t \geq s_j.$$

[21] Letting $J = decl(H)$, we use that $M = G_M(\mathcal{P})(J)$ by definition of stable model.

For each $j \in S$, let $\rho_j = (st_j, bf_j)$ denote the source configuration of transition j, and let m_j denote the set of (tagged) messages delivered in transition j. Since $D_j = deduc_{\mathcal{P}}(st_j(x_j) \cup untag(m_j))$ by the operational semantics, Claim 6.2 implies that it is sufficient to show for each $j \in S$ that

$$st_j(x_j) \cup untag(m_j) \subseteq all_M(x_j, t) \text{ for all } t \geq s_j.$$

As additional simplification, $st_j(x_j) \cup untag(m_j) = st_j(x_j)$ because j is a heartbeat transition. In the base case, $j = \min(S)$, i.e., j is the first transition of x_j. So, $st_j(x_j) = H(x_j)$ by the operational semantics; hence, $decl(H) \subseteq M$ implies $st_j(x_j) \subseteq all_M(x_j, t)$ for all t. For the induction hypothesis, let $j \in S$ with $j > \min(S)$, and we assume for all $k \in S$ with $k < j$ that

$$st_k(x_k) \subseteq all_M(x_k, t) \text{ for all } t \geq s_k.$$

For the inductive step, we show $st_j(x_j) \subseteq all_M(x_j, t)$ for all $t \geq s_j$. Let k be the predecessor of j in S (which exists because $j > \min(S)$). By the operational semantics, $st_j(x_j) = H(x_k) \cup induc_{\mathcal{P}}(D_k)$. The inclusion of $H(x_k)$ in M is established as in the base case. Next, the induction hypothesis on k gives $st_k(x_k) \subseteq all_M(x_k, u)$ for all $u \geq s_k$. Hence, by Claim 6.2 we have $D_k \subseteq all_M(x_k, u)$ for all $u \geq s_k$. Then Claim 6.3 gives $induc_{\mathcal{P}}(D_k) \subseteq all_M(x_k, u+1)$ for all $u \geq s_k$. Using $st_j(x_j) = H(x_k) \cup induc_{\mathcal{P}}(D_k)$, $x_j = x_k$, and $s_j = s_k + 1$, we can thus say $st_j(x_j) \subseteq all_M(x_j, t)$ for all $t \geq s_j$. □

Proof of Claim 6.5

Let $R(y, \bar{a}) \in async_{\mathcal{P}}(D)$ with $y \in \mathcal{N}$, derived by a rule φ and valuation V. By Lemma Appendix C.1, it suffices to show $\texttt{cand}_R(x, s, y, u, \bar{a}) \in M$ for some $u \in \mathbb{N}$.

Let $\varphi' \in \mathcal{P}$ be the original rule on which φ is based. Let $\varphi'' \in pure_{\text{SZ}}(\mathcal{P})$ be the result of applying transformation (5.10) to φ'. Note, φ'' is positive because φ' is positive. Let V'' be the extension of V to assign x and s to respectively the sender variable and send-timestamp variable of φ'', and to assign some arbitrary $u \in \mathbb{N}$ to the arrival-timestamp variable of φ''. Let ψ be the ground rule based on φ'' and V''. Note, $head_\psi = \texttt{cand}_R(x, s, y, u, \bar{a})$. Because ψ is positive, we have $\psi \in ground_M(\mathcal{P}', I)$ where $\mathcal{P}' = pure_{\text{SZ}}(\mathcal{P})$ and $I = decl(H)$. It remains to be shown that $pos_\psi \subseteq M$, so that $head_\psi \in M$. Transformation (5.10) implies $pos_\psi = V(pos_\varphi)^{\Uparrow x,s} \cup \{\texttt{all}(y), \texttt{time}(u)\}$. First, we have $\{\texttt{all}(y), \texttt{time}(u)\} \subseteq decl(H) \subseteq M$. Second, $V(pos_\varphi)^{\Uparrow x,s} \subseteq D^{\Uparrow x,s} \subseteq all_M(x, s)^{\Uparrow x,s} \subseteq M$. □